

# **Project documentation**

for

# **DeepGreen**

**Prepared by:**

Chu Shun Yuan

**25/07/2025**

# 1. Introduction

## 1.1 Purpose

This document provides the project documentation for DeepGreen website application. It serves as a comprehensive guide for all stakeholders, detailing the application's description, interface requirements, system features, and other relevant system considerations. It serves as a reference for developers, testers, and stakeholders to ensure a shared understanding of the system's intended behaviour and design and applies specifically to the current release of the DeepGreen application.

## 1.2 Intended Audience and Reading Suggestions

This document is intended for all stakeholders of the DeepGreen application, including two primary user groups: general users and system administrators, including project team members such as developers and testers.

**General users:** Our main target users are those who want to quickly retrieve accurate information, access summarized knowledge, real-time answers from a vast knowledge base and understand the sentiment of various market sectors. Relevant sections for information are listed below:

- 1. Introduction
- 2.5 User Documentation
- 3.1 User Interface
- 4. System Features

**App Developers:** To understand app functionality, logic, and integrations. It would be beneficial to review the entire document. However, developers can focus on development-focused sections in the following sequence

- 2.5 User Documentation
- 3. External Interface Requirements

- 4. System Features
- 5. Code Structure

**Project Managers:** Oversee the project's development and ensure alignment with Financial-tech goals. They can focus on the user-focused sections in the following sequence

- 2. Overall Description
- 3. External Interface Requirements
- 4. System Features

### 1.3 Product Scope

DeepGreen is an AI-powered knowledge retrieval platform developed to align with the goal of enhanced information accessibility and intelligent data utilization. Its primary purpose is to enhance access to diverse knowledge and insights, while empowering users to efficiently retrieve, synthesize, and understand complex information. By offering a one-stop solution for information retrieval and understanding, DeepGreen supports the broader objective of promoting timely, accurate, and informed decision-making. The app also reflects the project's goal of leveraging advanced AI and digital tools to democratize access to critical knowledge, fostering both informed productivity and continuous learning.

### 1.4 References

1. LangChain:

<https://www.langchain.com/>

2. Chroma:

<https://docs.trychroma.com/docs/overview/introduction>

3. Unstructured:

<https://docs.unstructured.io/self-hosted/overview>

4. Ollama:

<https://github.com/ollama/ollama>

5. Docker:

<https://www.docker.com/>

6. yfinance:

<https://github.com/ranaroussi/yfinance>

7. API Ninja:

<https://api-ninjas.com/>

## 2. Overall Description

### 2.1 Product Perspective

**DeepGreen** is a new, self-contained AI-powered financial analysis platform which integrates advanced AI capabilities for financial data analysis and insight generation into one seamless and powerful tool.

Through DeepGreen, financial analysts, investors, and researchers at Albizia Capital will be able to rapidly access and analyse vast amounts of financial information from anywhere, allowing them to gain timely and reliable insights into market trends, company performance, and investment opportunities. Users will be able to engage with the platform via a conversational interface, querying market data, earnings reports, and other financial documents to obtain concise answers and summarized insights. This allows users to get a deeper, data-driven understanding of financial dynamics before making critical investment or strategic decisions, significantly reducing manual research time and potentially highlighting unforeseen risks or lucrative opportunities. Furthermore, users can upload new financial documents for ingestion, continuously expanding the system's knowledge base and enabling up-to-date sentiment analysis. This allows the user to have a more efficient and informed financial analysis journey, streamlining research workflows and enhancing decision-making.

### 2.2 Product Functions

The DeepGreen application is designed to serve its primary users: financial analysts, investors, and researchers at Albizia Capital. The following section summarizes the key functionalities that the platform will provide to its users.

- **Chat Interface**

- Users can ask natural language questions about financial data, company reports, market trends, and economic indicators via a conversational interface.
  - Users can obtain concise answers, summaries, and extracted insights from the underlying financial knowledge base.
  - Users can request synthesis of information from multiple documents or data points.
- **Upload & Ingest**
    - Users can upload various financial documents (e.g., earnings reports, analyst call transcripts, press releases, research papers) for ingestion into the knowledge base.
    - The system processes and indexes uploaded documents, making their content retrievable and analysable.
- **Sentiment Analysis**
    - Users can view and analyse sentiment scores related to specific companies, sectors, market events, or news articles.
    - Users can track historical sentiment trends to identify shifts in market perception.

## 2.3 Operating Environment

### 2.3.1 Production Environment of DeepGreen

This subsection describes the setting in which the DeepGreen application is put into operation.

Setting	Description
Web Browser Support	The DeepGreen application is optimized for modern web browsers including, but not limited to, Google Chrome (latest two major versions), Mozilla Firefox (latest two major versions), Microsoft Edge (latest two major versions), and Apple Safari (latest two major versions).

Responsive Design	<p>The DeepGreen application utilizes responsive web design principles to ensure optimal viewing and interaction across a variety of screen sizes and devices (desktops, laptops, tablets, and mobile browsers).</p> <ul style="list-style-type: none"> <li>- <b>Flexible Layouts:</b> Achieved through CSS Flexbox and Grid.</li> <li>- <b>Relative Units:</b> Usage of rem, em, vw, vh, and percentages for dimensions and font sizes instead of fixed pixel values.</li> </ul>
Connectivity Requirements	<p>The DeepGreen application is designed to operate primarily within a Local Area Network (LAN) environment for its core runtime functionalities. Once the application, its dependencies, and the open-source AI models are initially set up and downloaded, an active and stable internet connection is not required. To refresh sentiment analysis, an active and stable internet connection will be required.</p>

### 2.3.2 Development Environment of DeepGreen

This subsection describes the setting in which the DeepGreen application is built and tested during the development phase.

Setting	Description
Development using Visual Studio Code	Visual Studio Code (VS Code) is the primary Integrated Development Environments (IDEs) used for DeepGreen's development. They provide comprehensive features for coding, debugging, version control integration, and extension support to streamline the development workflow for both frontend and backend components.
Frontend Development using HTML, CSS, JavaScript	HTML, CSS, and JavaScript are used for the web-based user interface. Modern JavaScript frameworks (e.g., React, Vue.js, or Svelte) are employed to build dynamic, responsive, and interactive user experiences.

Backend Development using Python & FastAPI	Python is the core programming language for the backend, leveraging its extensive libraries for AI/ML, data processing, and web development. FastAPI is used to build the backend RESTful API services, chosen for its high performance, automatic interactive API documentation (Swagger UI), and robust data validation features. <i>Edition: Python (version 3.9+)</i>
Vector Database	ChromaDB is used as the primary knowledge base for storing vector embeddings of financial documents. This allows for efficient semantic search and retrieval of relevant information for the RAG process.
HTTP Clients	Requests is used for making asynchronous HTTP requests to external APIs, including financial data providers.
Containerization	Docker is used to containerize the application's services (frontend, backend, database). This ensures consistent development, testing, and production environments, simplifying deployment and dependency management.

