DOCUMENTATION

This application allows users to upload PDFs, which are processed and indexed by the system, enabling them to ask questions about the content. It leverages **Amazon Bedrock** for embeddings, **LangChain** for retrieval-based question answering (RAG), and **FAISS** for vector search. The system consists of two key components:

- **Admin Interface**: Uploads PDFs, processes them, and creates the vector store.
- **User Interface**: Allows users to query the uploaded PDF content using natural language.

Both services are hosted using **Streamlit** and deployed via **Docker** containers.

Models used:

Amazon Titan Embedding G1 - Text

Anthropic Claude 2.1

Admin Interface (admin.py)

- Build Admin Web application where AdminUser can upload the pdf.
- The PDF text is split into chunks
- Using the Amazon Titan Embedding Model, create the vector representation of the chunks
- Using FAISS, save the vector index locally
- Upload the index to Amazon S3 bucket (You can use other vector stores like OpenSearch, Pinecone, PgVector etc., but for this demo, I chose cost effective S3)

Command to Run Admin Application:

```
- docker build -t admin_app .
- docker run -p 8083:8083 admin_app
```

Functionality

1. Upload PDF:
   - Admin can upload a PDF file.
   - The file is processed, split into chunks, and embedded using Amazon Bedrock embeddings.
   - The vector store (FAISS) is created and saved locally.
2. Store in S3:
   - The vector store is uploaded to an S3 bucket to be accessed by the user app.

Key Components:

- **`boto3`**: Interacts with S3 to store the vector index.
- **`PyPDFLoader`**: Loads and splits the PDF into pages.

- **`RecursiveCharacterTextSplitter`**: Splits documents into chunks of manageable size for embedding.
- **`FAISS`**: A vector store that stores document embeddings for efficient similarity search.
- **`BedrockEmbeddings`**: Amazon Bedrock is used to embed the chunks for vector search.

Steps:

1. **File Upload**: The admin uploads a PDF.
2. **Document Splitting**: The document is split into chunks of size 1000 characters with 200 characters overlap.
3. **Vector Creation**: Embeddings are created using Amazon Bedrock and saved into a FAISS index.
4. **Store on S3**: The FAISS index is saved locally and then uploaded to S3 for retrieval by the user app.

User Interface (user.py)

- Build User Web application where users can query / chat with the pdf.
- At the application start, download the index files from S3 to build local FAISS index (vector store)
- Langchain's RetrievalQA, does the following:
    - Convert the User's query to vector embedding using Amazon Titan Embedding Model (Make sure to use the same model that was used for creating the chunk's embedding on the Admin side)
    - Do similarity search to the FAISS index and retrieve 5 relevant documents pertaining to the user query to build the context
    - Using Prompt template, provide the question and context to the Large Language Model. We are using Claude model from Anthropic.
    - Display the LLM's response to the user.

Command to Run User Application:

- docker build -t user_app .
- docker run -p 8084:8084 user_app

Functionality

1. Load Vector Store:
    - Downloads the FAISS vector store from S3.
    - Loads the vector index into memory to make it available for querying.
2. Ask Questions:
    - The user inputs a question about the uploaded PDF.
    - The system retrieves the most relevant document chunks and passes them to an LLM (Amazon Bedrock) to generate a concise answer.

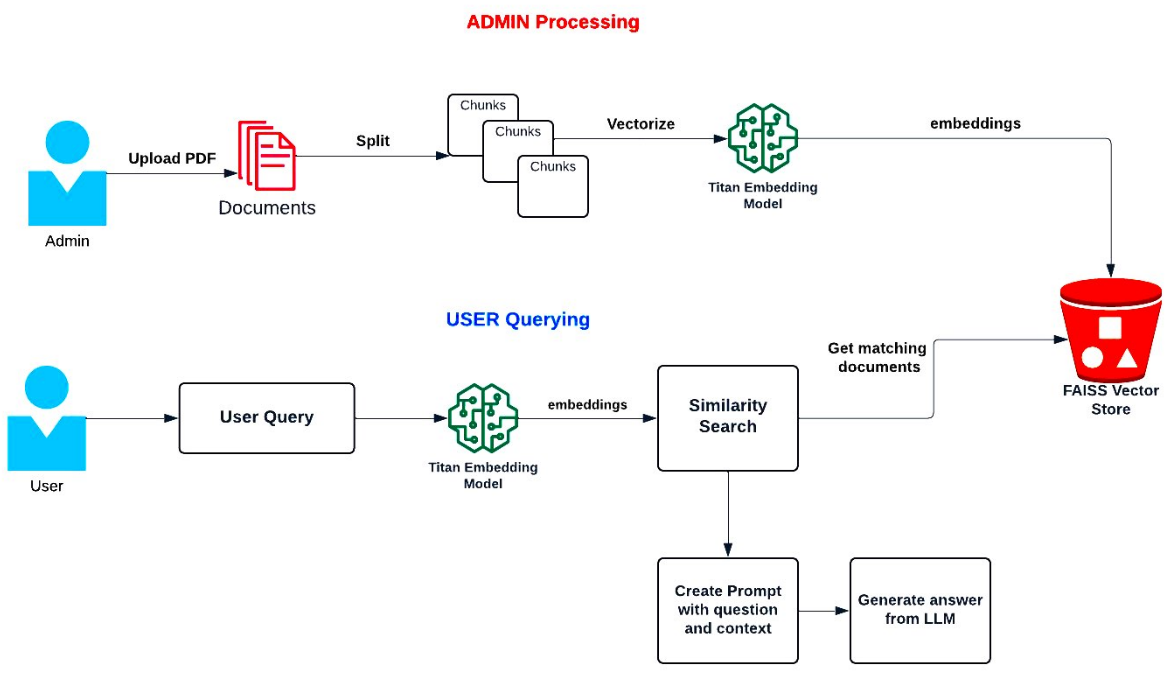
Key Components:

