CSCI 5253
Datacenter Scale Computing
Project Proposal

Team Members :
Rahul Ganesan
Yashwant Gandham

1. Title: DocChat - AI Chat for Your Documents!

Description: DocChat is a cloud based AI service that allows users to chat with their own documents, with no query limits. Users upload their PDFs, and the system automatically processes them and lets them ask questions about their content. From academic papers to reports, DocChat helps you find insights instantly, without going through the entire document yourself.

We plan to utilize LangChain to orchestrate document ingestion, embedding retrieval, and conversational memory, ensuring smooth, context-aware chats between users and their private knowledge bases. A specific list of software and hardware tools is mentioned below.

2. Project Goals

1. **Enable Document Based AI Chat :**

   Our primary goal is to empower users to interact with their own PDF documents using chat in a conversational manner. This provides them with answers and personalized insights based on their document..

2. **Automatic Document Processing:**

   By parsing PDFs into text and smaller chunks for efficient processing, we generate embeddings for the document. This is the core for finding relevant information related to the user query.

3. **Information Retrieval and Context Aware Conversations:**

   The embeddings generated are stored in a vector database and uses LLM to generate answers corresponding to the user query based on document embeddings/context. This retrieves the most relevant text chunks as a response.

4. **End To End User Experience with Privacy**

   The response from the LLM is presented to the user in a chat interface. The documents uploaded are not made public (restricted to each user). They are not shared with anybody else. Overall, the application is SaaS, with an aim to provide a smooth end to end experience to all users.

3. List of Hardware and Software Components

- **Frontend:** React + TailwindCSS
- **Backend API:** FastAPI / REST APIs for sending and receiving data
- **LLM:** Google Gemini
- **Embeddings:** Google Gemini/Vertex AI Embeddings API
- **Vector DB:** FAISS/Pinecone
- **PDF Parsing:** PyPDF2/pdfplumber
- **LangChain** for Orchestration (retriever, memory, chain)
- **Storage:** Google Cloud Storage for storing PDF
- **Compute :** Google E2 Medium VMs