## 2.4 Design and Implementation Constraints

This section covers all constraints and pitfalls which have limited or will limit options available to the DeepGreen application, both during the initial development stage and any further development stages to update or enhance the application.

### 2.4.1 Front-End Constraints

Financial analysts typically require highly precise, data-dense, and highly configurable interfaces. They expect robust data tables, sophisticated charting tools, and rapid interaction. Consumer-grade UI/UX might not suffice.

## **2.4.2 Back-End Constraints**

### *2.4.2.1 Cost of operation*

OCR procedures for PDF parsing requires high compute power to be efficient, which necessitates high performing hardware that is costly. An alternative to this is to utilise PDF parsing API(Adobe etc) from service providers, but this will add to the cost as well as potential compromise of confidential financial documents. Additionally, making over 10,000 API calls a month to obtain transcripts requires an active subscription to API Ninja, adding onto the operational costs.

### *2.4.2.2 Security*

Backend communication must use HTTPS and TLS protocols exclusively to ensure encrypted data transfer. Sensitive data (including financial information, and proprietary knowledge base content) must be encrypted at rest (e.g., database encryption, encrypted file storage) and during data transfer (via TLS).

### *2.4.2.3 API Dependencies*

All external API integrations (e.g., Financial Market Data APIs like StockAnalysis API, API Ninja) must conform to their respective API documentation, Service Level Agreement (SLA) constraints and its limitations. For example, with a free subscription to API ninja, users will not be able to retrieve transcripts from previous years.

Dependency on external APIs introduces potential single points of failure, necessitating comprehensive contingency strategies such as fallback mechanisms (e.g., using cached data), to maintain application stability, reliability, and data accuracy, especially for critical financial insights.

### *2.4.2.4 Duration of processing*

Despite architectural optimizations and resource allocation, the inherent computational complexity and multi-stage nature of the Retrieval-Augmented Generation (RAG) pipeline introduce specific limitations regarding processing duration within the DeepGreen application.

**Batch Sentiment Analysis Latency:** The comprehensive sentiment analysis process for financial data is a particularly resource-intensive, multi-step operation. This involves:



1. **Extraction and Data Aggregation:** Identifying and extracting approximately 5500 company tickers, along with their respective sectors and market capitalization.
2. **Document Retrieval:** Subsequently retrieving two quarters of associated earnings transcripts for each identified company.
3. **LLM-Based Sentiment Scoring:** Utilizing a Large Language Model (LLM) to meticulously analyze the sentiment of each individual transcript and generate a corresponding score.

This extensive data volume, combined with sequential processing and intensive LLM inference, translates into a significant compute time, estimated at approximately 12 hours per full analysis cycle. This duration currently prevents users from accessing the latest sentiment results for immediate, real-time market analysis, thereby imposing a constraint on the responsiveness and timeliness of sentiment insights available within the application.

#### *2.4.2.5 Inconsistency of sentiment analysis*

There is an inherent variability in LLM interpretation, translating into a limitation on the **uniformity and comparability of sentiment scores** across different companies or over various quarters. While the analysis provides valuable insights, users should be aware that perfect apples-to-apples comparisons of sentiment scores may be hindered by these inconsistencies, potentially requiring additional human review or contextual understanding for definitive conclusions. Additionally, while Large Language Models (LLMs) are highly adept at contextual understanding, the lack of standardized formatting of answers may lead to some scores being omitted from the calculation of sentiment change for certain companies

## **2.5 User Documentation (Developer Setup Guide)**

To facilitate project setup and contribution for new developers, a comprehensive README.md file is maintained in our GitHub Repository. This document outlines the essential steps required to get the DeepGreen application running on a local development machine. The following summarizes the main setup procedures:

### **2.5.1 Set Up**

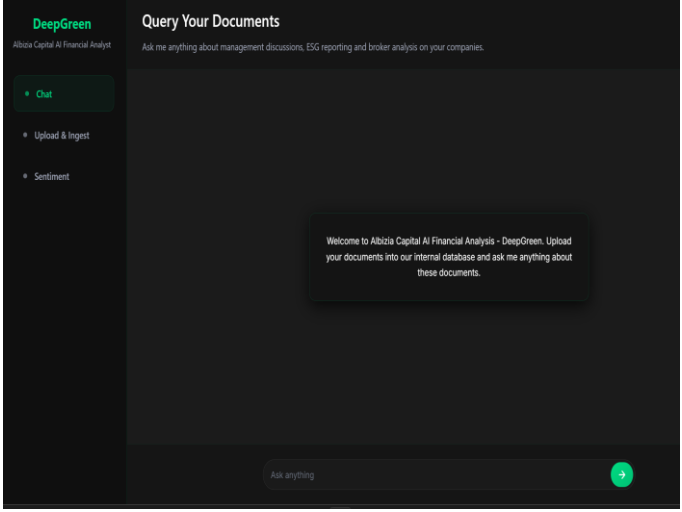
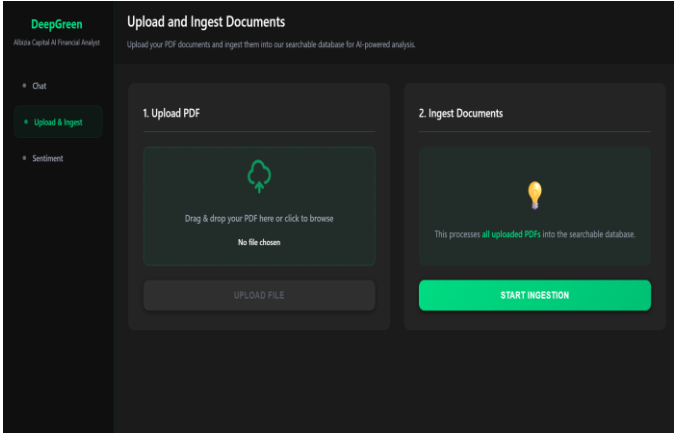
1. Download Docker

