Expert Blueprint: LangChain-Based PDF Question-Answering Agent using the Gemini API

I. Executive Summary: The EONVERSE Gemini RAG Agent Blueprint

A. Project Goals and Architectural Overview

The objective of this project is to construct a production-grade, highly reliable Retrieval-Augmented Generation (RAG) system specifically designed for conversational question-answering over unstructured PDF data. This solution leverages the Python-based LangChain framework and the Google Gemini API to deliver rapid, accurate, and contextually grounded responses. The primary deliverable is a functional, elegant Streamlit application [1] demonstrating advanced RAG capabilities suitable for a competitive technical evaluation.

Architectural Thesis: The RAG 2.0 Pipeline

The architecture adopts a modular RAG pipeline structure, often referred to as RAG 2.0, which clearly separates the computationally expensive data ingestion process from the latency-sensitive retrieval phase.[2]

1. **Indexing Pipeline (Offline Process):** This pipeline is responsible for data ingestion, including loading documents from the PDF source using PyPDFLoader, semantically segmenting the text via RecursiveCharacterTextSplitter [3], converting the segments into high-density vectors using GoogleGenerativeAIEmbeddings [4], and storing these vectors

in the FAISS vector index.[5] This process is treated as a one-time setup or a resource-cached operation within the Streamlit session.
2. **Retrieval and Generation Pipeline (Runtime Process):** This critical component handles the real-time user query. It is orchestrated by the LangChain ConversationalRetrievalChain, which integrates chat history, retrieves relevant documents from FAISS, and synthesizes the final answer using the ChatGoogleGenerativeAI model.[6]

The system maximizes modularity, allowing developers to swap out components—such as replacing FAISS with ChromaDB or migrating to a different embedding model—without altering the application's core logic.[8]

B. Key Technical Differentiators for Competition Judging

To achieve a high score in a competitive challenge, the agent must surpass a simple query-response loop by demonstrating advanced engineering rigor.

- **Conversational Context Mastery:** The system employs the ConversationalRetrievalChain to address the inherent challenge of context loss in RAG systems. It uses the ongoing chat history alongside the new user question to synthesize a refined, contextually complete "standalone question".[6] This rephrasing capability ensures that follow-up questions (e.g., "What about the second one?") still trigger highly relevant document retrieval, showcasing robust memory management.
- **Source Citation and Traceability:** A crucial element for factual reliability is verifiability. The agent is engineered to extract and display document metadata—specifically the source filename and the relevant page number(s)—alongside the generated answer.[9] This post-retrieval processing layer transforms the output from a simple black-box answer into an auditable response, significantly increasing user trust and technical credibility.
- **Robustness and Efficiency:** The indexing pipeline is hardened against external failures, particularly API rate limits, which commonly affect large embedding operations. The inclusion of sophisticated retry logic and concurrency limits during the vectorization phase demonstrates mature development practices necessary for enterprise-ready frameworks.[11] The strategic use of Streamlit's caching further ensures the heavy indexing computation is only executed once per session, guaranteeing minimal latency during the live demonstration.

II. Foundational Technology Stack Justification

The selection of the technology stack is driven by the requirements for speed, stability, competitive differentiation, and seamless integration.

A. Orchestration Framework and LLM Integration

The project relies on Python as the core programming language, coupled with the LangChain framework for orchestration.[13] LangChain is specifically chosen because it provides critical abstractions for connecting Large Language Models (LLMs) with external data sources, simplifying the complex RAG workflow into manageable components like chains, loaders, and agents.[14]

Gemini API Selection and Optimization

The Gemini ecosystem provides a unified solution for both generative capabilities and high-quality embeddings.

1. **LLM Selection (gemini-2.5-flash):** The gemini-2.5-flash model is selected for the runtime chain (ChatGoogleGenerativeAI). This model offers an exceptional balance between high-speed inference and advanced reasoning capabilities.[7] Its low latency is essential for maintaining a fluid, conversational user experience in the Streamlit application, especially when performing the two-step RAG process (rephrasing the question, then generating the final answer).[6]
2. **Embeddings Selection (models/gemini-embedding-001):** Using the dedicated GoogleGenerativeAIEmbeddings model ensures vector quality and compatibility within the Google ecosystem.[4] This unified approach guarantees optimized data handling and avoids potential integration issues arising from mixing embedding providers.

