**Documentation**

**Data Acquisition and Ingestion:**

1. Data Acquisition: (sharepoint\_file\_acquisition.py)

The SharePoint data acquisition process begins with the initialization and configuration of the environment, where necessary credentials and paths are loaded from a .env file, and logging is set up for tracking. Next, the process involves acquiring an OAuth2 access token using the Microsoft Authentication Library (MSAL) to authenticate against Azure AD, allowing access to the Microsoft Graph API. Once authenticated, the script retrieves the site ID and drive ID from the specified SharePoint site, enabling access to the document library.

The next step involves fetching user permissions, where the Microsoft Graph API retrieves the list of users and their associated permissions, including details like Teams membership and roles, which are then saved in a CSV file. The process then moves on to retrieve the document metadata list – “deliverables\_list” from the SharePoint site, where document’s data is saved for future reference.

With the necessary data in hand, the script traverses all folders and files in the specified drive, identifying new, updated, or deleted items since the last run by comparing timestamps. For new or updated files, the script downloads the content from SharePoint, processes the files for ingestion, and extracts content and metadata, which are then used to update the vector store. Deleted files are removed from the vector store to ensure consistency.

Finally, the updated file metadata is saved, the vector store is refreshed, and the completion timestamp is recorded to track the last successful run. The entire process is logged for monitoring, and any errors encountered are handled gracefully. This end-to-end flow ensures that the vector store and metadata remain current with the latest content from SharePoint, supporting efficient retrieval and usage in downstream applications.

1. File Identification and Ingestion Process: (ingest.py)

The script begins with renaming files with uppercase extensions to lowercase to ensure consistency and processes each file based on its format. It first checks if the file is a PDF and processes it accordingly, distinguishing between original PDFs and those converted from presentations. By original PDF meaning which is in PDF format like pages, and there are some PDFs which were originally PPT/PPTX and then converted to PDF.

Because the content type of PDF and PPT varies in nature, there is a need to process them like their original format. This is done by checking the page aspect ratio as the page ratio of PDF and PPT is different.

For files in DOC, DOCX, PPT, or PPTX formats, the script converts them to PDF, processes the converted file, and then removes the temporary PDF to keep the workspace clean. The conversion is managed by the convert\_document\_to\_pdf and convert\_presentation\_to\_pdf functions, which handle the conversion of DOC/DOCX and PPT/PPTX files, respectively, ensuring they are converted within a specified timeout period.

Post-conversion, the files are processed using distinct functions: one dedicated to handling PPT/PPTX files and converted PDFs, and another for DOC/DOCX files and original PDFs. This separation ensures that each file type is processed according to its specific requirements.

If any file fails during processing, it is logged and recorded in a failed\_files.csv for further review. This automated pipeline streamlines the handling of SharePoint documents by converting them to a standardized format, facilitating efficient processing and ingestion into the target system.

1. Ingestion in case of PPT, PPTX or PDF which were originally PPT/PPTX: (pdf\_ppt\_loader.py)

The process begins by extracting each slide from a presentation file (e.g., PPT) and saving them as individual PNG images in a dedicated folder. For instance, a PPT file with 10 slides will have each slide saved as "slide\_1.png," "slide\_2.png," and so on. Each image is then sent to a Language Model (LLM) for image summarization, and the returned text is stored in a dictionary where the key is the slide name and the value is the corresponding summary of that image.

This dictionary, which contains all the slide information, is stored in a vector store, and a MultiVector Retriever is created. The MultiVector store includes two main components:

* VectorStore: This stores the summary of each individual slide.
* DocStore: This stores both the summary and the base64 string representation of each image.

Both the VectorStore and DocStore also contain metadata extracted from the file in the initial step, which facilitates filtering the data based on user queries.

Additionally, another MultiVector Retriever is created to summarize the entire document. The individual slide summaries in the dictionary are combined to create a shorter, comprehensive summary of the whole presentation. This overall summary is stored in:

* Summary VectorStore: Contains the "summary of the file {name\_of\_the\_file}"
* Summary DocStore: Contains both the "summary of the file {name\_of\_the\_file}" and a one-pager summary generated by passing the entire document to the LLM.