- **`boto3`**: Interacts with S3 to retrieve the FAISS index.
- **`FAISS`**: Loads the FAISS index and retrieves relevant document chunks.
- **`Bedrock`**: Amazon Bedrock is used to run the LLM (Claude-v2 in this case).
- **`RetrievalQA`**: A LangChain chain that handles retrieval-based question answering.

Steps:

1. **Load FAISS Index**: The app downloads the FAISS index from S3.
2. **Ask a Question**: The user inputs a question related to the PDF.
3. **Retrieve & Answer**: Relevant document chunks are retrieved, and an LLM provides an answer.
4. **Display Answer**: The answer is shown to the user along with an indicator of success.

Model Architecture and Docker Image



```
+-------------------------+
|      Admin (Streamlit)   |
|-------------------------|
| - PDF Upload             |
| - Text Splitting         |
| - Embedding with Bedrock|
| - Vector Store Creation |
| - Upload to S3           |
+-------------------------+
          |
          | S3 (Storage)
          v
+-------------------------+
|      User (Streamlit)    |
|-------------------------|
| - Download FAISS Index  |
| - Load Vector Store      |
| - LLM (Claude-v2)        |
| - Query and Answer       |
+-------------------------+
```

Deployment Using Docker

Admin Service (admin.py):

- Dockerfile exposes port 8083 for the admin interface.
- After uploading a PDF, it is split, embedded, and stored in a vector store, which is then uploaded to S3.

```
FROM python:3.11
EXPOSE 8083
WORKDIR /app
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . ./
ENTRYPOINT [ "streamlit", "run", "admin.py", "--server.port=8083", "--server.address=0.0.0.0" ]
```

User Service (user.py):

- Dockerfile exposes port 8084 for the user interface.
- It retrieves the FAISS index from S3 and allows users to ask questions.

```
FROM python:3.11
EXPOSE 8084
WORKDIR /app
COPY requirements.txt ./
RUN pip install -r requirements.txt
COPY . ./
ENTRYPOINT [ "streamlit", "run", "user.py", "--server.port=8084", "--server.address=0.0.0.0" ]
```

Architecture & Approach

1. PDF Ingestion and Chunking:

- **Admin Flow**: The PDF files uploaded by the admin are chunked into smaller parts. Chunking is essential because it allows for more efficient embedding and retrieval. This is handled using `RecursiveCharacterTextSplitter` from LangChain.

2. Embeddings and Vector Store Creation:

- **Embeddings**: The chunks of text are converted into vector embeddings using `Amazon Titan` through the `BedrockEmbeddings` API.
- **Vector Store**: FAISS (Facebook AI Similarity Search) is used to store these embeddings in a local index. The index is serialized and stored in S3.

3. Retrieval-Augmented Generation (RAG):

- **User Flow**: When a user asks a question, the system queries the FAISS index to retrieve the most relevant document chunks based on the user's question. This context is passed to the `Claude v2` model via a carefully designed prompt.
- **Answer Generation**: Using the retrieved context, the `Claude v2` model generates a response. The RAG approach ensures that responses are grounded in the content of the uploaded PDF.

4. Deployment:

- Both the Admin and User applications are containerized using Docker, making it easy to deploy the applications in any environment. The admin app is accessible at port `8083` and the user app at port `8084`.

Key Components of RAG App

A. Amazon Bedrock:

- Provides language models for embeddings (`Titan` for text embeddings) and LLM inference (`Claude v2` for question-answering).

B. FAISS:

- Used for storing and retrieving document embeddings, enabling efficient vector similarity searches.

C. LangChain:

- Provides utility functions for text splitting, embeddings, and building QA chains.

D. S3:

- Acts as a storage backend for the serialized FAISS vector indexes.