Expert Blueprint for a LangChain AI PDF Q&A Chatbot

The production of an AI PDF Q&A Chatbot utilizing LangChain represents a sophisticated application of Retrieval-Augmented Generation (RAG). This comprehensive blueprint details the architectural design, workflow implementation, tool selection rationale, and best practices necessary to develop a robust system suitable for the EONVERSE AI Intern Screening Challenge, satisfying the requirements for logical flow, quality implementation, and independent exploration.[1]

I. Project Foundation and Architectural Overview

1.1 Problem Statement and Project Objective

Large Language Models (LLMs) are formidable tools but are fundamentally constrained by **static knowledge**, meaning their training data is frozen at a specific point in time, and **finite context windows**, limiting the volume of proprietary information they can process simultaneously.[2] When these models are tasked with answering domain-specific questions about internal documentation or user-uploaded PDFs, they often resort to guessing or generating plausible but inaccurate statements (hallucination). This deficiency makes standard LLMs unreliable for enterprise-grade, grounded Q&A systems.

The core **Project Objective** is to implement a trustworthy RAG pipeline that overcomes these limitations. The RAG architecture enables the LLM to access up-to-date, domain-specific information dynamically by querying a vector database built from unstructured data.[3] The outcome will be an application that can reliably answer questions about specific PDF contents while citing its sources, thereby fulfilling the EONVERSE challenge requirement to "Create a LangChain agent that answers questions from PDF docs".[1] The success of this implementation relies heavily on establishing a scalable, dependable architecture that ensures the LLM

receives the necessary context to appropriately reply to user queries.[3]

1.2 The Two Phases of the RAG Lifecycle

A production-quality RAG system is structurally composed of two independent pipelines that must be optimized separately [3]: the Indexing Pipeline and the Retrieval and Generation Pipeline.

Indexing Pipeline (Offline Preparation)

This phase is executed offline and focuses on data preparation and index creation. It involves ingesting documents, extracting text, transforming the data through cleaning and chunking, converting the resulting segments into high-dimensional vector embeddings, and finally persisting these embeddings in a Vector Database (VDB).[3] The primary goal here is non-negotiable: to build a **trustworthy, accurate vector search index**.[3] If the underlying index is poor, the LLM will inevitably be fed incorrect context, leading to inaccurate responses regardless of the model's quality.[5] Therefore, substantial effort must be concentrated on the quality of PDF parsing and chunking in this initial phase.

Retrieval and Generation Pipeline (Runtime Execution)

This phase executes every time a user submits a query. It involves dynamically fetching relevant external knowledge at query time.[2] The steps include receiving the query, using an intermediary model to optimize the search query (especially for conversational history), searching the VDB to retrieve the most relevant context, packaging that context along with the original user query into a final prompt template, and passing this to the LLM to synthesize the final, grounded answer.[4]

1.3 Recommended Architecture Blueprint (LangChain-Centric)

The architecture leverages LangChain as the central orchestration framework, providing the necessary modules for data loading, splitting, vectorization, and chain execution (LCEL).[7] The system is deployed via a Streamlit interface.[9]

Architecture Components and Flow

1. **User Interface (Streamlit):** Handles PDF upload and manages the conversational flow using session state.[10]
2. **Indexing Pipeline (Preparation):**
    - **Document Loader:** Processes the uploaded PDF, ideally using a specialized parser like UnstructuredLoader for robust text and layout extraction.[7]
    - **Text Splitter:** Breaks the text into semantically relevant chunks (e.g., using RecursiveCharacterTextSplitter or the more advanced ParentDocumentRetriever components).[11]
    - **Embedding Model:** Converts these chunks into dense vector representations.[5]
    - **Vector Store (Chroma):** Stores the vector index persistently, allowing the Streamlit application to reload the knowledge base quickly without re-embedding the entire document on every session restart.[3]
3. **Retrieval Pipeline (Runtime):**
    - **Query Transformation:** If conversation history is present, a specialized sub-chain reformulates the user's implicit query into an explicit, standalone query optimized for semantic search.[6]
    - **Retriever:** Queries the Vector Store with the standalone query, fetching the top K relevant chunks of context.
    - **LLM Chain (LCEL):** LangChain Expression Language (LCEL) constructs a unified sequence that injects the retrieved context and the conversation history into a structured prompt, which is then passed to the LLM.[8]
    - **LLM (OpenAI/HuggingFace):** Generates the final, grounded response, adhering to strict instructions regarding source citation.[14]

II. Tool Stack and Core Component Selection Strategy

Choosing the right components ensures technical quality and efficient resource utilization, addressing both the "Technical Skill" and "Creativity" criteria of the challenge.[1]

2.1 PDF Parsing and Document Loading: Prioritizing Robust Extraction

The initial extraction of text from a PDF is the most common point of failure for RAG systems. While LangChain offers standard loaders like PyPDFLoader [15], these often struggle with complex documents containing tables, figures, headers, and footers.