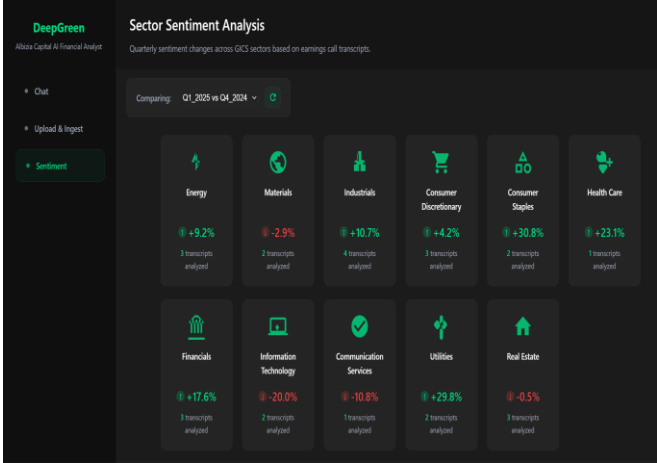
2. Download Visual Studio Code
3. Download Ollama
4. Open the Command Prompt and pull AI models (one embedding model and one LLM model) using the following command “ollama pull (model-name)”
5. Clone the GitHub Repository: Clone the DeepGreen project repository onto your local machine using Git.
6. Configure Environment Variables: Create a .env file in the root directory and populate it with necessary API keys and the Ollama models you downloaded.
7. Make a directory named “data”, “logs” and “chroma\_db”
8. Edit the Dockerfile based on the operating system (Windows or Mac)
9. Build the Dockerfile using the following command in the terminal “docker build -t (your-docker-username)/(name-of-the- app):latest .”
10. Run the built image in a container using the following command “docker run -d --name (name-to-give-your-container) --gpus all -p 80:80 -p 11434:11434 -p 8000:8000 -v ./data:/app/data -v ./chroma\_db:/app/chroma\_db -v ./logs:/app/logs -v ./env:/app/.env (your-docker-username)/(name-of-the- app):latest”
11. Access in Browser: Open your web browser and navigate to the local development URL (http://localhost:80 or similar) to confirm the application is running.
12. Deploy for internal usage by making sure the local device is connected to LAN cable, locate the IP address and share this URL, http://YOUR\_IP\_ADDRESS

### 3. External Interface Requirements

#### 3.1 User Interfaces

The DeepGreen Web app is developed using standard web technologies (HTML, CSS, and JavaScript). The UI adheres to a clean, professional, and intuitive web design system, prioritizing clarity and data density for its target audience of financial analysts.

	<h4>Chat Page</h4> <ul style="list-style-type: none"><li>• Users will key in their question in the text box</li><li>• A green send button is beside the text box for users to send their query</li><li>• Consistent green styling for primary actions</li></ul>
	<h4>Upload and Ingest Page</h4> <ul style="list-style-type: none"><li>• Users click on ‘Drag &amp; drop box’ or drop their files into the container</li><li>• The greyed-out “UPLOAD FILE” button will be activated and turn green. It is now ready for upload.</li><li>• Users press the upload button</li><li>• The documents are now ready to be ingested into the database</li><li>• Users will press the “START INGESTION” button and</li></ul>

	<p>documents will be added into the database</p> <ul style="list-style-type: none"> <li>Consistent green styling for primary actions</li> </ul>
	<p>Sentiment Page</p> <ul style="list-style-type: none"> <li>An interface consisting of 11 cards, showing the sentiment analysis for the 11 GICS sector.</li> <li>Period filters (Drop down box for different time periods) allow users to view the analysis for particular periods.</li> <li>Refresh button gives users the option to refresh the analysis for the current latest period or start the analysis for the next period.</li> <li>Consistent green styling for primary actions</li> </ul>

## 3.2 Software Interfaces

### 3.2.1 APIs and Data

DeepGreen relies on multiple external APIs to enable real-time communication, data retrieval, and user interaction across different software components. These communications follow standardised RESTful API protocols over HTTPS to ensure data security and consistency.

1. API Ninja (<https://api-ninjas.com/api/earningscalltranscript>):

This API provides updated full transcripts of earnings calls for major companies every quarter. The retrieved transcripts is used by the LLM to generate a sentiment analysis. The app sends a GET request containing Company ticker symbol, year and quarter and receives a JSON response with full transcript.

## 2. Stock Analysis API :

This API is used to obtain the tickers in the US stock market as well as its GICS sector. The retrieved tickers and sectors is used to fetch the transcripts from API Ninja and the GICS sector is used to group companies by sector. The app sends a GET request receives a JSON response with tickers and sectors.

### **3.2.2 Database**

The DeepGreen application employs database strategy designed to efficiently manage data types essential for its Retrieval-Augmented Generation (RAG) capabilities and overall application functionality.

DeepGreen primarily utilizes ChromaDB as its vector database. This specialized NoSQL database is optimized for storing high-dimensional vector embeddings, which represent the semantic meaning of financial documents and queries. ChromaDB is central to the RAG pipeline, enabling highly efficient semantic search and similarity retrieval. It allows the application to quickly find the most relevant document chunks from the vast knowledge base based on the user's natural language queries. All sensitive data residing in ChromaDB is subject to encryption.

## 4. System Features

### 4.1 Chat

#### 4.1.1 Description and Priority

**Description:** Allows the user to input and ask questions regarding the documents stored in the database

**Priority:** High

#### 4.1.2 Stimulus/Response Sequences

**Use Case ID:** SCM01

##### **Flow of Events**

1. The user navigates to the “Chat” tab.
2. The user clicks on a chat box on the bottom of the screen.
3. The user types in their question in the chat box.
4. User clicks the send button.
5. User receives the answer to the question

##### **Alternate Flows:**

AF01-SCM01: The user does not enter anything on the chatbox.

1. The user presses the send button without anything in the chat box.
2. The system displays an error message asking the user to enter a question

### 4.2 Upload & Ingest

#### 4.2.1 Description and Priority

**Description:** Allows the user upload and ingest new documents into the database

**Priority:** High

#### 4.2.2 Stimulus/Response Sequences

**Use Case ID:** SCM02

**Flow of Events**

1. The user navigates to the “Upload & ingest” tab.
2. The user clicks on a drop box and select files to upload or drags local files into the drop box
3. The user clicks the upload button.
4. User clicks the ingest button.

**Alternate Flows:**

AF01-SCM02: The user chose a file that is not in PDF format.

1. The user uploads a file in a non-PDF format into the drop box.
2. The system displays an error message informing the user that only PDF is allowed to be ingested

### 4.3 Sentiment analysis

#### 4.3.1 Description and Priority

**Description:** Allows the user view or refresh the sentiment analysis for the latest period

**Priority:** High

#### 4.3.2 Stimulus/Response Sequences

**Use Case ID:** SCM03

**Flow of Events**

1. The user navigates to the “Sentiment” tab
2. User views the sentiment values for each sector
3. User presses the refresh button
4. User presses the “Refresh Now” button

## 4.4 Retrieve sentiment analysis for next period

### 4.3.1 Description and Priority

**Description:** Allows the user to retrieve the sentiment analysis for the next period

**Priority:** High

### 4.3.2 Stimulus/Response Sequences

**Use Case ID:** SCM04

#### **Flow of Events**

1. The user navigates to the “Sentiment” tab
2. User views the sentiment values for each sector
3. User presses the refresh button
4. User presses the “Analyse Latest Quarter” button

Alternative Flows:

AF01-SCM04: The user chose a file that is not in PDF format.