API Parameter Control

For a factual Question-Answering system based on documents, precision is prioritized over creativity. Therefore, for the ChatGoogleGenerativeAI initialization, setting the temperature parameter to 0 is crucial.[7] This low temperature setting minimizes stochasticity in the

model's output, forcing it to adhere strictly to the retrieved factual context, thereby reducing the probability of factual hallucination—a key weakness RAG systems must actively mitigate. Furthermore, the framework allows for configuring advanced controls, such as overriding default safety settings if specific technical content within the PDFs is being inadvertently blocked.[15] Demonstrating controlled parameter usage signals a professional shift from generic LLM experimentation to precise, domain-specific engineering.

B. Vector Store Selection: FAISS for Speed

The choice of vector store is optimized for the speed required in a live competition demo.

- **FAISS Rationale:** FAISS (Facebook AI Similarity Search) is chosen due to its extremely fast, highly optimized indexing and in-memory similarity search capabilities.[5] Since the primary performance objective during the demonstration is rapid retrieval after the initial setup, FAISS provides the necessary low-latency performance when searching through the vectors generated from the PDF documents.[17] The faiss-cpu package is sufficient for local development and competitive demonstration purposes.[5]
- **Comparative Analysis:** While persistent options like ChromaDB [18] offer on-disk storage for multi-session data persistence, the requirement for peak in-session performance and rapid indexing upon PDF upload makes FAISS the superior choice for this specific competition environment.[16]

Performance Bottleneck Mitigation through Caching

The indexing phase—loading the PDF, chunking the text, and generating thousands of embeddings—is the primary factor contributing to initial application latency. A poorly handled indexing function would cause the Streamlit application to freeze or restart the index creation on every user interaction, leading to a disastrous user experience.

The effective solution involves integrating the RAG pipeline with Streamlit's resource management tools. By wrapping the document processing and vector store creation function with the Streamlit decorator @st.cache_resource, the costly index generation process is executed only once per session.[19] The resulting FAISS index object and the embedding model instance are securely stored in a persistent resource cache, allowing instantaneous access during subsequent user queries. This integration of Streamlit's caching mechanisms with the LangChain RAG pipeline effectively solves the core latency challenge inherent to running intensive AI workloads within reactive web frameworks, ensuring a smooth, production-like

conversational flow.

Table II. Tech Stack Selection Rationale

Component Category	Selected Tool/API	Reason for Selection	Competitive Advantage
Orchestration	LangChain (Python)	Provides standardized, modular RAG component abstraction.	Enables rapid component swapping and RAG 2.0 design.[2]
LLM	ChatGoogleGenerativeAI (Gemini 2.5 Flash)	Superior speed and robust factual reasoning capabilities for low-latency chat.	Optimal performance for conversational retrieval.[7]
Embeddings	GoogleGenerativeAIEmbeddings	Optimized embedding performance and ecosystem consistency.	Maximizes vector quality and throughput.[4]
Vector Store	FAISS (faiss-cpu)	Fastest in-memory indexing and similarity search for demonstration performance.	Ensures rapid indexing and retrieval after initial load.[5]
Memory/UI	Streamlit & StreamlitChatMessageHistory	Native UI components and seamless session state persistence.	Robust, context-aware chat history management.[20]

C. User Interface and State Management

Streamlit is selected as the front-end framework due to its capacity for rapid prototyping and its native support for machine learning workflows, including built-in chat elements and file uploaders.[1] Critically, Streamlit's tight integration with LangChain's memory system via StreamlitChatMessageHistory [21] allows the RAG agent to easily access and manage conversation context stored in the application's session state, which is essential for the conversational nature of the challenge.

III. The Retrieval-Augmented Generation (RAG) Indexing Pipeline

The indexing pipeline is responsible for transforming raw PDF content into a searchable vector representation. This process is centralized in the rag_core.py utility file for architectural separation.

A. Document Loading and Pre-processing

Input Handling and Document Loading

The Streamlit application handles the user's uploaded PDF file buffer. This buffer must be temporarily written to the local disk to be processed by the PyPDFLoader.[17] The loader reads the PDF, converting its content into a list of LangChain Document objects.[14] The core benefit of using PyPDFLoader is its automatic preservation of essential document metadata, which typically includes the source filename and the page number for the content extracted from that page. This metadata is the non-negotiable foundation for the competitive citation feature.[9]

Strategic Chunking for Retrieval Quality

Large documents must be divided into smaller, semantically coherent segments, or "chunks," to ensure that the embedding model captures a single, focused idea per vector.[22]

- **Splitter Mechanism:** The blueprint mandates the use of the RecursiveCharacterTextSplitter. Unlike simple fixed-size splitters, this recursive approach attempts to maintain structural integrity by iteratively splitting the document based on a list of separators (e.g., \n\n, . , etc.).[3] This methodology preserves semantic boundaries better, resulting in higher-quality chunks that improve downstream retrieval accuracy.
- **Optimal Parameters:** The configuration is set with a chunk_size of 1000 characters and a chunk_overlap of 100 characters.[17] A chunk size of 1000 provides sufficient local context for the embedding model to generate a semantically rich vector, while the overlap of 100 characters ensures continuity between adjacent chunks, preventing semantic gaps that could otherwise occur during the split.