The recommended approach is to use the **Unstructured Loader** (Unstructured.io). Unstructured is exceptionally strong at extracting structured information and preserving document layout semantics, providing a higher quality Document object for downstream processes.[7] Alternatively, PyMuPDF (via fitz) offers strong low-level control for extracting text and metadata like positions, which may be necessary if precise table data discernment is required.[7] For a robust demo handling diverse documents, UnstructuredLoader is preferred for its ability to generate "LLM-ready data".[7]

2.2 Embedding Model Selection: The Engine of Relevance

The quality of the embedding model directly correlates with the relevance of the retrieved context. A weak embedding model translates the user's query and the document chunks poorly, meaning the search fails to identify the correct document sections.[5]

The selection can be either commercial or open-source:

- **Commercial (Baseline):** OpenAI's text-embedding-3-small provides high accuracy and ease of integration.
- **Open-Source (Recommended):** Open-source models like BGE (Baidu Gan Embedding), E5, or Nomic models should be used to demonstrate proficiency outside of proprietary APIs.[5] These offer excellent performance while allowing for transparency, debugging, and avoiding vendor lock-in. Benchmarks often show BGE and E5 models as top contenders for RAG-style search.[5]

2.3 Vector Store Selection: Balancing Speed and Persistence

The vector database choice affects both performance and deployability.

- **FAISS:** Suitable for rapid, in-memory testing but requires the document to be re-indexed (re-embedded) every time the application starts.[8] This introduces unacceptable startup latency for a deployed Streamlit application.
- **Chroma:** Recommended choice. Chroma supports persistence, allowing the vector index to be saved to disk (persist_directory) and loaded quickly. This is essential for a publicly deployed demo, as it greatly improves initialization speed and maintains a consistent knowledge base across sessions.[3]

2.4 LLM Selection Strategy: Optimizing for Synthesis

Since the RAG architecture explicitly provides the external context, the LLM's primary function shifts from knowledge retrieval to sophisticated text synthesis. Therefore, extremely large models are often unnecessary.

- **OpenAI:** gpt-3.5-turbo offers high reliability in adhering to system prompts and delivering quality output rapidly.
- **HuggingFace Models (Recommended for Challenge):** Using a small, instruct-tuned open-source model demonstrates efficiency and control.[1] Models in the 2B-8B parameter range, such as Qwen 2.5-3B-Instruct, Llama 3 8B, or Gemma 2-2B-it, are highly suitable.[17] These models can be deployed using the HuggingFaceHub wrapper, or locally via Ollama for maximum speed and control over costs, integrating seamlessly with LangChain.[18]

III. Detailed Step-by-Step LangChain Workflow Implementation

The technical workflow should be implemented using Python and structured into the two distinct phases.

A. Phase 1: The Indexing Pipeline (Data Ingestion)

This pipeline must be executed when the PDF is first uploaded or when the application initializes if a pre-indexed store is not found.

Step 1.1: Load Documents and Extract Metadata
The process begins with importing the PDF content into LangChain's Document format. The UnstructuredPDFLoader is initialized to handle the document structure, ensuring that crucial metadata—especially page numbers, which are necessary for source citation—are extracted and attached to the document pages.15
Step 1.2: Text Splitting and Chunking
The monolithic document must be broken down into smaller segments suitable for embedding and retrieval. The RecursiveCharacterTextSplitter is the standard tool, designed to split text hierarchically using a list of separators (e.g., \n\n, \n, .,).19 A common starting configuration uses a chunk_size of 1000 characters and a chunk_overlap of 200 characters. This overlap ensures that semantic bridges are maintained between neighboring chunks, preventing critical context from being split at boundaries.19
Step 1.3: Create Embeddings and Store Index
The resulting chunks are vectorized using the selected embedding model. This data transformation converts semantic meaning into numerical form. The chunks and their associated embeddings are then stored in the persistent Vector Store (Chroma). The Chroma.from_documents() method handles both the embedding generation and storage in a single step, persisting the index to a specified directory (./index_store/chroma_db).3 Finally, the vectorstore object is converted into a Retriever, configured with a parameter K (e.g., $K=4$) to specify how many chunks should be retrieved per query.

