Project Summary: AI Copilot for Renewable Energy Data Rooms

(Elevator Pitch)

Problem: Renewable energy professionals (analysts, investors, developers) waste valuable time manually sifting through dense project data rooms (PDFs, DOCX) to find critical information. This process is slow, inefficient, and risks overlooking key details.

Solution: We built the AI Copilot, an intelligent assistant powered by Google Gemini. Users upload project documents, and the Copilot answers natural language questions directly based on the provided content.

Who Benefits: Anyone involved in renewable energy project analysis, due diligence, or investment decision-making.

Impact & Outcome: The AI Copilot significantly accelerates information retrieval. It provides accurate, context-aware answers complete with direct citations (source file and page number), improving efficiency and reducing oversight. Key features working today include multi-format document processing (PDF, DOCX, TXT), Retrieval-Augmented Generation (RAG) using FAISS vector search for grounded answers, and dynamic example question suggestions based on uploaded files, all accessible through an interactive Streamlit interface. This tool transforms hours of manual review into seconds of targeted querying.

Video Demo Script Outline (Points)

1. **Introduction & Problem (Goal: ~10-15 sec)**
   - **Problem:** Navigating complex and lengthy renewable energy project documents (data rooms) to find specific information is time-consuming and inefficient for analysts, developers, and investors.
   - **Solution:** Introducing the AI Copilot – an intelligent assistant that allows users to upload project documents (PDF, DOCX, TXT) and ask natural language questions.
   - **Impact:** Get quick, accurate answers with direct citations to the source documents, significantly speeding up due diligence and analysis.
2. **Tech Stack Overview (Goal: ~10-15 sec)**
   - **Core:** Python-based application.
   - **AI/ML:**
     - **LLM:** Google Gemini 1.5 Pro (via google-generativeai library) for question answering and generating example questions based on context.
     - **Embeddings:** Google's embedding-001 model for converting text chunks

and queries into vectors.

- **Data Processing & Retrieval:**
  - **Document Parsing:** PyMuPDF (fitz) for PDF text extraction, docx2txt for Word docs.
  - **Text Chunking:** RecursiveCharacterTextSplitter (likely from LangChain, though not explicitly imported in document_processor.py).
  - **Vector Store:** FAISS (Facebook AI Similarity Search) for efficient similarity search between query and document chunk embeddings (Retrieval-Augmented Generation - RAG core).
- **UI:** Streamlit for the interactive web interface (file upload, chat, document management).
- **Workflow:** Documents are processed, chunked, embedded, and indexed in FAISS. User queries are embedded, relevant chunks retrieved via FAISS, and passed to Gemini along with the query to generate a cited answer.

3. **Implementation Highlights (Goal: ~15-20 sec)**
   - **RAG Pipeline:** Successfully implemented the core RAG architecture, retrieving relevant text chunks using FAISS vector search before calling the LLM, ensuring answers are grounded in the provided documents.
   - **Citation Generation:** The LLM prompt (qa_engine.py) is specifically engineered to require the model to output both the answer *and* structured citation data (filename, page, relevant quote), which is then parsed and displayed.
   - **Dynamic Example Questions:** Implemented a feature (generate_example_questions in qa_engine.py) where the assistant suggests relevant questions based on the filenames of the currently uploaded documents, improving user guidance.
   - **Modular Design:** Code is structured into separate modules (document_processor.py, document_store.py, qa_engine.py, app.py) for better organization and maintainability.
4. **Challenges & Limitations (Goal: ~15-20 sec)**
   - **LLM JSON Formatting:** Ensuring the Gemini model consistently returned valid JSON containing both the answer and the structured citations required careful prompt engineering and adding fallback parsing logic (extract_citations_from_text in qa_engine.py).
   - **Chunking Strategy:** Finding the optimal chunk_size and chunk_overlap in document_processor.py is crucial for balancing context granularity and retrieval effectiveness – this likely required some experimentation.
   - **Embedding Relevance:** Ensuring the embedding-001 model captured the semantic meaning effectively for similarity search within the specific domain

of renewable energy documents.
    - **Scalability:** The current FAISS implementation runs in memory (document_store.py), limiting the total size of documents that can be processed simultaneously. A persistent vector database would be needed for larger data rooms.
    - **Content Types:** Currently handles text well but doesn't interpret complex tables, images, or scanned (non-searchable) PDFs effectively.
5. **Optional Reflection (Goal: ~5 sec)**
    - **Takeaway:** The quality of the RAG system heavily depends on the synergy between effective document chunking, relevant embedding models, and precise prompt engineering for the LLM.