1. The user starts analysis before the allowed period.
2. The system displays an error message informing the user that analysis is only allowed after a specified date



## 5. Code Structure

### 5.1 Chat functionality Core Code Documentation

This code snippet outlines the essential components and logic that power DeepGreen's Retrieval-Augmented Generation (RAG) chat interface, enabling users to interact with the financial knowledge base conversationally.

#### 5.1.1 Initialisation of LLM model and vector store

```
try:
    logger.info(f"Initializing OllamaEmbeddings with model: {OLLAMA_EMBEDDING_MODEL} and base URL: {OLLAMA_BASE_URL}")
    embeddings = OllamaEmbeddings(model=OLLAMA_EMBEDDING_MODEL, base_url=OLLAMA_BASE_URL)

    logger.info(f"Initializing ChatOllama with model: {OLLAMA_LLM_MODEL} and base URL: {OLLAMA_BASE_URL}")
    llm = ChatOllama(
        model=OLLAMA_LLM_MODEL,
        base_url=OLLAMA_BASE_URL,
        temperature=0.1,
        num_ctx=16384
    )

    logger.info(f>Loading ChromaDB from {DB_DIR} with collection: {COLLECTION_NAME}")
    vector_store = Chroma(
        collection_name=COLLECTION_NAME,
        embedding_function=embeddings,
        persist_directory=DB_DIR,
    )
    retriever = vector_store.as_retriever(search_kwargs={"k": 15})
    logger.info("RAG components initialized successfully.")
except Exception as e:
    logger.error(f"Error initializing RAG components. Querying will be unavailable: {e}")
    retriever = None
    llm = None
```

1. The code attempts to initialize the embeddings model which converts text into numerical vectors.
2. It then initializes the LLM with the chosen LLM model, the URL ollama is serving on, a temperature for response creativity, and num\_ctx for context window size). This is the conversational AI model.
3. Next, it loads or initializes the vector\_store (ChromaDB), linking it to the embeddings function. This step makes the stored financial knowledge base accessible.
4. Finally, it creates the retriever object from the vector\_store, which is the component specifically designed to fetch relevant documents based on a query.

#### 5.1.2 Query API Endpoint

```

@app.post("/query/", summary="Query the RAG System")
async def query_rag(request_body: QueryRequest, http_request: Request, response: Response):
    """
    Queries the RAG system with a given text query, using the specific logic and prompt
    from your ask_with_params.py script, and maintains conversation history per user
    using a session cookie.
    """

    # 1. Session Identification: Get session_id from cookie or generate a new one
    session_id = http_request.cookies.get(SESSION_COOKIE_NAME)
    if not session_id:
        session_id = str(uuid.uuid4())
        response.set_cookie(
            key=SESSION_COOKIE_NAME,
            value=session_id,
            httponly=True, # Prevent JavaScript access to the cookie
            samesite="Lax", # Or "None" if cross-site, but requires secure=True
            secure=False # Set to True if deployed with HTTPS
        )
        logger.info(f"New session created and cookie set for ID: {session_id}")
    else:
        logger.info(f"Existing session found with ID: {session_id}")

    # 2. Get/Create ChatContext for this session ID
    chat_session = get_chat_session_by_id(session_id)

    query = request_body.query

    if retriever is None or llm is None:
        logger.error("RAG components (retriever or LLM) are not initialized. Cannot process query.")
        raise HTTPException(status_code=500, detail="RAG system is unavailable. Check server logs for initialization errors.")

    logger.info(f"Received query for session {session_id}: {query}")

    try:
        # 4. Perform RAG Retrieval
        retrieved_docs = retriever.invoke(query)
        retrieved_documents_content = [doc.page_content for doc in retrieved_docs]
        logger.info(f"retrieved_documents_content: {retrieved_documents_content[:200]}...")
        source_set = extract_sources_from_raw_chunks(retrieved_documents_content)
        source_list = list(source_set)

        # 5. Get current conversation context string from the session
        prev_convo_context_string = chat_session.get_context()
        logger.info(f"Previous conversation context for session {session_id}: {prev_convo_context_string[:200]}...")

    except Exception as e:
        logger.error(f"Error during RAG query for '{query}' (session {session_id}): {e}")
        raise HTTPException(status_code=500, detail=f"Error processing query: {e}. Ensure Ollama server is running and accessible and models ({OLLAMA_EMBEDDING_MODEL},

```

```

    OLLAMA_LLM_MODEL) are not initialized. Cannot process query.")

    retrieved_data_string = "\n\n---\n\n".join(retrieved_documents_content)
    # 7. Format the system prompt with the separated contexts
    formatted_system_prompt = SYSTEM_PROMPT_CONTENT.format(
        previous_conversation=prev_convo_context_string if prev_convo_context_string else "No previous conversation.",
        retrieved_data=retrieved_data_string
    )

    # 8. Create structured chat messages for the LLM
    messages = ChatPromptTemplate.from_messages(
        [
            SystemMessage(content=formatted_system_prompt),
            HumanMessage(content=query),
        ]
    )

    # 9. Invoke the LLM
    response_obj = llm.invoke(messages.format_messages())
    response_content = response_obj.content

    logger.info(f"Generated response for query: '{query}' for session {session_id}")
    logger.info(f"Sources: {source_list}")
    source_string = ", ".join(source_list)

    # 10. Add the LLM response to chat session history
    # Add current query to chat session history
    chat_session.add_query(query)
    response_content = chat_session._clean_llm_output(response_content) # Clean the LLM output
    chat_session.add_answer(response_content)

    return {"query": query, "response": response_content, "sources": source_string}

except Exception as e:
    logger.error(f"Error during RAG query for '{query}' (session {session_id}): {e}")
    raise HTTPException(status_code=500, detail=f"Error processing query: {e}. Ensure Ollama server is running and accessible and models ({OLLAMA_EMBEDDING_MODEL},

```

This asynchronous FastAPI endpoint is the main entry point for user queries to the RAG system.

1. Decorator: `@app.post("/query/", summary="Query the RAG System")`
  - Defines a POST endpoint at the `/query/` path.
  - The summary provides a brief description for API documentation tools (like FastAPI's Swagger UI).
2. Function Signature: `async def query_rag(request_body: QueryRequest, http_request: Request, response: Response):`
  - `request_body: QueryRequest`: Automatically validates the incoming JSON body against the `QueryRequest` Pydantic model.
  - `http_request: Request`: Provides access to the raw HTTP request, used here to read cookies.
  - `response: Response`: Allows modification of the HTTP response, used here to set cookies.
3. Core Logic Breakdown:
  1. Session Identification & Management:
    - Retrieves a `session_id` from the incoming request's cookies.
    - If no `session_id` is found, a new UUID is generated, and a new session cookie (`SESSION_COOKIE_NAME`) is set in the response. This cookie is `httponly` (not accessible by JavaScript), `samesite="Lax"` (for security against CSRF), and `secure=False`.
    - A `ChatContext` object (via `get_chat_session_by_id`) is retrieved or created for the `session_id` to manage conversation history.
  2. Retrieval Phase:
    - `retrieved_docs = retriever.invoke(query)`: The core retrieval step where the user's query is used by the retriever to find the most semantically relevant document chunks from the `vector_store`.
    - `retrieved_documents_content`: Extracts the text content from the retrieved document objects.
    - `source_set/source_list`: Extracts unique sources from the retrieved content, ensuring users know where the information came from.
  3. Context Preparation:

