Introduction

Sheep Bot is a playful yet functional web-based assistant designed to help users interact with documents, such as PDFs, in a simple and intuitive way. The core functionality allows users to upload files, manage them, and ask questions directly related to the document content — making it especially useful for students, educators, or professionals who work with a lot of reading material. The project aims to simplify document navigation by combining file management with conversational AI, all wrapped in a light-hearted, sheep-themed interface.

The current interface already has a clean layout with a soft color scheme, a central chatbot area, and a sidebar for file management. To make the interface more engaging and unique, several design enhancements can be introduced. For example, a **sheep-themed personality** can bring fun and friendliness to the user experience. Adding animations — like a sheep walking across the screen during loading — can make even wait times enjoyable.

Design can be further enriched with **interactive document previews**, file thumbnails, and hover effects for actions like delete or rename. The chat input box can suggest possible queries such as “Summarize this file” or “Define technical terms,” guiding the user intuitively. Incorporating **light/dark mode** with a fun toggle animation, like a sheep jumping over a fence to change themes, would add a delightful touch.

Additionally, turning the bot into a **chat-style interaction system** with message bubbles, typing indicators, and AI responses in a conversational tone would greatly enhance usability. Pairing this with a cartoon sheep avatar and playful responses like “Baa! Got it!” adds charm and branding value.

In short, Sheep Bot blends functionality with personality. It’s not just another bot — it’s your woolly smart assistant that makes working with documents more fun, visual, and interactive.

System Architecture

- The **Document Research Chatbot** is an intelligent tool that helps users understand large documents through natural language queries. It combines a simple interface with powerful backend processing to deliver fast and useful answers.

- The **User Interface (UI)** acts as the main entry point. It can be built using platforms like **Streamlit**, **Gradio**, or tools like **Swagger UI**. Users can upload files, ask questions, and get answers in a chat-like format.

- The **API Layer**, built with **FastAPI**, handles user input and routes it to the right modules. It ensures smooth communication between the front end and backend.

- The **Document Processing Module** prepares uploaded files. It first breaks documents into smaller parts (chunking), then converts these chunks into **semantic embeddings** using **Sentence Transformers**. This makes the data easier to search and understand.

- A **Vector Database** like **ChromaDB** stores these embeddings, allowing the system to quickly find the most relevant document sections in response to user queries.

- The **Clustering Module** uses **KMeans** to group similar content together, helping users see common themes or topics across documents.

- A **Citation Module** links each answer to its source within the document, helping maintain accuracy and supporting proper referencing.

- Finally, the system supports flexible deployment on platforms like **Hugging Face Spaces**, **Render**, or **Railway**, making it easy to access and scale.

- Overall, this chatbot is a complete solution for exploring, understanding, and getting answers from complex documents in a simple and interactive way.

Methodologies

To process and understand large sets of documents, a few important techniques are used. These help in breaking down content, finding relevant parts, grouping similar ideas, and providing references. Here's how each step works:

1. **Document Chunking:**

   Instead of working with full documents, they are split into smaller parts like paragraphs or sections. This helps in understanding the content more clearly and finding relevant parts faster.

2. **Semantic Embedding:**

   Each small chunk is converted into a numeric form (called a vector) using a pre-trained model known as Sentence Transformers. This step turns text into a format that computers can compare easily. The vector represents the meaning of the text, not just the words.

3. **Semantic Search:**

   When a user asks a question or types a query, that input is also turned into a vector. The system compares this query vector with the document vectors using a method like cosine similarity. This helps in finding the chunks that are closest in meaning to the query.

4. **Clustering:**

   Chunks that are similar in meaning are grouped together using a method called KMeans clustering. This helps in identifying common topics or themes across different documents.

5. **Citation Extraction:**

   Each chunk that is shown as a result includes information about where it came from in the original document. This ensures that users can trace the source of the information, which is helpful for citations or further reading.

These methods work together to make document search smarter and more useful.