1-Page Report Sections

1. **Challenge Tackled:**
    - **Problem:** Professionals in the renewable energy sector (investors, analysts, developers) face significant challenges in efficiently extracting key information and insights from large volumes of project documentation (technical specifications, permits, financial models, legal agreements) stored in data rooms. Manual review is slow and prone to oversight.
    - **User:** Professionals needing to perform due diligence, analysis, or information retrieval on renewable energy projects.
    - **Solution:** Developed an AI-powered assistant ("AI Copilot") using a Retrieval-Augmented Generation (RAG) approach. Users upload project documents, and the assistant answers natural language questions with answers directly sourced and cited from the provided documents.
2. **Tools / ML Models Used:**
    - **LLM (Reasoning & Generation):** Google Gemini 1.5 Pro (google-generativeai) - Used for synthesizing answers based on retrieved context and generating dynamic example questions.
    - **Embeddings:** Google embedding-001 (google-generativeai) - Used to create vector representations of document chunks and user queries for similarity search.
    - **Vector Store:** FAISS (faiss-cpu) - Used for efficient indexing and similarity search of document chunk embeddings.
    - **Document Parsing:** PyMuPDF (fitz), docx2txt - Used to extract text content from PDF and DOCX files.
    - **Text Splitting:** RecursiveCharacterTextSplitter (LangChain conceptually) - Used to break down large documents into smaller, manageable chunks for embedding.

    - **Web Framework:** Streamlit (streamlit) - Used to build the interactive user interface.
    - **Core Language:** Python
3. **What Worked Well:**
    - The RAG pipeline effectively retrieved relevant context from uploaded documents to answer specific user questions.
    - The system successfully generated answers grounded in the document content, providing citations (filename and page number) alongside the answer.
    - The dynamic example question generation feature provides useful, context-aware starting points for users based on uploaded file names.
    - The Streamlit UI provides a functional and straightforward interface for document upload, interaction, and viewing results.
    - The modular code structure (app.py, qa_engine.py, etc.) facilitated development and integration.
4. **What Was Challenging:**
    - **Prompt Engineering for Citations:** Getting the LLM to reliably return both a coherent answer *and* accurate, structured JSON citation data (including relevant text excerpts) required iterative refinement of the prompt in qa_engine.py. Implemented fallback text-based citation extraction for robustness.
    - **Document Complexity:** Handling variations in document formatting, especially within PDFs, could sometimes affect text extraction quality (document_processor.py). Poor quality scans or complex layouts remain a limitation.
    - **Context Window Management:** Balancing the number of retrieved chunks (max_context_chunks in qa_engine.py) to provide sufficient context to the LLM without exceeding its input limits.
    - **FAISS Index Management:** Ensuring the in-memory FAISS index (document_store.py) was correctly updated (additions/deletions) and mapped to chunk metadata, especially with the use of IndexIDMap.
5. **How You Spent Your Time (Example 24h Breakdown):**
    - **0–3h:** Planning (Refining core goal, features based on plan doc), Environment Setup, Initial Library Selection.
    - **3–8h:** Document Processing (Implementing document_processor.py - text extraction, cleaning, chunking).
    - **8–14h:** Embedding & Vector Store (Implementing document_store.py - Gemini embeddings, FAISS indexing/search/deletion).
    - **14–19h:** QA Engine & Prompting (Implementing qa_engine.py - RAG logic,

prompt engineering for answers and citations, dynamic examples).

- **19–22h:** UI & Integration (Building app.py with Streamlit, connecting modules).
- **22–24h:** Testing, Debugging (JSON parsing, citation accuracy), Final Touches.

6. **Optional Additions:**
    - *(Placeholder: Could include a screenshot of the Streamlit UI here)*
    - **API References:** Google AI for Python SDK (google-generativeai), Streamlit, FAISS library, PyMuPDF.
    - **One-sentence reflection:** "If we had 24 more hours, we'd integrate a persistent vector database for scalability and add support for analyzing tabular data within documents."