- `prev_convo_context_string = chat_session.get_context()`: Fetches the formatted string of previous turns in the current user's conversation.
  - `retrieved_data_string = "\n\n---\n\n".join(retrieved_documents_content)`: Concatenates the retrieved document chunks into a single string to be passed to the LLM as "The Data."
4. Prompt Engineering:
- `formatted_system_prompt = SYSTEM_PROMPT_CONTENT.format(...)`: Dynamically inserts the `prev_convo_context_string` and `retrieved_data_string` into the `SYSTEM_PROMPT_CONTENT` template.
  - `messages = ChatPromptTemplate.from_messages(...)`: Constructs the final list of messages for the LLM, comprising the `SystemMessage` (with the formatted prompt) and the `HumanMessage` (with the user's query).
5. LLM Invocation:
- `response_obj = llm.invoke(messages.format_messages())`: Sends the prepared messages to the initialized LLM for generation.
  - `response_content = response_obj.content`: Extracts the generated text response from the LLM's output.
6. History Update & Response:
- `chat_session.add_query(query)`: Adds the current user query to the session's history.
  - `response_content = chat_session._clean_llm_output(response_content)`: Cleans or post-processes the LLM's raw output.
  - `chat_session.add_answer(response_content)`: Adds the processed LLM response to the session's history.
  - Returns a JSON response containing the original query, the response from the LLM, and a string of sources.

## 5.2 Ingestion Core Code Documentation

This code snippet achieves the overall goal to incrementally populate DeepGreen's knowledge base with processed financial documents, making them available for semantic search and contextual retrieval by the RAG system.

```

# --- Main ingestion function, adapted from your provided main() ---
def ingest_documents_to_chroma(data_dir: str = DEFAULT_DATA_DIR, chroma_path: str = DEFAULT_CHROMA_PATH, use_adobe_api: bool = False, reset_collection: bool = False):
    """
    Processes documents from the specified data_dir and ingests them into ChromaDB.
    This function will add new documents to an existing collection or create a new one.
    It's designed to be called multiple times.
    """
    logger.info(f"Starting ingestion process from {data_dir} to {chroma_path}...")

    if not os.path.exists(data_dir):
        logger.warning(f"Warning: Data directory {data_dir} does not exist. No documents to load.")
        return

    def get_pdf_file_paths(data_path):
        """
        Most efficient method to get PDF file paths
        Uses os.scandir() - fastest and most memory efficient for file discovery
        """
        try:
            with os.scandir(data_path) as entries:
                return [os.path.join(data_path, entry.name)
                        for entry in entries
                        if entry.is_file() and entry.name.lower().endswith('.pdf')]
        except FileNotFoundError:
            logger.error(f"Directory not found: {data_path}")
            return []

    file_paths = get_pdf_file_paths(data_dir)

    if not file_paths:
        logger.info("No PDF files found to process. Exiting ingestion.")
        return
    else:
        logger.info(f"Processing {len(file_paths)} PDF files...")

```

```

for i, file_path in enumerate(file_paths):
    file_basename = os.path.basename(file_path)
    logger.info(f"\n{' '*60}")
    logger.info(f"PROCESSING FILE {i+1}/{len(file_paths)}: {file_basename}")
    logger.info(f"{' '*60}")

    if file_basename in added_files:
        logger.info(f"File {file_basename} was previously added. Skipping.")
        continue

    langchain_docs = []
    if use_adobe_api:
        logger.warning(f"Adobe API extraction failed. Falling back to Unstructured.io.")
    else:
        logger.info(f"Using Unstructured.io to partition {file_basename}...")

    if not langchain_docs: # If Adobe API was not used, or failed, use Unstructured
        try:
            # --- ADDED LOGGING AROUND partition_pdf ---
            logger.info(f"Attempting to partition {file_basename} with Unstructured.io (strategy='hi_res', model='yolox'). This may take time...")
            elements = partition_pdf(
                filename=file_path,
                strategy="hi_res",
                infer_table_structure=True,
                hi_res_model_name="yolox",
                chunking_strategy="basic",
                # Consider adding infer_ocr_text=False here if your PDFs are text-based and not scanned.
                # This can drastically reduce memory and time.
                # infer_ocr_text=False,
            )
            logger.info(f"SUCCESS: Finished partitioning {file_basename}. Found {len(elements)} unstructured elements.")

            logger.info("Converting unstructured elements to Langchain Documents...")
            langchain_docs = convert_unstructured_elements_to_langchain_documents(elements)
            logger.info(f"SUCCESS: Converted to {len(langchain_docs)} Langchain Documents.")
        except Exception as e:
            logger.error(f"Error partitioning or converting {file_basename} with Unstructured: {e}", exc_info=True)
            logger.error(f"Skipping {file_basename} due to processing error.")
            continue

```

```

# Initialize text splitter
logger.info("Initializing RecursiveCharacterTextSplitter for chunking...")
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=2000,
    chunk_overlap=200,
    length_function=len,
    is_separator_regex=False,
)

# Split documents into chunks
logger.info("Splitting documents into chunks...")
chunks = text_splitter.split_documents(langchain_docs)
logger.info(f"Original documents split into {len(chunks)} chunks.")

# Filter complex metadata to avoid Chroma issues
logger.info("Filtering complex metadata...")
filtered_chunks = filter_complex_metadata(chunks)
logger.info(f"Filtered chunks to {len(filtered_chunks)} valid documents.")

try:
    existing_ids_in_db = set(vector_store.get(include=[])[ 'ids' ])
    logger.info(f"Number of existing documents in DB before adding current file's chunks: {len(existing_ids_in_db)}")
except Exception as e:
    existing_ids_in_db = set()
    logger.warning(f"Could not retrieve existing document IDs (likely collection not yet created or error): {e}")
    logger.info("Proceeding assuming collection might be empty or new.")