This approach ensures that both detailed and holistic summaries of the presentation are available for retrieval, aiding in efficient and comprehensive content retrieval based on user queries.

1. Ingestion in case of PDF, DOC or DOCX: (pdf\_loader\_MV.py)

The process begins by extracting each slide from a presentation file (e.g., PPT) and saving them as individual PNG images in a dedicated folder. For instance, a PPT file with 10 slides will have each slide saved as "page\_1.png," "page\_2.png," and so on.

* 1. Text Extraction involves two methods:
     1. Method 1: The extracted image is passed through an OCR model (Florence-2) to extract the raw text. This raw text is then sent to a Large Language Model (LLM), where it is organized into a structured format. Once structured, the text undergoes another round of processing by the LLM to generate a concise summary.
     2. Method 2: The entire slide image is directly sent to the LLM (specifically GPT), which handles both OCR and the structuring of the text in one go. The structured text is then summarized by the LLM, creating a streamlined version of the content.
  2. Table Extraction is managed by sending each slide to the open-source model "TahaDouaji," which extracts tables as images. These table images are then processed by the LLM to generate summaries. The summarized table content is stored in a dictionary, where each key corresponds to a slide, and the value is the summary of the table image along with its base\_64 string representation.
  3. Image Processing follows a similar dual-method approach:
     1. Method 1: Extracted images from the slides are sent to the open-source model OpenBMB, which provides a detailed summary of the visual content.
     2. Method 2: The images are sent to GPT, which generates a summary of the image content. As with the other processes, these summaries are stored in a dictionary alongside their base\_64 strings.

A MultiVector Retriever is then created, which integrates and stores all the summarized content—text, tables, and images—along with their associated metadata in a vector store. The original document and its summarized content are preserved in a docstore, ensuring comprehensive data organization. This setup not only enables efficient retrieval of content but also allows filtering based on specific user queries.

Finally, an additional MultiVector Retriever is used to generate a comprehensive summary of the entire document. This involves combining the individual slide summaries from the dictionary into a shorter, cohesive summary that encapsulates the key points of the entire presentation. The overall summary is stored in two places:

1. Summary VectorStore: This contains the "summary of the file {name\_of\_the\_file}," a concise representation of the document's content.
2. Summary DocStore: This contains both the "summary of the file {name\_of\_the\_file}" and a one-pager summary generated by passing the entire document through the LLM. This one-pager offers an even more condensed view of the document, making it ideal for quick reference.

Together, these steps form a robust pipeline for processing and summarizing presentation files, enabling efficient storage, retrieval, and analysis of the content.

**Data Retrieval**

When a user submits a query, the system first checks whether it is a first-time interaction or if it includes previous conversations (i.e., chat history). If there is existing chat history, the query is rephrased to form a new standalone question that takes into account the prior conversation. This allows the system to interpret and respond appropriately, especially in cases where the user's query relies on context from earlier interactions (e.g., "Tell me more about it," where "it" refers to something discussed earlier).

Once the new question is formed, it is passed to an intent identification module. The system categorizes the query into one of four possible intents:

1. Direct Response: If the query can be answered directly, the system uses the content\_generator\_salutation function to provide a straightforward, polite response.
2. General Knowledge Query: For queries that require general external knowledge, the system utilizes a different generator (content\_generator\_GPT). This generator taps into a broad range of information, often using a large language model like GPT-3.5.
3. Regular Retrieval-Augmented Generation (RAG): If the query requires specific information retrieval (e.g., from a database or document repository), the system invokes the content\_generator.
4. Summary RAG: If the intent involves summarizing information, a specialized generator (content\_generator\_summary) is used to create a concise response.

Based on these 4 intents, the query is routed.

**Direct Response Workflow**:

When a query is routed for a direct response, particularly in cases where the user greets the AI assistant, the process ensures that the response is context-aware and polite.