B. Vectorization and Resilience (Advanced Indexing Implementation)

The indexed documents must be vectorized using the high-performance Gemini embedding model before storage in the FAISS index.

Robustness Requirement: Handling API Rate Limits

High-throughput embedding operations, especially when processing large, multi-page PDF documents, are prone to triggering API rate limits defined by the model provider.[11] A sequential, unmanaged call to the embedding API for thousands of chunks will frequently result in failure due to reaching the maximum calls per minute.

To ensure stability and resilience, the embedding function must implement defense layers:

1. **Retry Logic (Exponential Backoff):** The GoogleGenerativeAIEmbeddings component must be initialized with built-in retry mechanisms, often utilizing exponential backoff logic.[11] This means that the model component is configured to automatically retry failed requests (due to transient network errors or rate limit warnings) with an exponentially increasing wait time between attempts. In LangChain, this can be achieved using the .with_retry(stop_after_attempt=X) method.[11]
2. **Concurrency Limiting:** Even with retry logic, persistently high-volume traffic must be managed proactively. If batch embedding or parallel processing is utilized, explicitly

setting a max_concurrency limit ensures that only a controlled number of requests are sent to the API provider concurrently.[12] By limiting the number of concurrent calls, the system decreases the frequency of requests, thus avoiding hard API limits and stabilizing the ingestion process, regardless of the size of the uploaded PDF.[12]

FAISS Indexing and Storage

Once the chunking and robust vector generation are complete, the resulting vectors and their corresponding chunked documents are committed to the FAISS index.[5] The process utilizes the FAISS.from_documents(documents, embeddings) method [17], which efficiently maps the embedded vectors to their textual content and preserved metadata, preparing the index for rapid similarity searching at query time. This entire expensive operation is then wrapped in Streamlit's @st.cache_resource decorator, ensuring it is executed only once per Streamlit session.

IV. Core Implementation: Building the Conversational Agent Chain

The runtime logic is built around maintaining context and retrieving relevant information efficiently.

A. The Conversational Retrieval Chain (ConversationalRetrievalChain)

A basic RAG implementation fails when users ask follow-up questions that depend on previous turns of the dialogue (e.g., "Tell me about the energy use in that section."). The document retriever, which performs a semantic search based *only* on the new query, would miss the contextual entity ("that section").

The ConversationalRetrievalChain specifically solves this problem by seamlessly integrating memory into the retrieval process.[6] The chain executes a mandatory two-step process per query, hidden from the user:

1. **Question Rephrasing:** An internal prompt, utilizing the history buffer (supplied via

StreamlitChatMessageHistory), instructs the Gemini LLM to analyze the new, ambiguous user question and the prior conversation context. The LLM then generates a contextually complete, "standalone question" (e.g., "What is the energy usage in Section 3, titled 'Advanced Diagnostics'?").

2. **Retrieval and Synthesis:** This standalone question is passed to the FAISS Retriever, which fetches the top k relevant documents (search_kwargs={"k": 5}).[17] These retrieved documents are then bundled with the original user question and history, passed back to the Gemini LLM, which generates the final, contextually appropriate, and factual response.[6]

B. Integrating Streamlit Chat History

Effective memory management is paramount for the conversational agent. The Streamlit framework simplifies this by providing the StreamlitChatMessageHistory utility.[21]

- **History Object Initialization:** The memory component is initialized in the main app.py script as StreamlitChatMessageHistory(key="chat_messages").[21] This object leverages st.session_state to automatically store all previous user and AI messages in the session.
- **Prompt Structuring for Context:** When constructing the RAG chain, a ChatPromptTemplate must be defined. This template must include a MessagesPlaceholder placeholder.[21] This component correctly injects the full list of historical BaseMessage objects from the StreamlitChatMessageHistory into the prompt, ensuring the Gemini LLM has access to the complete dialogue required for the question rephrasing step.[23]

C. LLM Configuration and Security Hardening

Custom Prompt Engineering

The system prompt (a set of instructions given to the LLM) is engineered with two primary constraints to maximize RAG quality:

1. **Factuality Enforcement:** The prompt must explicitly instruct the Gemini LLM to answer the user query *only* based on the context provided by the retrieved documents. If the

context does not contain the answer, the LLM must be instructed to state that the information is unavailable, preventing ungrounded speculation or hallucination.

