Project Plan: Mini RAG Application

This document outlines the step-by-step plan to build, host, and document the Mini RAG (Retrieval-Augmented Generation) application as per the assessment requirements.

Overall Goal

Build and deploy a functional RAG web application where a user can input text, and then ask questions about that text. The application will retrieve relevant information, rerank it for quality, and use a Large Language Model (LLM) to generate a final answer with citations.

Project Steps

We will build this project in five distinct phases:

Step 1: Project Setup & Foundation

- **Goal:** Prepare the development environment, choose our technology stack, and set up version control.
- **Tasks:**
  1. Initialize a public GitHub repository.
  2. Structure the project (we'll use a single-file React frontend and a Python backend).
  3. Select our tools (e.g., Pinecone for the vector DB, Gemini for embeddings, Cohere for reranking, and Gemini for the LLM).
  4. Set up environment variable files (.env and .env.example) to manage API keys securely.
  5. Create the initial README.md file to be updated as we progress.

Step 2: Backend - The RAG Engine

- **Goal:** Develop the core logic for the entire RAG pipeline. This is the "brain" of our application.
- **Tasks:**
  1. **Document Chunking:** Write a Python script to split the user's input text into smaller, manageable chunks with a defined size and overlap.
  2. **Embedding & Storage:**
     - Integrate the embedding model to convert text chunks into vector embeddings.
     - Set up the cloud vector database and write the logic to upload (upsert) these embeddings along with their metadata (e.g., source text, chunk number) for citations.
  3. **Retrieval & Reranking:**
     - Develop the function to take a user's query, embed it, and retrieve the most similar chunks from the vector database.
     - Integrate the reranker model to improve the relevance of the retrieved chunks.
  4. **Answer Generation:**

        - Create a well-formatted prompt that includes the user's query and the reranked context.
        - Call the LLM API with this prompt to generate the final, grounded answer.

Step 3: Backend - API Server

- **Goal:** Expose the RAG engine's functionality through web APIs so our frontend can communicate with it.
- **Tasks:**
    1. Build a simple web server (e.g., using FastAPI in Python).
    2. Create a /process-text endpoint that handles text uploading, chunking, embedding, and storage.
    3. Create a /query endpoint that takes a user question and returns the final answer with citations.
    4. Implement basic error handling and logging.

Step 4: Frontend Development

- **Goal:** Create a simple and intuitive user interface for the application.
- **Tasks:**
    1. Develop a single-file React application.
    2. Build the necessary UI components: a text input area, a query box, and a display panel for the answer and its sources.
    3. Implement state management to handle user input, loading states, and final results.
    4. Connect the UI to the backend APIs, ensuring smooth data flow and error handling.

Step 5: Deployment & Documentation

- **Goal:** Make the application publicly accessible and provide comprehensive documentation.
- **Tasks:**
    1. Deploy the backend API to a free hosting service (like Railway or Render).
    2. Deploy the React frontend to a free hosting service (like Vercel or Netlify).
    3. Thoroughly test the live application.
    4. Complete the README.md with all required sections:
        - Live URL.
        - Architecture diagram and setup instructions.
        - Details on the models and strategies used.
        - A "Remarks" section for any limitations or potential improvements.

Ready to start? We'll begin with **Step 1: Project Setup & Foundation**. Let me know when you're ready to dive in!