B. Phase 2: The Retrieval and Generation Pipeline (Runtime Execution)

This pipeline runs sequentially upon every user query, orchestrated by LangChain Expression Language (LCEL).

Step 2.1: Implement History-Aware Retrieval
For a successful conversational chatbot, the system must maintain memory across turns. When a user asks a follow-up question (e.g., "What about the second one?"), the system must understand that "the second one" refers to an entity mentioned earlier. This is achieved through History-Aware Retrieval, utilizing the create_history_aware_retriever utility.6 An initial LLM call, prompted by a "rewrite question instructions" system prompt, analyzes the current user input ({input}) and the chat_history ({chat_history}) to generate a standalone, explicit search query. This standalone query is then passed to the retriever for accurate context search.6
Step 2.2: Build the Core Retrieval Chain (LCEL Orchestration)
The main RAG chain is built using LCEL components for modularity and clarity.8 The chain sequence involves:

1. **Context and Question Mapping:** A dictionary mapping ensures the output of the retriever (context) and the current question (question) are correctly passed downstream. The retrieved documents are formatted into a single string for prompt injection using

format_documents_as_string.
2. **Answer Generation:** The formatted context, combined with the question and conversation history, is injected into the final RAG Prompt Template. The create_stuff_documents_chain is used to efficiently manage the context stuffing and LLM invocation.[6]

Step 2.3: Final LLM Generation and Source Citation
The LLM processes the complete, grounded prompt and generates the response. Crucially, the final chain must be configured to return both the generated text and the retrieved source documents. The metadata associated with these documents (e.g., page number from the PDF parser) is extracted and presented to the user, grounding the response and allowing for easy verification of the facts.14

IV. Optimization and Advanced RAG Techniques (Best Practices)

To demonstrate a deep understanding of RAG engineering and technical leadership, adopting advanced techniques beyond basic chunking is necessary.

4.1 Advanced Chunking Strategy: Parent Document Retrieval

Using a fixed chunk size often forces a trade-off: small chunks maximize search precision but strip away surrounding context needed for the LLM to synthesize a comprehensive answer; conversely, large chunks provide context but dilute the embedding, reducing search accuracy.

The **Parent Document Retrieval (PDR)** strategy resolves this conflict.[20] PDR employs two sets of chunks:

1. **Child Chunks:** Small, optimized chunks stored in the vector database and used solely for *retrieval precision*.[12]
2. **Parent Documents:** Larger segments, often the full original page or a large section, which are only retrieved after a successful search and passed to the LLM for *contextual completeness*.[21]

The ParentDocumentRetriever in LangChain orchestrates this flow: the query retrieves the precise Child Chunks, and then the Retriever uses the metadata (e.g., Parent ID) of those small chunks to fetch the corresponding larger Parent Documents for the LLM.[12] This

technique significantly improves retrieval quality and is a hallmark of sophisticated RAG design.

4.2 Retrieval Strategy: Addressing Contextual Gaps

While history-aware retrieval (Section III-B) ensures multi-turn relevance, another failure mode relates to the intrinsic limitation of dense retrieval. Semantic search (vector comparison) excels at finding conceptually similar passages. However, it can fail to locate documents that contain highly specific or technical terms, such as proper names, model numbers, or regulatory codes, which often appear in technical PDFs.

This suggests that relying solely on dense (vector) retrieval is suboptimal. A robust system should consider **Hybrid Search**, which combines dense retrieval with **Sparse Retrieval** (keyword-based methods like BM25).[22] Sparse retrieval ensures that literal keyword matches are not missed, significantly boosting the system's overall recall, particularly critical when working with structured or highly technical documentation.

4.3 Prompt Engineering for Grounding and Trustworthiness

The prompt template acts as the contractual agreement with the LLM. It must be carefully engineered to enforce grounded behavior and maximize user trust.[14]

The RAG system prompt must include the following directives:

- **Strict Adherence:** Explicitly instruct the LLM: "You are a grounded assistant. Use ONLY the provided context." If the context is missing, the LLM must be mandated to state: "I do not have enough evidence from the document to answer that question." This prevents hallucination.[6]
- **Source Citation:** The prompt must instruct the LLM to cite its sources inline using the page numbers or titles provided in the retrieved document metadata (e.g., "The mechanism is detailed in the section on kinetics [Page 4]"). This is vital for accountability and trustworthiness.[14]
- **Context Structure:** Use distinct markers, such as XML tags, to delineate the provided context within the prompt, ensuring the LLM clearly separates the retrieved information from the system instructions or the conversation history.[6]