2. **Source Auditing Requirement:** The prompt must implicitly or explicitly acknowledge the retrieval mechanism to ensure the LLM outputs a response that is clearly derived from the provided sources.

Output Auditability: Enabling Citations

For the citation feature to function, the RAG chain structure must be modified to return more than just the final answer text. The chain needs to return the complete response object, which includes the list of source documents that were retrieved and used for synthesis. This mandates a specific configuration (e.g., using a custom Runnable or setting return_source_documents=True depending on the LangChain version and chain type).[24] By ensuring the RAG core outputs both the generated text and the list of LangChain Document objects with metadata, the subsequent UI layer receives the necessary data (source and page) to construct the verifiable citations.[9]

V. Full Code Implementation: Modular Python Structure

A modular file structure is critical for maintainability, clarity, and competitive rigor.[19] The application logic is strictly separated into three core files to keep the main Streamlit application clean and focused solely on the user interface and workflow management.

A. Project File Manifest and Structure

Table III details the recommended modular structure designed for professional AI application deployment.

Table III. Recommended Project Folder Structure and Purpose

Folder/File	Purpose	Dependencies/Key Components	References
/	Project Root Directory	-	-
app.py	Main Streamlit UI, state management, chat workflow orchestration, and presentation layer.	Streamlit, rag_core.py, utils.py	[1]
rag_core.py	Core RAG pipeline functions: document loading, chunking, vector storage (FAISS), and conversational chain initialization.	LangChain, Gemini API, FAISS, PyPDF	[4]
utils.py	Helper functions for environment setup, secure API key handling, and common utilities.	python-dotenv, os, st.secrets	[26]
requirements.txt	Defines all necessary Python dependencies (e.g., langchain-google-genai, faiss-cpu, streamlit, pypdf) for reproducibility.	pip	[28]
.env	Local configuration for GEMINI_API_KEY (must be excluded from Git).	-	[27]

data/	Directory placeholder for temporarily uploaded PDF documents.	-	-

B. Code Requirements for Core Components

1. utils.py: Environment Setup and Security

This file ensures the secure and robust configuration of the environment, particularly the critical API key.

Python

```
# utils.py
import os
# from dotenv import load_dotenv # Use if running locally with a.env file

def setup_environment(api_key_source):
    """Securely configure the GEMINI_API_KEY from secrets or.env."""
    # In a production Streamlit app, st.secrets is preferred.
    # Locally, we check os.environ or a passed value.

    # Check if the key is already set (e.g., via st.secrets on Hugging Face)
    if not os.getenv("GEMINI_API_KEY"):
        os.environ = api_key_source # Load the key

    if not os.getenv("GEMINI_API_KEY"):
        raise ValueError("GEMINI_API_KEY not found. Please set it via environment variables or Streamlit secrets.")
    return os.getenv("GEMINI_API_KEY")
```

```python
# Placeholder for citation formatting function (used by app.py)
def format_citations(source_documents):
    """Extracts and formats source file and page numbers for display."""
    citations = set()
    for doc in source_documents:
        metadata = doc.metadata
        # PyPDFLoader usually stores 'source' (filename) and 'page' (0-indexed)
        if 'source' in metadata and 'page' in metadata:
            # Add 1 to page number for human readability
            citations.add(f"File: {os.path.basename(metadata['source'])}, Page: {metadata['page'] + 1}")

    if citations:
        citation_text = "\n\n**Sources:**\n" + "\n".join([f"- {c}" for c in sorted(list(citations))])
        return citation_text
    return ""
```

2. rag_core.py: The RAG Pipeline Implementation

This file houses the heavy lifting, including the cached indexing function and the chain setup.

