IR Project Final Report

*

Nguyen Huu Hao 21125160
Phan Trung Duc 21125035
Quan Phu Long 21125080
Nguyen Dinh Triet 21125097
Nguyen Si Minh 21125081

I. Introduction

A. Background

Retrieval-Augmented Generation (RAG) is a technique that merges information retrieval with generative models to enhance natural language processing tasks. In RAG, documents are first chunked into smaller units, such as sentences or paragraphs, to make the retrieval process more efficient and focused on relevant content. These chunks are then vectorized using pre-trained language models, which convert the text into dense numerical representations (embeddings) that capture semantic meaning. The vectorized chunks are indexed in a database, allowing for fast retrieval based on similarity to the query. This process enhances the system's ability to find contextually relevant information quickly and accurately, improving the quality of the generated responses.
By incorporating real-time, external knowledge, RAG addresses the limitations of traditional generative models, improving accuracy and reducing hallucinations in the generated content. This approach is especially useful in applications such as question-answering, dialogue systems, and summarization, where access to up-to-date and relevant information is essential.

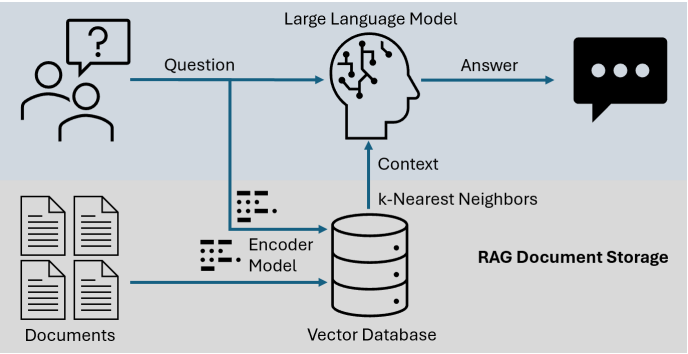

Fig. 1. Illustration of the naive RAG system

B. Objectives of the Retrieval System

The main objective of this retrieval system is to help users stay informed by providing access to the latest news and updates from across the internet. It is designed to quickly retrieve current information, allowing users to remain aware of important developments in various fields.
In addition to keeping users updated on news, the system also functions as an intelligent assistant to aid with document management. It helps users by efficiently processing and analyzing various types of documents, including academic papers, legal contracts, and other important texts. The system enables users to easily read, understand, and extract relevant information from these documents, improving productivity and streamlining tasks that involve extensive reading or research.

II. System Architecture

A. Architecture

Raw documents are indexed and stored on disk with IDs, metadata, text, and images where text is chunked. Only vectors and their ID are loaded into RAM within their respective space.

B. Basic Retrieval Pipeline

Users perform vector searches, and the system retrieves relevant vector IDs, which can be further explored for detailed information. A preprompted Large Language Model (LLM) then processes these documents to generate and return a summarized answer with supporting evidence, ensuring efficient and accurate information retrieval.

C. Why This Architecture

1) **Semantic Searching**
Traditional keyword-based retrieval methods like BM25 rely on exact matches between query terms and document terms. However, users may misspell words or use different terms that convey similar meanings. This architecture addresses that issue by enabling semantic search, which allows the system to retrieve relevant documents even if the query differs from the exact terminology used in the documents. This flexibility makes the retrieval process more user-friendly and accurate.