1. The system starts by formatting the chat history. If there's a prior conversation, it formats it accordingly; otherwise, it notes "No Previous Conversation." This helps the AI understand the context of the interaction.
2. Model Selection: The system decides which AI model to use based on the value of llm:
   1. If llm is set to "GPT," the system uses AzureChatOpenAI, configured through environment variables (GPT-3.5)
   2. Otherwise, it defaults to using ChatOllama with a specific model, llama3\_1.
3. Prompt Crafting: A detailed prompt is created to instruct the AI on how to respond. The prompt includes:
   1. A description of the AI’s characteristics (helpful, knowledgeable, polite).
   2. Instructions to take into account the previous interactions.
   3. A directive to provide an appropriate salutation and a brief reference to the last topic discussed, ensuring the conversation flows smoothly.
4. Chain Creation: A chain is created to link various elements: formatted chat history, user question, prompt, model, and an output parser.
5. The AI generates a response asynchronously, providing text chunks in a streaming fashion. This allows for real-time response delivery to the client.
6. Conversation Update: Once the full response is generated, the conversation is updated with the new AI-generated text. The system returns a chat ID and message ID as JSON, allowing the conversation to be tracked and referenced later.

**GPT Response Workflow:**

When a query is routed for a GPT knowledge, particularly in cases where the user asks AI assistant for external knowledge, the process ensures that the response is context-aware and polite.

1. The system starts by formatting the chat history. If there's a prior conversation, it formats it accordingly; otherwise, it notes "No Previous Conversation." This helps the AI understand the context of the interaction.
2. Model Selection: The system uses AzureChatOpenAI, configured through environment variables (GPT-3.5) for general knowledge.
3. Prompt Crafting: A detailed prompt is created to instruct the AI on how to respond. The prompt includes:
   1. A description of the AI’s characteristics (helpful, knowledgeable, polite).
   2. Instructions to take into account the previous interactions.
   3. A directive to provide an appropriate salutation and a brief reference to the last topic discussed, ensuring the conversation flows smoothly.
4. Chain Creation: A chain is created to link various elements: formatted chat history, user question, prompt, model, and an output parser.
5. The AI generates a response asynchronously, providing text chunks in a streaming fashion. This allows for real-time response delivery to the client.
6. Conversation Update: Once the full response is generated, the conversation is updated with the new AI-generated text. The system returns a chat ID and message ID as JSON, allowing the conversation to be tracked and referenced later.

**Summary and Normal RAG**

Though the routes for Summary RAG and Regular RAG differ, the core processes they follow are quite similar. The key distinction lies in the retrievers they utilize.

1. Retriever Selection:
   1. When a user query is routed to these functions, the first step is to select the appropriate retriever. However, if the user has applied filters, such as selecting a specific country, region, or document, the system creates a new multi-vector retriever. This is done by passing the filters as search\_kwargs to the retriever, ensuring that the retrieval process is tailored to the user's preferences.
2. Context Building with Relevant Chunks:
   1. The system retrieves relevant chunks from the docstore. During retriever creation, these docstore chunks are mapped to vector store chunks. While similarity scoring is performed on the vector store chunks, the original chunks from the docstore are used for context building.
   2. A total of four chunks are retrieved. These chunks contain metadata, including a specific field called permissions related to the original document from which the chunk was derived. Only chunks where the user has permission to view the original document are considered for further processing.
3. Chunk Classification and Data Preparation:
   1. Once a chunk is considered, it is classified based on its content type:
      1. If the chunk is an image, it is added to a list called images.
      2. If the chunk is text, it is added to a list called texts.

Additionally, a summary of each chunk is stored in a list called summary

1. LLM Context Preparation:
   1. The collected data is then prepared for submission to the language model (LLM). The context window of the LLM is populated with various parameters:
      1. If the user has opted to include images and the images list is not empty, these images are sent to the LLM's context window.
      2. If the user has chosen not to include images, the summaries stored in the summary list are sent instead.
   2. Regardless of these choices, the original text chunks are always sent to the context window if they are present, but not their summaries.
2. The LLM processes the context and generates a final answer, which is then returned to the user. This approach ensures that the response is both relevant to the user's query and aligned with their preferences regarding content types.
3. Conversation Update: Once the full response is generated, the conversation is updated with the new AI-generated text. The system returns a chat ID and message ID as JSON, allowing the conversation to be tracked and referenced later