Python

```python
# rag_core.py
import os
from langchain_community.document_loaders import PyPDFLoader
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_Genai import GoogleGenerativeAIEmbeddings, ChatGoogleGenerativeAI
from langchain_community.vectorstores import FAISS
from langchain.chains import ConversationalRetrievalChain
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_community.chat_message_histories import StreamlitChatMessageHistory
import streamlit as st

# Function 1: Document Processing and Indexing (Cached)
@st.cache_resource
def get_vector_store(pdf_file_path):
    # Load document
```

```
    loader = PyPDFLoader(pdf_file_path)
    documents = loader.load() # Load all documents from PDF

    # Split documents into chunks
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,
        chunk_overlap=100,
        separators=["\n\n", "\n", ".", " ", ""]
    )
    docs = text_splitter.split_documents(documents)

    # Initialize Embeddings with Rate-Limit Handling
    # The ChatGoogleGenerativeAI object handles retries implicitly via max_retries
    embeddings = GoogleGenerativeAIEmbeddings(
        model="models/gemini-embedding-001",
        # Explicit concurrency management may be added if using a parallel map function
    )

    # Create FAISS Vector Store [5, 17]
    vector_store = FAISS.from_documents(docs, embeddings)

    return vector_store

# Function 2: Conversational Chain Initialization
def get_conversation_chain(vector_store, history_key="chat_messages"):
    # 1. Initialize LLM (Gemini 2.5 Flash for speed)
    llm = ChatGoogleGenerativeAI(
        model="gemini-2.5-flash",
        temperature=0,  # Factual precision required
        max_retries=2 # Built-in retry mechanism
    )

    # 2. Setup Retriever
    retriever = vector_store.as_retriever(
        search_type="similarity",
        search_kwargs={"k": 5} # Retrieve top 5 documents
    )

    # 3. Setup Memory and History
    memory = StreamlitChatMessageHistory(key=history_key)

    # 4. Create the Conversational Retrieval Chain
    # This chain automatically handles chat history to form a "standalone question."
```

```
    conversation_chain = ConversationalRetrievalChain.from_llm(
        llm=llm,
        retriever=retriever,
        memory=memory,
        # IMPORTANT: Set to True to allow extraction of source metadata for citations
        return_source_documents=True
    )

    return conversation_chain
```

VI. Frontend Engineering: Streamlit UI and State Management

The Streamlit interface provides the necessary structure for file upload, state persistence, and a modern chat environment.

A. Interactive UI Design and Flow Control

The application uses Streamlit's native components for an intuitive user experience.[20]

1. **Sidebar Configuration:** The sidebar is the control center. It must contain:
    - **API Key Input:** A masked text input field allows the user to securely enter the GEMINI_API_KEY locally. On deployment platforms like Hugging Face Spaces, it is mandated that the key be managed via st.secrets instead.[26]
    - **PDF File Uploader:** The st.file_uploader component handles the user input, accepting files of type pdf.[30]
    - **Processing Trigger:** A dedicated st.button labelled "Process Documents" initiates the computationally intensive RAG indexing pipeline, separating it from the chat runtime.
2. **Main Chat Interface:** The central area uses st.chat_message and st.chat_input to simulate a traditional messaging application.[31] Status messages are integrated into the main flow, providing feedback when the indexing process is running (e.g., "Indexing 1500 chunks, please wait...") or when a process completes, improving transparency and managing user expectations.

B. State Persistence and Conditional Logic

The proper use of st.session_state is vital for maintaining context and controlling the application flow.[19]

- **Initialization:** The application must initialize session state variables at startup. Key flags include st.session_state.pdf_processed = False and st.session_state.conversation_chain = None.
- **Flow Control:** The core chat input field (st.chat_input) is conditional. It is only displayed and made active once the indexing process is successfully completed and st.session_state.pdf_processed is set to True. This prevents the system from receiving queries before a retriever is available.
- **History Persistence:** The conversation history is automatically managed by the StreamlitChatMessageHistory object, which is tied to st.session_state["chat_messages"].[21] This ensures that when the script re-runs due to user interaction, the history is instantly available for the next turn of the ConversationalRetrievalChain.

C. Implementing the Citation Feature (Competitive UX)

The citation feature is a significant competitive differentiator that demonstrates auditability. This requires coordination between the rag_core.py (which must return source documents) and app.py (which processes the output).

1. **Data Extraction:** The result object returned by the ConversationalRetrievalChain is parsed to extract two components: the generated text answer and the list of source_documents.[24] Each document object within this list contains the necessary metadata (file name and page number, which are preserved during the loading and chunking stages).[9]
2. **Formatting and Display:** Simply listing page numbers is insufficient for professional presentation. The Streamlit component st.markdown is utilized to format the extracted citations cleanly.[10] The metadata, consisting of the original source file and the (page + 1) for human-readable page numbering, is compiled into a dedicated, numbered list or footnote section appended immediately after the AI's response.[9]

- **Advanced Traceability:** Achieving the highest level of traceability involves making the citation clickable, allowing the user to view the precise source page.[10] While complex for

local Streamlit deployment (often requiring a separate file server), the architecture supports this: the citation could be formatted as an anchor link pointing to a locally served PDF with a page fragment identifier (e.g., document.pdf#page=N) using st.components.v1.iframe.[10]

VII. Competitive Add-on Features and Enhancements

These optional features significantly increase the project's complexity score and utility, moving it beyond a standard RAG template.

A. Multi-Document Summarization Utility

Adding a summarization feature demonstrates the system's versatility beyond simple QA.