2) **Utilizing Semantic Metadata**
In many document retrieval systems, metadata such as title, date, and author are typically used for basic filtering rather than fully leveraged in the retrieval process. This architecture goes a step further by considering the semantic meaning of metadata during retrieval. For example,

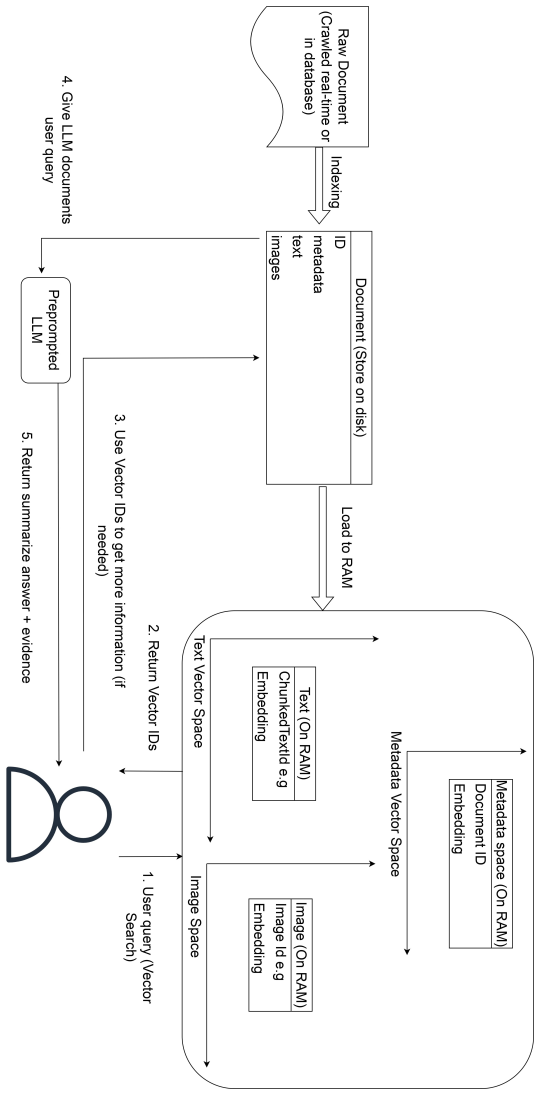

Fig. 2. System architecture

instead of just filtering documents by date or author, it can use these elements to better understand context and relevance, improving overall search results.

3) **Chunking Documents for Efficient LLM Input**
In this architecture, documents are chunked before being passed to the Large Language Model (LLM). Instead of sending the entire document, only relevant portions are fed into the model. This approach saves on token usage, allowing for more efficient processing. It also ensures that the LLM focuses only on the most pertinent information, avoiding unnecessary computation and maximizing the quality of results.

III. Data Collection

A. Crawling Data

1) *Medium Data Crawling*

- **API Usage:** Medium articles are crawled using the RapidAPI service, which allows searching for articles based on specific keywords. The API provides access to article metadata (title, author, publish date, etc.) and the full content of the articles.
- **Article Fetching:** For each article, the following steps are performed:
  - Search articles based on a keyword.
  - Fetch detailed article metadata and content (including text and images).
  - Embedding of article content and images for further processing.
  - Data is then stored in a structured format (JSON files), including the content and metadata.

2) *CNN Data Crawling*

- **Web Scraping:** CNN articles are scraped directly from the website. Using tools like BeautifulSoup or other scraping techniques, article links, titles, metadata, and content are extracted from the CNN pages.
- **Content Extraction:** After retrieving the article's HTML, the content is parsed and cleaned to remove unnecessary elements like ads, scripts, etc. The focus is on extracting the main body of the article and any associated images.
- **Data Storage:** Similar to Medium, the content from CNN is structured and stored in JSON format, including article text and images for further embedding or analysis.

B. Preprocess Data

- **Text Preprocessing:** For both CNN and Medium, the content is preprocessed by removing unwanted characters and splitting the text into paragraphs or sentences. This helps in organizing the content for embedding and later retrieval.
- **Image Preprocessing:** Images scraped from CNN and Medium are processed by generating embeddings through models like NomicEmbedVision (for image content). These embeddings are stored as part of the article data to allow for semantic search or further analysis.
- **Text Embedding:** The text of the articles is embedded using text embedding models. This transforms the textual content into vectors that capture the semantic meaning of the text. This embedding is stored alongside the article data and enables efficient indexing and retrieval.

