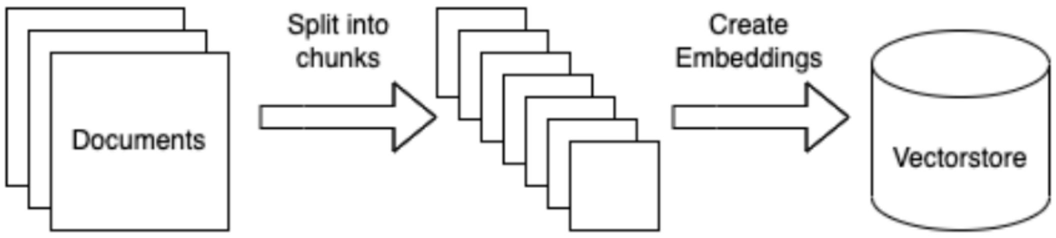
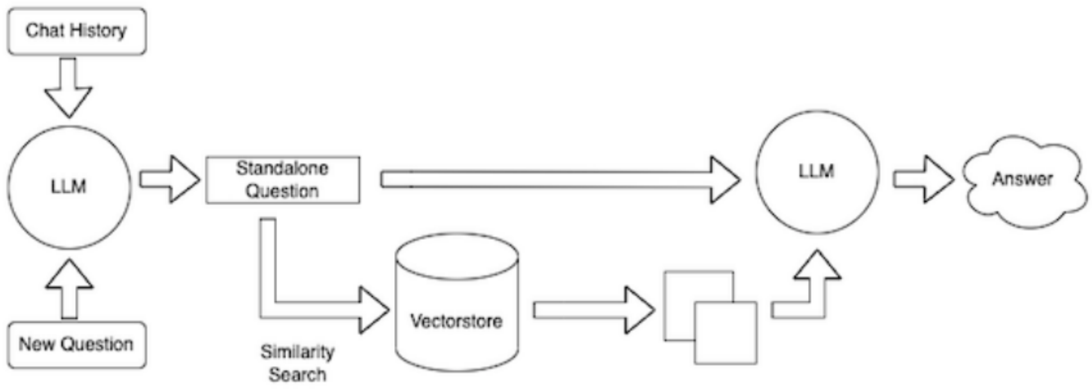
**Design, Code Flow and Ideas:**





**This section explains the detailed process of preparing data for use.**

**Load Data**: The initial step is to load the data and convert it into a structured format compatible with LangChain. Documents in LangChain include both the text content and relevant metadata (origin, etc.).

**Split Text**: To optimize model input, divide documents into smaller text segments. Finding the ideal segment size requires experimentation, as excessively large or small segments can hinder performance.

**Create embeddings and store in VectorStore**: Next, generate embeddings (numerical representations) for each text segment and store them in a vector store. This enables efficient similarity-based searches for relevant information.

Remember to re-run 'ingest_data.py' if you modify the text-splitting process or introduce new data.

**Query Data**: With the data prepared, let's integrate it into a conversational chatbot interface. The ConversationalRetrievalChain in LangChain provides the foundation for question-answering capabilities. Here are key considerations for customization:

- Conversation History: Enable a seamless user experience by maintaining the conversation history for context and follow-up questions.
- QA Prompt: Tailor the question-answering process by customizing the prompt sent to the language model.
- Long Conversations: Condense the chat history and the current query into a single question for lengthy conversations. This maintains focus and prevents retrieval of irrelevant information.
- Source Citation: If desired, configure the model to provide the original source of its answers.
- Language Model: Experiment with different language models supported by LangChain to power your chatbot.

## Core Functionalities

- **Data Ingestion (ingest_data.py):** Loads and processes text documents of various formats for answering questions.
- **Question Answering Web Application (app.py):** Provides a user interface for asking questions about a text document and utilizes the backend for handling question-answering.
- **Question Answering Logic (query_data.py):** Houses the core logic behind retrieving relevant information and formulating answers, involving an OpenAI language model.

**Code Breakdown**

**File: ingest_data.py**

1. **Data Loading (loader)**
   - Imports the UnstructuredFileLoader to handle plain text files.
   - Loads a file named "myresume.txt".
   - (Commented out code demonstrates how to load PDF, Word, and plain text files from a directory).
2. **Text Splitting (text_splitter)**
   - Utilizes CharacterTextSplitter to divide the document into smaller chunks of 600 characters with 100 characters overlap for effective processing.
3. **Embedding Creation (embeddings)**
   - Embeds the text chunks into vector representations using the OpenAIEmbeddings model.
4. **Vector Store Creation (vectorstore)**
   - Employs FAISS.from_documents to construct a vector store (like an index) for efficient similarity-based searches.
   - Serializes and saves the vector store as "vectorstore.pkl" for future use.

**File: app.py**

1. **Imports, Setup, and Configurations**
   - Imports necessary modules, including Gradio for the web interface.
   - (Commented out code demonstrates loading environment variables for OpenAI API keys).
   - Defines functions to set OpenAI API keys and retrieve the query answering chain (get_basic_qa_chain - I'll get into the details of this in query_data.py).
2. **Gradio Interface**
   - Creates a gradio.Blocks to construct the web UI.
   - **Key components:**
     - Title: "Gen-AI-Intro(Know-Me-From-My-Resume)"

        - OpenAI API key input field
        - Chatbot interface
        - Examples of possible questions
        - Powered by LangChain and Demo application attribution
3. **Event Handling**
    - Defines actions triggered by user inputs (submitting questions and setting the API Key).

**File: query_data.py**

1. **Helper Functions**
    - load_retriever(): Loads the FAISS vector store from "vectorstore.pkl".
    - CONDENSE_QUESTION_PROMPT: A LangChain prompt template to rephrase follow-up questions as standalone ones.
    - QA_PROMPT: LangChain prompt template for tailoring the language model to provide answers from a document in the context of HR recruitment.
2. **Answering Models (chain_options)**
    - Provides various flavors of answering models using LangChain:
        - **basic**: A simple Q&A setup.
        - **with_sources**: Returns answers along with the source sections of the document.
        - **custom_prompt**: Employs the QA_PROMPT for custom tailoring.
        - **condense_prompt**: Uses CONDENSE_QUESTION_PROMPT to handle follow-up questions.

**File: cli_app.py**

- **Command-Line Application**
    - Simple command-line interface for the user to select a QA model.
    - Takes the user's question and calls the selected model.
    - Prints the answer (and source documents if the 'with_sources' model is chosen).