- **Functionality:** A dedicated button in the Streamlit sidebar triggers a large-scale analysis of all uploaded documents.
- **Architectural Detail:** This feature requires a separate LangChain chain, such as the load_summarize_chain, specifically configured with the Map-Reduce strategy for handling large files.[32] The Gemini LLM is tasked with generating smaller summaries for each chunk (Map step) and then synthesizing these smaller summaries into a single comprehensive summary (Reduce step). This demonstrates effective chunk-by-chunk processing for documents that exceed the LLM's context window.

B. Export Chat History Functionality

Providing an audit trail is a hallmark of enterprise applications.[28]

- **CSV Export (Standard Utility):** The robust default implementation involves creating a helper function that retrieves the entire list of messages from st.session_state.chat_messages. This list is then converted into a Pandas DataFrame containing columns for the user, assistant, and timestamp, and exposed to the user via Streamlit's st.download_button as a CSV file.[28]
- **PDF Export (High-Value Feature):** Exporting the chat history in a professionally formatted PDF is a strong competitive add-on. This requires utilizing external Python

libraries like pdfkit to convert the rendered Streamlit HTML output of the chat history into a PDF document.[34] This approach requires careful handling of temporary files and external prerequisites (such as wkhtmltopdf needing to be installed in the deployment environment) but offers a superior final output document.[35]

C. Advanced Telemetry and Auditing

Demonstrating awareness of operational metrics like cost and performance is crucial for expert-level evaluation.

- **Token Usage Tracking:** The ChatGoogleGenerativeAI integration in LangChain provides result.usage_metadata in its response object.[7] This metadata includes precise counts for input_tokens, output_tokens, and total_tokens consumed by the LLM call. The application should parse and display these metrics for each response, either in the sidebar or directly below the generated answer. This visible auditing demonstrates cost awareness and MLOps preparedness.
- **Debug/Audit Mode:** A configurable toggle switch in the sidebar can activate a debug mode. When active, the application displays internal RAG pipeline steps that are normally hidden from the end user. This includes showing the raw, exact chunks of text retrieved from FAISS and, crucially, displaying the LLM-generated "standalone question" created by the ConversationalRetrievalChain.[6] This transparency is invaluable for judges and technical reviewers, allowing them to audit the RAG process and immediately diagnose poor retrieval performance (a common failure mode).

VIII. Deployment and Project Report Write-Up

A. Local Run Instructions

To ensure reproducibility, a standard, clear operational guide must be provided.

1. **Virtual Environment Setup:** Create and activate a dedicated Python virtual environment to isolate project dependencies.
2. **Dependency Installation:** Install all necessary packages defined in requirements.txt

using the command: pip install -r requirements.txt.[28]

3. **API Key Configuration:** Obtain a GEMINI_API_KEY and set it securely as an environment variable, ideally within a local .env file if not using st.secrets.[27]
4. **Application Launch:** Execute the Streamlit application using the command: streamlit run app.py.

B. Cloud Deployment on Hugging Face Spaces

Hugging Face Spaces is an optimal platform for hosting competition submissions, offering free hosting and automatic environment management.[36]

1. **Platform Selection and Structure:** Choose the Streamlit SDK when creating a new Hugging Face Space.[36] All project files (app.py, rag_core.py, utils.py, requirements.txt) must be uploaded into the repository, ensuring the requirements.txt is present for automatic dependency installation.[38]
2. **Secure Secrets Management (Non-Negotiable):** The local .env file must never be committed. For secure cloud deployment, the GEMINI_API_KEY must be configured using the Hugging Face Spaces Secrets interface.[39] This secret is then automatically accessible within the Streamlit application via st.secrets.[26] This methodology demonstrates strict security compliance and prevents API key exposure.

C. Official Project Report Narrative

The final submission requires a professional narrative that synthesizes the project's technical achievements and competitive advantages.

1. **Introduction and Problem Statement:** Contextualize the increasing need for reliable knowledge retrieval from proprietary documents in enterprise settings. Introduce the "EONVERSE Gemini RAG Agent" as a state-of-the-art solution designed to overcome the limitations of traditional keyword search and context-free LLM applications.
2. **Architectural Design Review:** Present a detailed discussion of the RAG 2.0 pipeline, emphasizing the decoupling of the indexing and retrieval phases. Detail the role of GoogleGenerativeAIEmbeddings for vectorization, the use of FAISS for high-speed similarity search, and the critical function of the ConversationalRetrievalChain in managing historical context.[4]
3. **Technical Justification and Innovation:** Justify the configuration choices, such as setting the LLM temperature to 0 for maximizing factual adherence. Highlight the

inclusion of robust rate-limit handling mechanisms (retry logic and concurrency limits) during the high-volume embedding phase, demonstrating architectural stability.[11] Explicitly emphasize the innovative citation mechanism as a demonstration of superior output auditability and traceability.[9]