4. Architectural Diagram

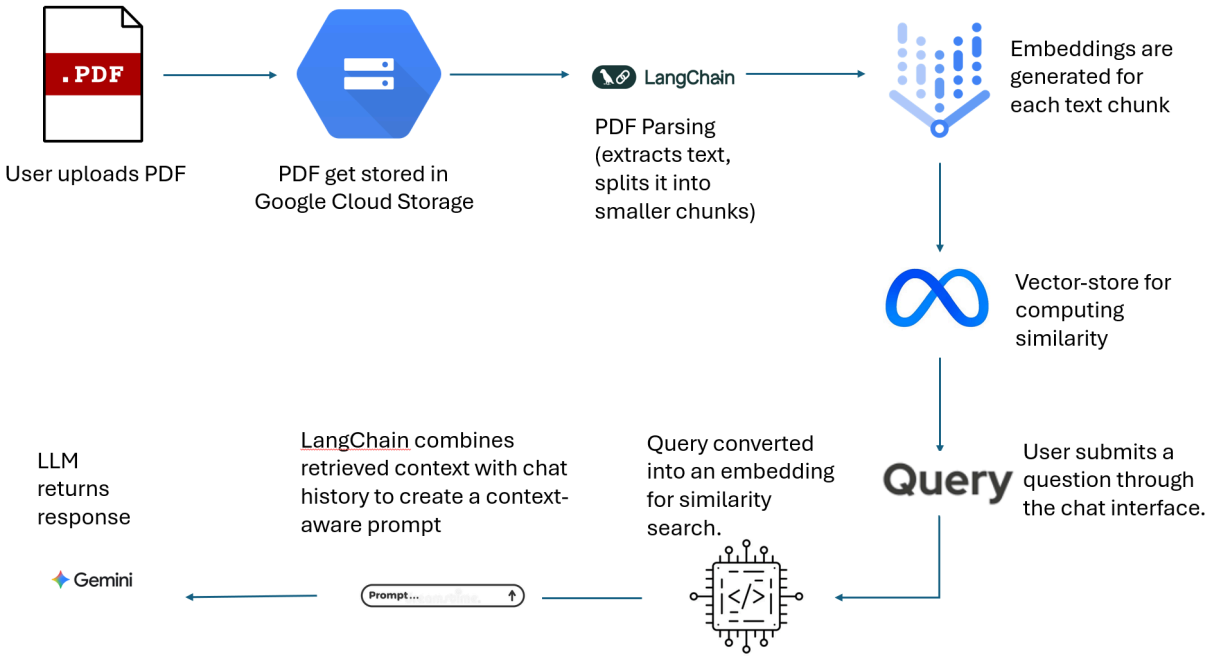


Figure 1: Diagram illustrating interactions between different system components

5. Description of the interaction of the different software and hardware components

Front-End Layer

- **Components:** React + TailwindCSS
- **Hardware:** Client Browser

Role: User interface for document upload and chat interaction

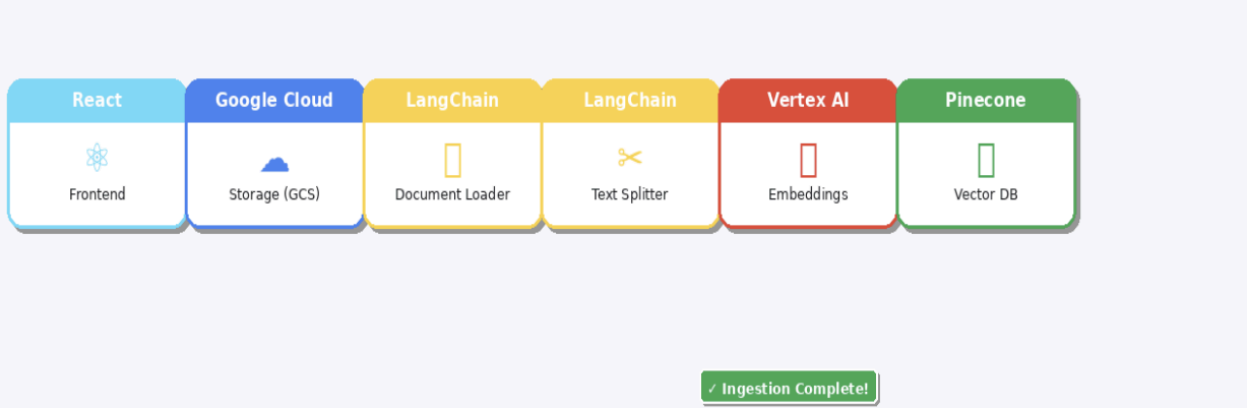
Interactions:

- Sends HTTP requests to REST API backend for document upload
- Streams user queries to backend API
- Receives and displays AI generated responses
- Manages user session and conversation history in browser state

We have split our Main workflow into 2 parts

1. Document Ingestion Phase
2. Query Phase

Document Ingestion Pipeline



Gif 1: Document ingestion pipeline workflow

To view Interactive Flow: rag_ingestion_flow.gif

1. PDF Upload Flow

a) React Frontend → FastAPI Backend

- **Protocol:** HTTP POST with multipart/form-data
- **Data:** PDF file binary + metadata (filename, user_id, timestamp)
- **Backend Processing:** REST/FAST API receives file, validates format/size, generates unique document ID

b) FastAPI Backend → Google Cloud Storage

- **Protocol:** Google Cloud Storage API
- **Data:** PDF file binary with metadata tags

2. Document Processing

a) Google Cloud Storage (GCS) → LangChain Document Loader

- **LangChain Component:** PyPDFLoader or PDFPlumberLoader
- **Process:**
    - Fetches PDF from GCS using Cloud Storage API
    - Extracts text content using PyPDF2/PDFPlumberLoader
    - Preserves document structure (pages, metadata)
    - Returns Document Objects with page_content and metadata

b) LangChain Document Loader → LangChain Text Splitter

- **LangChain Component:** RecursiveCharacterTextSplitter or TokenTextSplitter
- **Configuration:**
    - **Chunk size:** 512-1024 tokens (configurable)
    - **Chunk overlap:** 50-100 tokens (maintains context between chunks)
    - **Separators:** paragraphs, sentences, words (hierarchical splitting)
- **Output:** List of text chunks with metadata (source document, page number, chunk index)
- **Purpose:** Creates manageable text segments for embedding generation and retrieval

3. Embedding Generation

a) Text Chunks → Vertex AI Embeddings API

- **Protocol:** Google Cloud Vertex AI REST API
- **Auth:** Service account credentials with Vertex AI permissions
- **Model:** text-embedding-004
- **Process:**
    - LangChain sends batches of text chunks (batch size: 5-10 for rate limiting)
    - Vertex AI returns dense vector embeddings (768 or 1536 dimensions)
    - Each chunk gets a unique vector representation capturing semantic meaning

- **Rate Limiting:** Handles API quotas with exponential backoff retry logic

4. Embeddings → Vector Database

- **Protocol:** Pinecone/FAISS (REST API over HTTPS)
- **LangChain Component:** FAISS/PineconeVectorStore