C. Indexing

Indexing organizes preprocessed data by keywords, tags, or metadata to facilitate efficient storage and searchability. In this

process, chunking is employed to break lengthy documents into smaller, more manageable segments. This segmentation enables finer-grained indexing, yielding more accurate search results and better handling of large datasets. Specifically, we leverage **semantic chunking**, a strategy that optimizes the granularity of the indexed content while maintaining high retrieval performance.

1) Semantic Chunking Approach

Unlike traditional sentence-based chunking, we utilize **paragraph-based semantic chunking**. This approach is based on the fact that paragraphs in most documents are already thematically coherent units, and the sentences within a paragraph are generally related in meaning. As a result, **semantic relationships between adjacent sentences** within a paragraph are typically stronger compared to those between sentences spread across different paragraphs. For more details on the levels of chunking, refer to the article by Anurag Mishra on Medium.

This allows for the following advantages:

- **Reduced Chunking Complexity:** Since paragraphs naturally group related information, the need for additional chunking based on sentence boundaries is minimized. We only need to determine if two adjacent paragraphs can be logically merged, which simplifies the chunking process.
- **Improved Indexing Efficiency:** By working at the paragraph level, we avoid unnecessary splits and can directly focus on merging or splitting paragraphs that hold distinct meanings. This saves time during the indexing phase and ensures that the chunking operation is efficient.
- **Better Embedding Quality:** Paragraph-based chunks typically contain more context and meaning than sentence-based ones, leading to richer and more informative embeddings. Each paragraph-level embedding is likely to be more semantically coherent and thus more useful for subsequent retrieval tasks.
- **Reduced Embedding Redundancy:** With fewer, more meaningful chunks, we decrease the overall number of embeddings, which reduces the computational load and storage requirements while preserving high-quality representations of the data.

By indexing and embedding at the paragraph level, we ensure that the retrieval process can more effectively match queries with content that is contextually relevant, without the need for excessive segmentation of text.

2) Integration with Text and Image Embeddings

Both **text embeddings** (derived from the paragraph chunks) and **image embeddings** are stored in an index. This combined indexing of text and images enables rapid, multi-modal retrieval. Queries can be executed based on keywords, metadata (e.g., authorship, publication dates), or the content itself (e.g., searching by topic or visual characteristics). This indexing strategy ensures a high level of precision and scalability, making it easier to query large datasets with diverse content types.

IV. Indexing and Storage

A. Indexing Techniques

Indexing is the process of converting raw data, such as text and images, into embeddings, which are vector representations that capture the semantic meaning of the data. Vectors that are closer to each other in the vector space are more semantically similar. In the retrieval phase, the system identifies the top k vectors most similar to the query vector.

For embedding models, we use nomic-embed-text-v1.5 for text and nomic-embed-vision-v1.5 for images, both of which support multi-modal retrieval. Notably, nomic-embed-text-v1.5 is considered to be competitive with OpenAI's text-embedding-small-3 in terms of text embedding quality.

B. Embedding Quantization for Memory Efficiency

One of the biggest challenges in any Retrieval-Augmented Generation (RAG) system is the size of the vectors. Typically, `nomic-embed-text-v1.5` generates a vector with 768 elements of type `float32`, which is approximately 3KB per vector.

For scalability, as the system's data grows, we employ a quantization technique — specifically, binary quantization for text embeddings. In binary quantization, each element in the vector is mapped to a single bit, where:

$$x = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Note that bit-quantization is currently supported only on Qdrant Cloud. However, locally in RAM, we simulate binary quantization by converting the vector to `int8` (with values of 1 and 0 and decreasing the size to 4x). This approach has been shown to work effectively.

It's important to note that we only apply binary quantization to text embeddings and not to image embeddings, due to the distinct distribution patterns of image vectors in the vector space.