```

```

chunks_to_add = []
ids_to_add = []

for j, chunk in enumerate(filtered_chunks):
    # Prepend filename to the chunk content
    chunk.page_content = f"[Source: {file_basename}]\n\n{chunk.page_content}"

    # Generate ID and add to chunk metadata
    chunk_with_id = generate_document_id(chunk, j)
    chunk_id = chunk_with_id.metadata["id"]

    if chunk_id not in existing_ids_in_db:
        chunks_to_add.append(chunk_with_id)
        ids_to_add.append(chunk_id)
    else:
        logger.info(f"Chunk with ID {chunk_id} already exists in DB. Skipping.")

if chunks_to_add:
    logger.info(f"👉 Adding {len(chunks_to_add)} new chunks from {file_basename} to ChromaDB.")
    # --- ADDED LOGGING AROUND ChromaDB ADD ---
    logger.info(f"Attempting to add {len(chunks_to_add)} documents to ChromaDB for {file_basename}...")
    add_documents_in_batches(vector_store, chunks_to_add, ids_to_add)
    logger.info(f"SUCCESS: Added {len(chunks_to_add)} documents to ChromaDB for {file_basename}.")
    # --- END ADDED LOGGING ---
    newly_added_files.add(file_basename)
else:
    logger.info(f"No new chunks to add for {file_basename} after deduplication.")

# Update Added_files.txt with all files successfully processed in this run
if newly_added_files:
    logger.info(f"\nUpdating Added_files.txt with {len(newly_added_files)} newly processed files...")
    with open(added_files_path, "a") as f:
        for filename in newly_added_files:
            f.write(f"{filename}\n")
    logger.info("Added_files.txt updated.")
else:
    logger.info("No new files were added during this ingestion run.")

logger.info("\nIngestion process complete for all specified files.")

```

1. Initial Setup: It first locates PDF documents within a specified data directory and initializes the OllamaEmbeddings model (for converting text to numerical vectors) and the ChromaDB instance (the vector database itself).
2. Deduplication & Incremental Ingestion: To ensure efficiency and prevent duplicate entries, it reads a list of previously processed files from a log file (Added\_files.txt) and also checks for existing document chunks directly within ChromaDB. Only new or unprocessed PDF files and their unique chunks are considered for addition.

3. Document Processing: For each new PDF file, it uses Unstructured.io to robustly parse the document, extracting text and identifying structural elements like tables. This extracted content is then converted into Langchain Document objects.
4. Chunking & Preparation: These larger documents are subsequently broken down into smaller, semantically relevant "chunks" using a RecursiveCharacterTextSplitter. Each chunk is enriched with its original filename (as a source reference) and assigned a unique identifier.
5. Vectorization & Storage: Finally, the prepared chunks are vectorized (converted into numerical embeddings) using the OllamaEmbeddings model, and these embeddings, along with their metadata, are added to the ChromaDB, making them searchable for the RAG system.
6. Logging: The function meticulously logs its progress, including which files are processed, skipped, and successfully added, and updates the Added\_files.txt log to track successful ingestions.

### **5.3 Sentiment Analysis Core Code Documentation**

This code snippet automates the entire process from data acquisition to quarter-over-quarter sentiment analysis of earnings call transcripts at both company and sector levels, and historical data management, providing a dynamic overview of sentiment across financial sectors.

```

def get_sentiment(): #THIS FETCHES THE LATEST DATA
    COLLECTION_NAME = "Financial_Reports" # Consistent collection name
    CSV_FILE = "data/sector_sentiment_analysis_summary.csv"

    # Check if CSV file exists
    if not os.path.exists(CSV_FILE):
        # Default periods if CSV doesn't exist
        first_transcript_year = 2024
        first_transcript_quarter = 3
        second_transcript_year = 2024
        second_transcript_quarter = 4
        print(f"\nCSV file '{CSV_FILE}' not found. Using default periods:")
        print(f"First transcript: Q{first_transcript_quarter} {first_transcript_year}")
        print(f"Second transcript: Q{second_transcript_quarter} {second_transcript_year}")
    else:
        # Read existing CSV and determine the next period
        try:
            df = pd.read_csv(CSV_FILE)

            # Method 1: If periods are stored in column names or headers
            # Look for patterns like "Q4_2024-Q1_2025" in column names
            period_pattern = r'Q(\d)_(\d{4})-Q(\d)_(\d{4})'
            periods_found = []

            # Check column names for period information
            for col in df.columns:
                if col != 'Sector': # Skip the Sector column
                    match = re.search(period_pattern, str(col))
                    if match:
                        # Extract both periods from the column name
                        first_quarter, first_year = int(match.group(1)), int(match.group(2))
                        second_quarter, second_year = int(match.group(3)), int(match.group(4))
                        periods_found.append(((first_year, first_quarter), (second_year, second_quarter)))

            if periods_found:
                # Find the latest period pair (based on the second period in each pair)
                latest_period_pair = max(periods_found, key=lambda x: (x[1][0], x[1][1]))
                latest_first_period, latest_second_period = latest_period_pair

                # The next analysis should start from the second period of the latest analysis
                first_transcript_year = latest_second_period[0]
                first_transcript_quarter = latest_second_period[1]

```

```

            # Calculate the next quarter for the second transcript
            if first_transcript_quarter == 4:
                second_transcript_year = first_transcript_year + 1
                second_transcript_quarter = 1
            else:
                second_transcript_year = first_transcript_year
                second_transcript_quarter = first_transcript_quarter + 1

            print(f"\nFound existing analysis. Automatically selecting next period:")
            print(f"First transcript: Q{first_transcript_quarter} {first_transcript_year}")
            print(f"Second transcript: Q{second_transcript_quarter} {second_transcript_year}")

        else:
            # Fallback to default if no periods found in CSV
            print(f"\nCSV file exists but no period information found. Using default periods:")
            first_transcript_year = 2024
            first_transcript_quarter = 3
            second_transcript_year = 2024
            second_transcript_quarter = 4
            print(f"First transcript: Q{first_transcript_quarter} {first_transcript_year}")
            print(f"Second transcript: Q{second_transcript_quarter} {second_transcript_year}")

```



```

csv_for_tickers = 'grouped_by_gics_sector.csv'
num_companies_per_sector = 2 # You can adjust this number. For now, we'll aim for this many.

print("Fetching fresh tickers and sectors...")
stockanalysis_df = get_tickers_and_sectors_from_stockanalysis() # Assuming this function is defined elsewhere

if stockanalysis_df.empty:
    print("No data retrieved from StockAnalysis.com API. Exiting.")
    return

print(f"Retrieved {len(stockanalysis_df)} tickers from StockAnalysis.com")

yfinance_market_cap_data = []

# Define retry parameters for market cap fetching
max_market_cap_retries = 3
initial_retry_delay_seconds = 5 # Initial delay between retries

print("Fetching Market Cap data using yfinance (with retries for failures)...")

for i, row in stockanalysis_df.iterrows():
    ticker = row['Symbol']
    current_market_cap = np.nan # Initialize to NaN, will be updated on success

    # --- Immediate Retry Logic for Market Cap Fetching ---
    for attempt in range(max_market_cap_retries):
        print(f"[{i+1}/{len(stockanalysis_df)}] Fetching Market Cap for {ticker} (Attempt {attempt + 1}/{max_market_cap_retries})...")

        fetched_cap = get_company_market_cap(ticker) # This returns float or np.nan

        if pd.notna(fetched_cap): # pd.notna() checks if a value is NOT NaN (includes numbers, None, etc.)
            current_market_cap = fetched_cap
            print(f"Successfully fetched market cap for {ticker} on attempt {attempt + 1}.")
            break # Market cap obtained, exit retry loop
        else:
            # If fetched_cap is np.nan, it means get_company_market_cap failed
            print(f"Failed to fetch market cap for {ticker} on attempt {attempt + 1}. Error details should be above.")
            if attempt < max_market_cap_retries - 1:
                # Exponential backoff for subsequent retries
                delay = initial_retry_delay_seconds * (2 ** attempt)
                print(f"Retrying {ticker} after {delay} seconds...")
                time.sleep(delay)