4. **User Experience and Auditing:** Detail the use of Streamlit for rapid UI development and the implementation of session state for seamless conversation memory. Discuss how the traceable citations enhance user trust and satisfy auditing requirements by grounding every response in verifiable source material.[10]
5. **Conclusion and Future Work:** Summarize the project's success in achieving speed, accuracy, and conversational depth. Outline future enhancements, such as migrating FAISS to a persistent cloud vector database (e.g., Pinecone for scale [16]), or transitioning the core orchestration from ConversationalRetrievalChain to a more flexible agentic architecture using LangGraph for complex tool use and branching logic.[40]

IX. Conclusions and Recommendations

The EONVERSE Gemini RAG Agent blueprint establishes a robust, high-performance conversational PDF Q&A system. The architectural decisions, particularly the reliance on FAISS for speed [16], the integration of the Gemini ecosystem for unified LLM/Embedding services [4], and the strategic use of Streamlit resource caching, ensure rapid indexing and low latency during demonstration.

The key competitive advantage lies in the functional implementation of the ConversationalRetrievalChain paired with verifiable source citations.[6] These elements move the project beyond a basic technical demonstration to a mature, trustworthy application capable of sustaining complex, multi-turn dialogues over unstructured data.

For future expansion, it is recommended to implement the advanced add-on features, specifically the chat history PDF export, as this completes the professional application lifecycle, providing a full audit trail for all interactions within the system.[34] Furthermore, exploring the use of structured output methods like JSON mode with Gemini models should be considered to enhance the reliability of extracting complex metadata, such as citations or structured summaries, ensuring better reliability than relying solely on tool calling constraints.[7]

Works cited

1. Langchain PDF App (GUI) | Create a ChatGPT For Your PDF in Python - YouTube, accessed on November 13, 2025, https://www.youtube.com/watch?v=wUAUdEw5oxM

2. Build a RAG agent with LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langchain/rag
3. RecursiveCharacterTextSplitter — LangChain documentation, accessed on November 13, 2025, https://api.python.langchain.com/en/latest/text_splitters/character/langchain_text_splitters.character.RecursiveCharacterTextSplitter.html
4. Google Generative AI Embeddings (AI Studio & Gemini API) - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/integrations/text_embedding/Generative_ai
5. Faiss - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/integrations/vectorstores/faiss
6. ConversationalRetrievalChain — LangChain documentation, accessed on November 13, 2025, https://api.python.langchain.com/en/latest/langchain/chains/langchain.chains.conversational_retrieval.base.ConversationalRetrievalChain.html
7. ChatGoogleGenerativeAI - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/integrations/chat/Generative_ai
8. Retrieval - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/javascript/langchain/retrieval
9. Create Citations in RAG Streamlit App : r/LangChain - Reddit, accessed on November 13, 2025, https://www.reddit.com/r/LangChain/comments/1dkjxos/create_citations_in_rag_streamlit_app/
10. Streamlit: How to add proper citation with source content to chat message - Stack Overflow, accessed on November 13, 2025, https://stackoverflow.com/questions/77455300/streamlit-how-to-add-proper-citation-with-source-content-to-chat-message
11. How to handle model rate limits - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/langsmith/rate-limiting
12. Dealing with rate limits | 🦜 Langchain, accessed on November 13, 2025, https://js.langchain.com/v0.1/docs/modules/data_connection/text_embedding/rate_limits/
13. LangChain overview - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langchain/overview
14. Build a semantic search engine with LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langchain/knowledge-base
15. ChatGoogleGenerativeAI - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/javascript/integrations/chat/Generative_ai
16. Comparing Vector Stores in LangChain: Chroma vs FAISS vs Pinecone. - Medium, accessed on November 13, 2025, https://medium.com/@warishayat/comparing-vector-stores-in-langchain-chroma-vs-faiss-vs-pinecone-c7aa1e46386c
17. Langchain , OpenAI and FAISS — Implementation | by LEARNMYCOURSE - Medium, accessed on November 13, 2025,