C. Storing Vectors to on disk

To optimize storage efficiency, we store indexed data in JSON files by converting raw floating-point vectors into a more compact format. Unlike Llama-Index, which stores vectors as raw floating-point values, we encode the vectors as strings of bits and then convert these bit strings into Base64 format. This significantly reduces the storage size. For instance, a JSON file from approximately 100KB to about 4KB. Furthermore, this method retains the readability of the data, allowing users to inspect their database while achieving substantial space savings.

V. Retrieval and Multi-Step Pipeline

A. Query Preprocessing

Query preprocessing refines user queries before they are embedded, ensuring clarity and accuracy. The query is first passed through a large language model (LLM) to correct syntax, grammar, and expression. Key steps include:

- **Grammar and Syntax Correction:** The LLM fixes errors, ensuring a well-formed query.
- **Semantic Refinement:** The LLM rephrases or clarifies the query to better express user intent.
- **Keyword Emphasis:** Important terms are identified and highlighted to improve focus.

Once refined, the query is converted into a vector representation, ready for embedding and efficient retrieval.

B. Retrieve text and images

To retrieve text, the user's query is embedded and binary quantized, just like the text embeddings. The distance metric used is Hamming distance, but since the library only supports Manhattan distance, we use it in this case, as it is equivalent to Hamming distance for binary quantization.

To retrieve images, the user's query is embedded but not quantized. The distance metric used for images is the well-known cosine similarity.

C. Adaptive RAG Multi-Step Retrieval Workflow

The system first searches the local database for relevant documents. If the results from the local database do not fully satisfy the user's query, the system will then fetch additional documents from the internet and reprocess the query with the newly retrieved data.

Images are retrieved as a secondary task. While they may not be directly related to the retrieved text, they are closely aligned with the overall meaning of the user's query.

D. Searching and indexing internet search results

When the user enters a retrieval request that is not already saved in our database, it will perform a live internet search for that specific information. After successfully fetching content from the internet, data is indexed and stored in a quantized database for later retrieval.

However, while the process of searching/crawling has been optimized to maximize the usage of available resources with multi-threading executions (multiple crawlers running in parallel), it is still quite a time-consuming process. Furthermore, the indexing step right after is also incredibly slow, so the user will have to wait a while before the final result is responded.

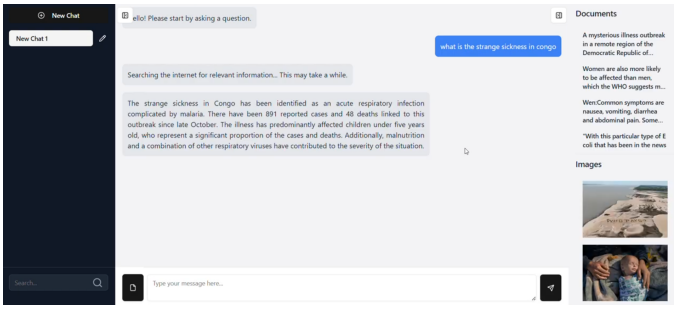

Fig. 3. Main Page

To avoid blocking the serving FastAPI application, we delegate the entire task of internet searching to a separate thread. The asynchronous flow is also carefully designed so that only one live search is conducted at any moment in first in, first out (FIFO) order, preventing resource starvation.

E. Handling Multi-Modal Data (Image-Text Retrieval)

The two embedding models used are multi-modal. However, quantizing the image embeddings results in poor performance due to the nature of the embedding vectors. As a result, the image embeddings are stored in `float32` format, while the text embeddings are quantized. Given that text data is more prevalent than image data, this decision is justified in terms of both retrieval accuracy and storage efficiency.

VI. User Interface (UI)

A. Overview

This section illustrates the User Interface (UI) and the interaction with the Retrieval-Augmented Generation (RAG) query system. The UI aims to provide users with an intuitive workflow for asking questions, viewing the answers, retrieving documents, and query preprocessing.