```

```

        else:
            print(f"Max retries ({max_market_cap_retries}) exhausted for {ticker}. Market cap will remain NaN for this run.")
    # --- End Immediate Retry Logic ---

    yfinance_market_cap_data.append({'Symbol': ticker, 'MarketCap': current_market_cap})

    # Small general delay between tickers to be courteous to the API,
    # separate from retry delays which are for specific failures.
    time.sleep(0.1)

# Step 3: Create DataFrame and merge
df_market_cap = pd.DataFrame(yfinance_market_cap_data)

df_combined = pd.merge(
    stockanalysis_df,
    df_market_cap,
    on='Symbol',
    how='left'
)

# Ensure MarketCap column is numeric, coercing any non-numeric (like if an error slipped through) to NaN
df_combined['MarketCap'] = pd.to_numeric(df_combined['MarketCap'], errors='coerce')

# --- NEW LOGIC: Filter out rows where MarketCap is NaN ---
initial_row_count = len(df_combined)
df_combined = df_combined.dropna(subset=['MarketCap'])
rows_removed_count = initial_row_count - len(df_combined)

if rows_removed_count > 0:
    print(f"Removed {rows_removed_count} tickers from 'combined_stock_data.csv' because their market cap could not be fetched after retries.")
else:
    print("All tickers retained in 'combined_stock_data.csv' as market caps were successfully fetched for all.")
# --- END NEW LOGIC ---

# Step 4: Save combined data
combined_output_file = 'combined_stock_data.csv'
df_combined.to_csv(combined_output_file, index=False)
print(f"Combined data saved to '{combined_output_file}'")

```



```

# Get a list of all unique sectors from the initial DataFrame, to ensure all sectors are covered
all_gics_sectors = companies_df_for_analysis['GICS_Sector'].unique()

for sector_name in sorted(all_gics_sectors): # Iterate through all possible sectors
    companies_data = sector_weighted_data.get(sector_name, []) # Get data for this sector, default to empty list if none

    weighted_avg_sentiment_change = None
    num_tickers_included = len(companies_data) # Count of companies included with valid data (that had valid % change)

    if companies_data: # Only proceed if there's actual data to weight
        total_market_cap_in_sector = sum(item[1] for item in companies_data) # sum of market_caps for companies *with valid sentiment change*

        if total_market_cap_in_sector > 0:
            weighted_sum_of_changes = sum(item[0] * (item[1] / total_market_cap_in_sector) for item in companies_data)
            weighted_avg_sentiment_change = round(weighted_sum_of_changes, 1) # Round to 1 decimal place

    # Format the combined value string exactly as requested: "(value, count)"
    if weighted_avg_sentiment_change is not None:
        combined_value_string = f"({weighted_avg_sentiment_change}, {num_tickers_included})"
    else:
        combined_value_string = f"(N/A, {num_tickers_included})" # Use N/A if sentiment change couldn't be calculated

    # Add to the list with ONLY the two desired columns for the current quarter's data
    csv_records_for_output.append({
        'Sector': sector_name,
        'dynamic_column_header': combined_value_string
    })

csv_file_name = 'data/sector_sentiment_analysis_summary.csv'
new_quarter_df = pd.DataFrame(csv_records_for_output)

if os.path.exists(csv_file_name):
    print(f"\nExisting CSV '{csv_file_name}' found. Merging new quarter data as a column.")
    existing_df = pd.read_csv(csv_file_name)

    # Merge the existing DataFrame with the new quarter's data
    # 'outer' merge ensures all sectors from both are kept.
    # This will add the new column or update if it already exists (if dynamic_column_header matches an existing one)
    combined_df = pd.merge(existing_df, new_quarter_df, on='Sector', how='outer')

```

```

# Fill any NaNs that might have appeared for existing columns (for new sectors) with "(N/A, 0)"
for col in combined_df.columns:
    if col != 'Sector' and combined_df[col].isnull().any():
        combined_df[col].fillna("(N/A, 0)", inplace=True)

# Reorder columns: 'Sector' first, then existing columns, then the new dynamic column.
ordered_columns = ['Sector']
# Add existing columns from previous file, excluding the current dynamic column
for col in existing_df.columns:
    if col != 'Sector' and col != dynamic_column_header:
        ordered_columns.append(col)
# Add the current dynamic column (it will be added only once if it exists or is new)
if dynamic_column_header in combined_df.columns and dynamic_column_header not in ordered_columns:
    ordered_columns.append(dynamic_column_header)

# Handle any other new columns that might have been introduced during merge but not explicitly ordered (unlikely but safe)
# This loop ensures that any columns added by 'outer' merge not in ordered_columns (very rare for this use case) are included.
for col in combined_df.columns:
    if col not in ordered_columns:
        ordered_columns.append(col)

# Ensure all columns in ordered_columns exist in combined_df before reindexing
ordered_columns = [col for col in ordered_columns if col in combined_df.columns]
combined_df = combined_df[ordered_columns]

print(f"✅ Appended new column '{dynamic_column_header}' to '{csv_file_name}'")

else:
    print(f"\nNo existing CSV '{csv_file_name}' found. Creating new file with data for '{dynamic_column_header}'")
    combined_df = new_quarter_df

# Write the combined DataFrame back to the CSV (mode='w' to overwrite the file with the new structure)
if not combined_df.empty:
    combined_df.to_csv(csv_file_name, mode='w', index=False, header=True)
    print(f"✅ CSV '{csv_file_name}' updated successfully.")
else:
    print("\nNo data to write to CSV after combining.")

```

1. **Determine Analysis Periods:** It intelligently identifies the two chronological quarters to analyze. If a summary CSV file (sector\_sentiment\_analysis\_summary.csv) exists, it

determines the next analysis period based on previously analyzed data; otherwise, it defaults to fixed recent quarters.