https://learnmycourse.medium.com/langchain-openai-and-faiss-implementation-90c541a90da8
18. Vector stores - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/integrations/vectorstores
19. Best Practices for Building GenAI Apps with Streamlit, accessed on November 13, 2025, https://blog.streamlit.io/best-practices-for-building-genai-apps-with-streamlit/
20. App Gallery - Streamlit, accessed on November 13, 2025, https://streamlit.io/gallery?category=llms
21. Streamlit | 🦜 LangChain, accessed on November 13, 2025, https://python.langchain.com/v0.1/docs/integrations/memory/streamlit_chat_message_history/
22. Design and Develop a RAG Solution - Azure Architecture Center | Microsoft Learn, accessed on November 13, 2025, https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/rag/rag-solution-design-and-evaluation-guide
23. LangChain ConversationBufferMemory: Complete Implementation Guide + Code Examples 2025 - Latenode, accessed on November 13, 2025, https://latenode.com/blog/langchain-conversationbuffer-memory-complete-implementation-guide-code-examples-2025
24. How to return citations - LangChain overview, accessed on November 13, 2025, https://js.langchain.com/v0.2/docs/how_to/qa_citations/
25. How to Structure and Organise a Streamlit App - Towards Data Science, accessed on November 13, 2025, https://towardsdatascience.com/how-to-structure-and-organise-a-streamlit-app-e66b65ece369/
26. Secrets management - Streamlit Docs, accessed on November 13, 2025, https://docs.streamlit.io/develop/concepts/connections/secrets-management
27. Building a Gemini Powered Chatbot in Streamlit | by Harisudhan.S - Medium, accessed on November 13, 2025, https://medium.com/@speaktoharisudhan/building-a-gemini-powered-chatbot-in-streamlit-e241ed5958c4
28. huseyincenik/streamlit_langchain: Langchain Project via Streamlit by using Gemini and OpenAI - GitHub, accessed on November 13, 2025, https://github.com/huseyincenik/streamlit_langchain
29. Create Your Own AI RAG Chatbot: A Python Guide with LangChain - DEV Community, accessed on November 13, 2025, https://dev.to/shreshthgoyal/create-your-own-ai-rag-chatbot-a-python-guide-with-langchain-dfi
30. Building a Chat Application with PDF Using Streamlit and LLM | by Syahmisajid - Medium, accessed on November 13, 2025, https://medium.com/@syahmisajid12/building-a-chat-application-with-pdf-using-streamlit-and-llm-db7049d24133
31. EXPESRaza/pdf-qa-app-with-langchain: A modular PDF Q&A app built with Streamlit, LangChain, and FAISS. Supports natural queries, exact term counting,

streaming output, and LLM selection (OpenAI, Ollama). Developed by prompting ChatGPT to generate a Cursor.ai scaffold for fast, extensible - GitHub, accessed on November 13, 2025, https://github.com/EXPESRaza/pdf-qa-app-with-langchain
32. AI Summarizer App LangChain, Python, Streamlit, OpenAI and Groq (PDF, CSV and Text Files) - YouTube, accessed on November 13, 2025, https://www.youtube.com/watch?v=0DDwvq8DxH8
33. mirabdullahyaser/Retrieval-Augmented-Generation-Engine-with-LangChain-and-Streamlit - GitHub, accessed on November 13, 2025, https://github.com/mirabdullahyaser/Retrieval-Augmented-Generation-Engine-with-LangChain-and-Streamlit
34. Streamlit to PDF: how to build & distribute PDF reports | by Niko Nelissen | Peliqan.io, accessed on November 13, 2025, https://medium.com/peliqan-io/streamlit-to-pdf-f6f4a68fed3b
35. Download PDF option - Using Streamlit, accessed on November 13, 2025, https://discuss.streamlit.io/t/download-pdf-option/19386
36. Streamlit Spaces - Hugging Face, accessed on November 13, 2025, https://huggingface.co/docs/hub/en/spaces-sdks-streamlit
37. Streamlit Spaces - Hugging Face, accessed on November 13, 2025, https://huggingface.co/docs/hub/spaces-sdks-streamlit
38. Deploy Streamlit App to Hugging Face Spaces - A Complete Guide - Shafiqul AI, accessed on November 13, 2025, https://shafiqulai.github.io/blogs/blog_4.html
39. How to Deploy AI Apps with Streamlit, Docker, and Hugging Face - YouTube, accessed on November 13, 2025, https://www.youtube.com/watch?v=Sx_MwcBQGOg
40. ReAct agent from scratch with Gemini 2.5 and LangGraph - Google AI for Developers, accessed on November 13, 2025, https://ai.google.dev/gemini-api/docs/langgraph-example
41. This NVIDIA RAG blueprint serves as a reference solution for a foundational Retrieval Augmented Generation (RAG) pipeline. - GitHub, accessed on November 13, 2025, https://github.com/NVIDIA-AI-Blueprints/rag