Query Processing Pipeline

Gif 2: User Query Flow

To view the interactive Flow: rag_query_flow.gif

1. Similarity Search

a) Query Embedding → Pinecone Vector Database

- **LangChain Component:** FAISS/PineconeVectorStore.similarity_search()
- **Search Parameters:**
  - Top K: 3-5 most similar chunks
  - Filter: `user_id` and optionally `document_id` (enforces data isolation)
  - Similarity threshold: Optional minimum score (e.g., 0.7)
- **Output:** List of most relevant text chunks with similarity scores

2. Answer Generation

a) Assembled Prompt → Google Gemini LLM

- **Protocol:** Vertex AI Generative AI API
- **Model:** gemini-1.5-pro
- **LangChain Component:** ChatGoogleGenerativeAI or ChatVertexAI
- **Configuration:**
    - Temperature: 0.3-0.7 (controls creativity vs. consistency)
    - Max output tokens: 512-2048
    - Top-p: 0.9
    - Safety settings: Block harmful content

b) LLM Response → FastAPI Backend → React Frontend

- **Backend Processing:**
    - Receives generated response from Gemini
    - Stores conversation turn in memory/database
    - Formats response with citations (chunk sources, page numbers)
    - Returns to frontend via HTTP response or WebSocket stream
- **Frontend Display**
    - Renders markdown-formatted response
    - Shows source citations as clickable links
    - Updates conversation history UI
    - Maintains scroll position and loading states

Data Flow Summary

1. Ingestion Path

```
User → React → FastAPI → GCS → LangChain Loader (PyPDF2) →
LangChain Splitter → Vertex AI Embeddings → Pinecone
```

2. Query Path

```
User → React → FastAPI → Vertex AI Embeddings (query) →
Pinecone (search) → LangChain (context assembly) →
Gemini LLM → FastAPI → React → User
```

6. A description of how you will debug your proposed project and what training or testing mechanism will be used.

Common Issues and Solutions

Issue : PDF Parsing

- **Problem:** Garbled text extraction
- **Debug:** Check encoding, try pdfplumber if PyPDF2 fails
- **Solution:** Implement fallback parser selection

Issue: Chunking

- **Problem:** Chunks too small/large or no overlap
- **Debug:** Log chunk sizes and inspect boundaries
- **Solution:** Tune chunk_size and overlap parameters

Issue : Retrieval Issues

- **Problem:** Low similarity scores or irrelevant results
- **Debug:** Check embedding dimensions match, inspect query preprocessing
- **Solution:** Adjust embedding model or add query rewriting

LLM Issues:

- **Problem:** Hallucinated answers not grounded in context
- **Debug:** Log retrieved chunks and LLM prompt
- **Solution:** Strengthen system prompt, adjust temperature, add citation requirements

Model Approach (Tuning and Parameter Selection)

1. **Experiment with:**
    a. **Chunk size:** Test 256, 512, 1024 tokens
    b. **Top-K retrieval:** Test k=3, 5, 10
    c. **LLM specific parameters :** Temperature, Prompt (with Prompt Engineering)

Testing and Debugging Approach

1. **Test individual functions in isolation:**
    a. **PDF parsing:** Verify text extraction, page count, metadata preservation
    b. **Chunking:** Validate chunk sizes, edge cases (very short/long documents)
    c. **API endpoints:** Test request validation, error handling, response formats

Integration Testing (Pipeline-Level)

1. **Test component interactions:**
    a. **Ingestion pipeline:** Upload PDF. Store the embeddings in vector DB
    b. **Query pipeline:** Submit query, and verify relevant chunks retrieved. Validate answer quality (if it's relevant to the question)
    c. **Conversation memory:** Multi-turn conversation maintains context
    d. **Error propagation:** Simulate failures (API timeout, storage full) and verify graceful degradation

End-to-End Testing (User Workflow)

1. **Simulate real user interactions:**
    a. Upload document through UI
    b. Wait for processing completion
    c. Ask questions and verify responses
    d. Check citations link to correct source pages

7. Explain why you believe this project idea will meet the eventual project requirements (use of four different cloud technologies)

DocChat satisfies the requirement of using four datacenter software components.
First, we utilize the RPC/API interface through REST/FastAPI for backend communication and REST APIs to the Vertex AI/Gemini, and Pinecone service. Second, Storage Services are implemented via Google Cloud Storage for persistent PDF document storage. Third, we plan to use FAISS/ Pinecone, which serves as a Vector database to store Embeddings with metadata in key-value structure to optimize the similarity search. Fourth, our infrastructure runs on a VM using Google Cloud E2 Medium to orchestrate the RAG pipeline. This architecture showcases a good use of all the modern datacenter technologies for building a SaaS based RAG system.