2. **Gather Company Data:** It fetches a list of public company tickers and their GICS sectors, then retrieves their market capitalization using `yfinance`, applying retry logic for robustness. Companies with unretrievable market cap data are excluded.
3. **Prepare Companies for Analysis:** The fetched company data is saved, grouped by sector, and then filtered to select a specified number of top companies per sector for detailed sentiment analysis.
4. **Initialize AI Models:** It sets up the necessary AI components: an `OllamaEmbeddings` model (for text vectorization), a `RecursiveCharacterTextSplitter` (for breaking documents into smaller chunks), and a `ChatOllama LLM` (for generating sentiment analysis).
5. **Process Each Company:** For every selected company:
  6. It checks if earnings call transcripts for the two analysis quarters are already stored in a dedicated ChromaDB vector store.
  7. If not found, it attempts to fetch the transcripts from an external API and then ingests them into the ChromaDB.
  8. The LLM is then prompted with both quarters' transcripts to analyze sentiment for each period and calculate the percentage change in sentiment score between them.
  9. Extracted sentiment scores and the calculated percentage change are stored.
10. **Aggregate Sector Sentiment:** After processing all individual companies, the function calculates a market-cap-weighted average of the sentiment changes for each GICS sector, providing an aggregate view of sentiment shifts within industries.
11. **Update Historical Summary:** Finally, it appends this newly calculated sector-level sentiment data (including the count of companies contributing to the average) as a new column to the `sector_sentiment_analysis_summary.csv` file, incrementally building a historical record of sector sentiment trends.

## 5.4 Unique Features of DeepGreen's Codebase

1. **Dual-Layer Document Deduplication for Content Consistency:** DeepGreen employs a sophisticated two-stage approach to prevent redundant document ingestion, ensuring the vector database remains clean and efficient:

- **Filename-Based Skip:** During the ingestion process (ingest\_documents\_to\_chroma), the system maintains a log of Added\_files.txt. Before processing any PDF, it checks if the file's base name already exists in this log. If so, the file is immediately skipped, efficiently avoiding re-processing already ingested documents.
  - **Chunk-Content ID Check:** Beyond just filename, the system generates a unique ID for each individual document chunk (e.g., using a hash of its content or a combination of filename and chunk index, via generate\_document\_id). Before adding a chunk to ChromaDB, it queries the database to see if a chunk with that exact ID already exists (chunk\_id not in existing\_ids\_in\_db). This is a crucial, granular check that prevents duplicate chunks from being stored even if, for instance, a file was re-ingested with a slightly different name but contained identical content, or if a previous ingestion run was interrupted. This ensures true content-based consistency in the vector store.
2. **High-Resolution PDF Parsing with Intelligent Table Extraction:** The ingestion pipeline prioritizes rich, accurate data extraction from complex financial PDFs:
- **Advanced Parsing Strategy:** DeepGreen utilizes Unstructured.io with a strategy="hi\_res" and hi\_res\_model\_name="yolox". This configuration leverages advanced computer vision models to accurately identify and extract various elements (text, tables, figures) from visually complex documents, including scanned PDFs.
  - **Direct Table Inference to Markdown:** Crucially, infer\_table\_structure=True ensures that Unstructured.io attempts to understand the tabular layout within PDFs. When tables are detected, the code converts them directly into Markdown table format within the extracted text content. This is a significant advantage, as Markdown tables are highly structured and readily interpretable by Large Language Models, enabling the LLM to accurately read and reason about financial data presented in tables. The LLM's system prompt is also explicitly designed to handle Markdown table interpretation, ensuring seamless integration.
3. **Continuous Chat Functionality with Session-Based Context:** DeepGreen provides a fluid, multi-turn conversational experience:

- **Session Cookie Management:** The FastAPI `/query/` endpoint (`query_rag` function) utilizes HTTP session cookies to uniquely identify each user's conversation. If a user doesn't have a `session_id` cookie, a new one is generated and set, creating a persistent (within browser session) context for that user.
  - **Persistent Conversation History:** A dedicated `ChatContext` object is associated with each `session_id`. This object stores the entire history of queries and responses for that specific user. When a new query comes in, the `get_context()` method of `ChatContext` retrieves the past conversation.
  - **Contextual LLM Prompting:** This `previous_conversation` history is dynamically injected into the LLM's system prompt (via the `{previous_conversation}` placeholder). This allows the LLM to understand follow-up questions, remember previous facts, and provide more coherent and contextually relevant answers over multiple turns, simulating a natural dialogue.
4. **Traceable LLM Answers with Source Attribution:** Ensuring transparency and trust in financial analysis is paramount:
- **Automatic Source Extraction:** When the RAG system retrieves relevant document chunks (`query_rag`), the `extract_sources_from_raw_chunks` utility parses these chunks to identify and collect the unique `[Source: filename]` tags.
  - **Inclusion in Response:** The extracted source names are then included in the API response that DeepGreen returns to the user. This feature allows users to immediately see which specific financial reports or documents contributed to the LLM's answer, enhancing the trustworthiness and verifiability of the information.
5. **Market Capitalization-Weighted Sector Sentiment Analysis:** DeepGreen's sentiment analysis provides a more realistic and impactful view of market dynamics:
- **Beyond Simple Averages:** Instead of simply averaging the sentiment changes of all companies within a sector, the `get_sentiment()` function calculates a market-cap-weighted average. This means that the sentiment change of larger companies (those with higher market capitalization) will have a proportionally greater influence on the overall sector sentiment score.

- Reflecting Market Impact: This weighting scheme is critical in finance as larger companies typically exert a greater influence on a sector's overall health and investor sentiment. By using market cap, DeepGreen's sector sentiment reflects actual market dynamics more accurately, providing more actionable insights.
6. Intelligent Period Tracking for Automated Sentiment Analysis: The `get_sentiment()` function is designed for seamless, continuous analysis:
- Automated Next Period Detection: It reads the existing `sector_sentiment_analysis_summary.csv` file and intelligently parses its column headers (e.g., "Q4\_2024-Q1\_2025") to identify the latest completed analysis period. It then automatically determines the *next chronological quarter-over-quarter period* to analyse, eliminating the need for manual date configuration.
  - Transcript Availability Check: Before performing sentiment analysis for a given company and its two target quarters, the system explicitly checks if valid parsed transcripts for *both* quarters are available (either from the vector database or fetched from an API). If either transcript is missing, the company is gracefully skipped for that analysis run. This "smart tracking" prevents the system from attempting analysis on periods for which data is not yet released or successfully retrieved, ensuring the robustness and accuracy of the analysis pipeline.

## **6. Business Rules**

### **Regulatory Compliance:**

- All data collection, processing, and storage practices must comply with applicable laws and standards.

### **Clear Communication of Limitations:**

- The application shall include clear notices stating that the service does not provide professional medical advice and that users should consult qualified health professionals for diagnosis and treatment.

### **Operational Guidelines:**

- External API integrations must adhere to the service-level agreements (SLAs) provided by respective third-party services.

### **User Input Validation:**

- The system must enforce strict validation rules on user inputs (e.g., format of email addresses, password complexity) to prevent erroneous or malicious data entry.

### **Audit and Monitoring:**

The system will maintain logs of user interactions and system events for auditing and troubleshooting purposes, and these logs must be secured and accessible only to authorised personnel.