Project Overview: Chat with Website Using RAG Pipeline

The project aims to build a Retrieval-Augmented Generation (RAG) pipeline that allows users to interact with structured and unstructured data extracted from websites. The system will crawl and store content, convert it into vector embeddings, and use a vector database for information retrieval. Users can ask natural language queries, and the system will generate contextually accurate responses with the help of a selected Large Language Model (LLM).

PYTHON libraries:

Here's a brief description of each library based on your usage:

1. requests: Used for making HTTP requests to interact with APIs or fetch data from the web.
2. hashlib: Provides hashing algorithms (e.g., SHA-256) to generate hashes of data, often used for data integrity or creating unique identifiers.
3. hmac: A secure method of hashing data using a secret key, commonly used for authenticating API requests or verifying data integrity.
4. json: Used for parsing and working with JSON data, a common format for web services and APIs.
5. BeautifulSoup: A library for parsing HTML and XML documents, primarily used for web scraping.
6. SentenceTransformer: Converts sentences or text into numerical vectors (embeddings) to perform tasks like similarity comparison or clustering.
7. FAISS: A library for efficient similarity search and clustering of high-dimensional vectors, typically used to search and compare embeddings in large datasets.

Approach

1. Data Ingestion:

Input: URLs or list of websites to crawl/scrape.

Process:

Crawl and scrape target websites for content-

- def scrape_website(url) in the code

Scrape website using Selenium for JavaScript-rendered content

- def scrape_website_with_selenium(url):

Extract key data fields, metadata, and textual content.

Segment the content into smaller chunks for better granularity. –

- # Chunking the text

  def chunk_text(text, chunk_size=300)

Convert content chunks into vector embeddings using a pre-trained embedding model.

Store the embeddings in a vector database with metadata for efficient retrieval

2. Query Handling:

Input: User's natural language query.

Process: # Query Handling

- def query_vector_search(query, vector_db, chunks, top_k=5):

Convert the user's query into vector embeddings using the same embedding model.

Perform a similarity search in the vector database to find the most relevant content chunks.

Pass the retrieved chunks to the LLM, along with prompts or agentic context, to generate responses.

3. Response Generation:

Input: Retrieved information from the vector database and the user query.

Process: # Generate a simple response

- def generate_response(query, context):

Use the LLM with retrieval-augmented prompts to generate accurate, context-aware responses.

Ensure factually correct answers by directly incorporating retrieved data into the response.

Output:

The system provides:

Accurate, context-rich answers to user queries.

Fact-based responses generated by the LLM using retrieved website content.

A seamless way to interact with large datasets or websites using natural language.

Example Websites given:

University of Chicago

Stanford University

University of North Dakota

University of Washington