Technologies & Frameworks

This project uses several modern tools and frameworks to handle text processing, search, clustering, and deployment efficiently. Here's a breakdown of the main technologies used:

1. **FastAPI:**

   FastAPI is a fast and lightweight Python framework used to create web APIs. It allows developers to build RESTful services quickly, with automatic documentation and high performance. In this project, FastAPI handles user requests and delivers results from the backend.

2. **Sentence Transformers:**

   This is a powerful Python library that turns sentences or chunks of text into vectors — numerical formats that represent the meaning of the text. These vectors make it possible to compare different pieces of text based on their meaning, which is crucial for semantic search.

3. **ChromaDB:**

   ChromaDB is an open-source vector database. It is used to store the vector representations (embeddings) of text. It allows for fast and accurate searching of similar vectors. When a user enters a query, the system compares the query's vector with those stored in ChromaDB to find the most relevant results.

4. **scikit-learn:**

   This popular machine learning library is used mainly for clustering in this project. It helps group similar document chunks using the KMeans algorithm, allowing the system to identify themes or patterns in the text data.

5. **Deployment:**

   The project is designed to run on the cloud. The deployment platform hasn't been finalized yet, but options include Hugging Face Spaces, Render, and Railway. These platforms allow the app to be accessed online easily.

Together, these tools create a smart, efficient system for understanding and organizing text.

Design Decisions & Challenges

During the development of this project, several key design decisions were made to ensure that the system remains modular, efficient, and easy to deploy.

1. **Modular Design:**

   The project is divided into separate parts such as API handling, document processing, storage, and clustering. This modular approach makes it easier to manage and scale each component without affecting the others.

2. **Resource Optimization:**

   To make sure the system works on free or low-cost cloud platforms, lightweight models and optimized document chunking strategies were used. This helps in reducing the load on memory and processing time.

3. **Error Handling:**

   Strong error handling is built into the code. Whenever something goes wrong, the system shows clear and helpful messages instead of crashing. This improves reliability and user experience.

4. **Deployment Flexibility:**

   The code is Docker-ready and not tied to any one cloud provider. This means it can be deployed on various platforms like Hugging Face Spaces, Render, or Railway without needing major changes.

Personal Review:

While building this project, I faced many challenges. I had trouble working with NLTK, connecting the frontend to the backend, and deciding how to handle APIs. The biggest challenge was understanding and implementing the logic — but I managed to solve all these issues. However, hosting has been the toughest part. I've already spent two days trying different methods, and still, the chances of hosting it before the deadline are low. This remains the only major issue I couldn't fully resolve.

Scalability & Deployment Considerations

1. Scalability:

The system is built using a modular design, where different tasks like handling APIs, processing documents, or storing data are separated into their own modules. This setup allows for **horizontal scaling**, meaning we can increase capacity by adding more workers or services without changing the core logic. For example, if more users start using the system at once, we can add more API instances or switch to a cloud-based vector database like Pinecone or Weaviate to handle the load better. This makes the system flexible and future-ready.

2. Deployment:

The project is ready to be deployed on any modern cloud platform that supports Docker or Python-based web applications. It does not depend on one specific service, which means it can be moved or hosted on platforms like **Render, Hugging Face Spaces, or Railway**, depending on what resources are available and how accessible the app needs to be for users. The codebase includes Docker support to make the deployment process faster and more consistent across platforms.

This design ensures that the system is not only ready for current needs but can also grow and adapt as demand increases. However, final deployment remains a challenge due to current hosting issues.

7. References

- **FastAPI Documentation** – Official docs for FastAPI, the web framework used to build the backend APIs.

  https://fastapi.tiangolo.com

- **ChromaDB Documentation** – Guides and examples for using ChromaDB, the vector database used for storing and searching embeddings.

  https://docs.trychroma.com

- **Sentence Transformers** – Library used to generate semantic embeddings from text using pre-trained transformer models.

  https://www.sbert.net

- **scikit-learn Documentation** – Used for machine learning utilities like KMeans clustering in the project.

  https://scikit-learn.org

Conclusion

In summary, Sheep Bot successfully demonstrates a combination of practical AI techniques and creative UI design. While some deployment challenges remain, the core functionalities like document parsing, theme detection, and semantic search were implemented effectively. This project reflects both learning and problem-solving throughout its development.