B. Technical Details

The application is served to the end-users using the Flask web framework following the REST convention. Users could interact with the application through a web page at localhost:4000, which is built using ReactJS. Note that the exposed API endpoints (`/api/*`) are designed to be headless, which allows for future UI extensions on other platforms like mobile or desktop.

C. Layout Description

The main page is divided into three primary components:

1) Left Sidebar

- **Chat History:** List past conversations by their user-provided titles and the user can delete the past conversation. This feature is implemented using the web browser's local storage API, so no user data is collected/stored by the application.
- **Query Preprocessor:** Generating context-aware sugges-

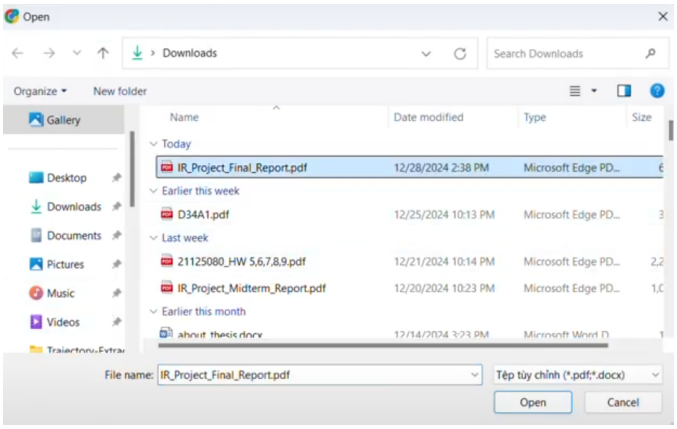

Fig. 4. Document Upload

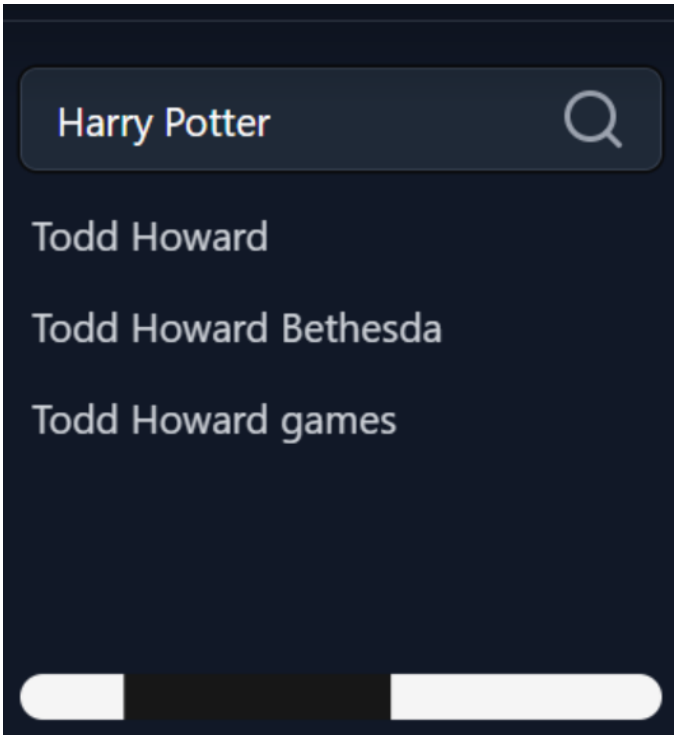

Fig. 5. Query preprocessing

tions based on the user's current input.

- **Document Upload:** Allow users to upload documents directly into the application for content processing or context generation.

2) Conversation Panel

- **User Query Input:** A text box at the bottom of the panel where the users type their questions. Clicking the submit button will send the query to the RAG system.

- **AI Generated Answer:** The system's response to the user's latest query.

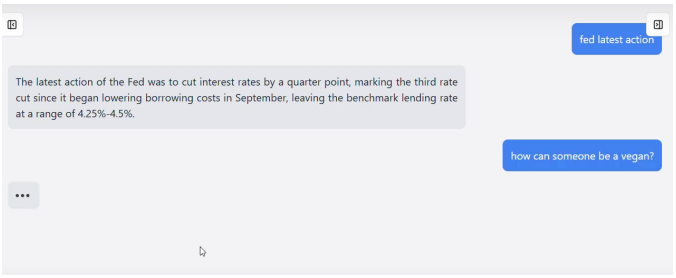

Fig. 6. Conversation Panel

3) Right Sidebar

The Right Sidebar displays a list of content retrieved from the user's search query, separated into two sections: documents and images.

- **Document:** The document section displays a list of documents retrieved from the user search query. Selecting any document in this list opens a dialog box that provides details, allowing users to review and analyze the content.

- **Image:** The image section displays all visual content retrieved from the search query.

VII. Summary

A. Future work

Future developments may include several key enhancements to improve the system's capabilities. First, we plan to implement a conversational Retrieval-Augmented Generation (RAG) model to facilitate more dynamic and context-aware interactions. Additionally, we aim to enable image-based retrieval, expanding the system's ability to handle multi-modal queries. Optimization of the database architecture will also be a focus, ensuring better performance and scalability. Lastly, we will explore and integrate higher-quality embedding models to further improve retrieval accuracy and overall system efficiency.

B. Conclusion

The proposed system presents an effective implementation of a Retrieval-Augmented Generation (RAG) model that keeps users up to date with the latest news and provides support for managing personal documents. It incorporates scalable features through embedding quantization, ensuring both efficient storage and retrieval. The system also offers a user-friendly interface, facilitating ease of use. Furthermore, it supports text-vision multi-modal retrieval, enhancing its versatility in handling both textual and visual data for more comprehensive query responses.

C. Contributions

21125160: Crawl data, Index data, Real-time search, Build Database
21125035: Semantic Chunking, Vision Embed, Document indexing, Database design, Query preprocess

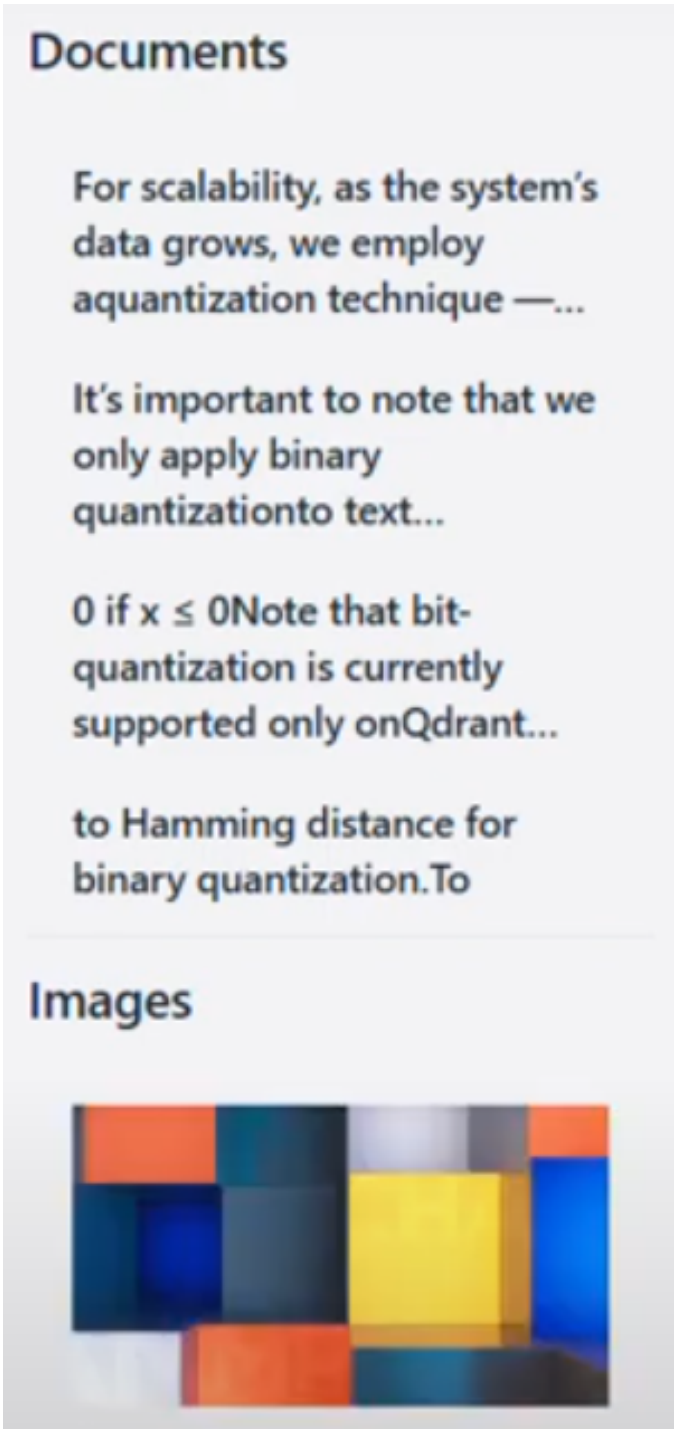

Fig. 7. Right Sidebar

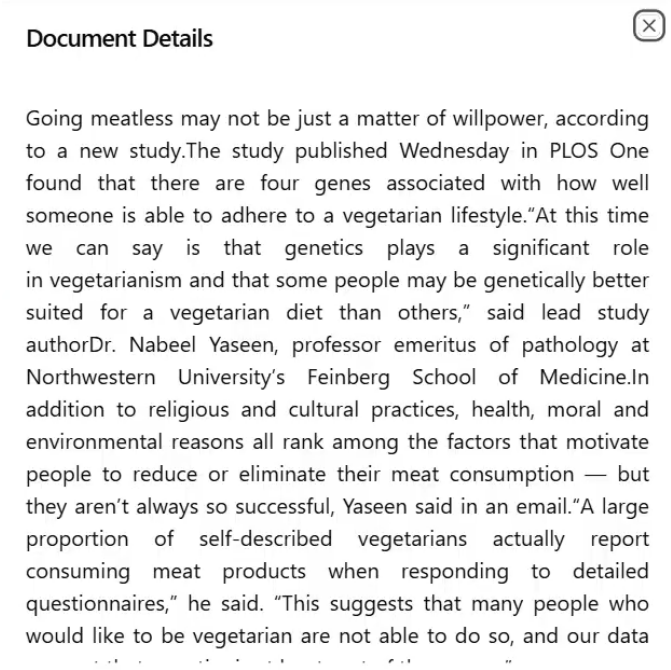

Fig. 8. Document Detail

21125080: Database design, Embedding Quantization, Build Database, Embed Text, Adaptive RAG
21125097: Front end, API, UI, Multi-threading optimization
21125081: Front end, API, UI

VIII. Submission

A. Source code and database

To execute the project, follow these steps:

1) **Access the Project Directory**: The project directory is available via the provided Google Drive link. All necessary libraries are pre-installed within this directory.
2) **Open VS Code**: Launch Visual Studio Code (VSCode).
3) **Configure the Virtual Environment**: In VSCode, edit the Python virtual environment to point to `resources/ir-project-venv`.
4) **Run the Application**: Open the terminal in VSCode and execute the following command:

```
py main.py
```

5) **Wait for the Backend Server**: After running the command, wait for the backend server to start. It will be accessible at localhost:4000.
6) **Database Location**: The project's database is located at `resources/quantized-db`.

By following these steps, the project will be set up and the backend server will be ready to use.

B. References

Links:

- Full project source code and database
- nomic-embed-text-v1.5
- nomic-embed-vision-v1.5
- Full demo video
- Github repo