V. Project Implementation Blueprint and Deployment

5.1 Suggested Project File/Folder Structure

A clean, modular file structure is essential for development, testing, and deployment.[23]

```
/pdf-rag-chatbot/
├──.env                # Environment variables for API keys and local configurations
├── requirements.txt      # Dependencies for Streamlit Cloud deployment
├── streamlit_app.py      # Main application entry point and UI logic
├── data/
│   └── uploaded_pdfs/    # Storage for user-uploaded documents (temporary or persistent)
└── core/
    ├── __init__.py       # Python module initialization
    ├── config.py         # Centralized configuration (chunk size, model names, K value)
    ├── ingestion.py      # Phase 1 logic: Document loading, chunking, and index creation
    └── rag_chain.py      # Phase 2 logic: LCEL chain definitions, prompts, and retrieval
configuration
```

5.2 Blueprint for a Minimal Functional Streamlit Demo

The streamlit_app.py serves as the user interface and orchestration layer, requiring efficient management of state and resources.[9]

Critical Performance Requirement: Caching the Index

Building the vector index (indexing and embedding) is an expensive operation. If this runs upon every user interaction, the application will be slow and inefficient. Therefore, the implementation must use Streamlit's resource caching (@st.cache_resource) to persist the Retriever object in memory. This ensures that the PDF is loaded and embedded only once during the application's lifespan, or until the user uploads a new PDF.

Functional Workflow in streamlit_app.py:

1. **Dependency and Key Setup:** Import libraries and handle API key retrieval securely using st.secrets for deployed applications.[9]
2. **Indexing Function:** Define the @st.cache_resource function (initialize_rag_system) that encapsulates the entire Phase 1 logic from ingestion.py. This function takes the uploaded PDF path, builds the Chroma index, and returns the configured Retriever.
3. **State Management:** Use st.session_state to store and manage the conversation history (st.session_state["messages"]), ensuring contextual continuity between turns.[10]
4. **UI Loop:**
    - Display the chat history using st.chat_message components.
    - Accept user input via st.chat_input.
    - Upon receiving input, execute the cached RAG chain (loaded from rag_chain.py), feeding it the user input and the entire st.session_state["messages"] as chat_history.
    - Display the streaming response from the LLM and update the session state with the new turn and source citations.

5.3 Deployment Recommendations (Streamlit Cloud)

Streamlit Community Cloud offers the most streamlined path for deployment.[24]

1. **Preparation:** Host the complete project directory (including requirements.txt) in a public GitHub repository.[24]
2. **Dependencies:** Ensure requirements.txt lists all necessary packages, including streamlit, langchain-core, chromadb, and the specific model integration packages (langchain-openai or langchain-community).[9]
3. **Secrets:** Securely manage API keys by creating a .streamlit/secrets.toml file or using the Streamlit dashboard interface to upload secrets.[24] This prevents exposing sensitive credentials in the codebase.

4. **Deployment:** Deploy the application directly from the GitHub repository via the Streamlit Cloud workspace.[24] The application must be configured to run streamlit_app.py.

VI. Optional Improvements and Project Extensions

These enhancements demonstrate further technical depth, satisfying the "Curiosity & Exploration" criteria of the challenge.[1]

6.1 Advanced Retrieval: Hybrid Search

As discussed in Section 4.2, implementing Hybrid Search significantly improves the system's ability to locate highly specific technical content. This involves combining the dense vector retrieval with sparse keyword retrieval (e.g., using the BM25Retriever from langchain_community). These two retrieval methods can be integrated using LangChain's EnsembleRetriever, which performs both searches and fuses the results based on a weighting algorithm. This approach yields higher quality context, ensuring both semantic relevance and keyword accuracy are captured.[22]

6.2 Implementation of Agentic RAG

Moving beyond a fixed, pre-defined chain structure and implementing an Agentic RAG system elevates the project's technical complexity. An Agent utilizes a capable LLM (the brain) to orchestrate a decision-making process.[26] The LLM is provided with various tools—in this case, the Retriever—and it decides dynamically, based on the user's query, whether to use the retrieval tool to find external context or to answer directly from its internal knowledge.[26] This implementation requires familiarity with frameworks like LangGraph, demonstrating advanced mastery of complex LLM workflows.[27]

6.3 Comprehensive Evaluation Pipeline

A professional RAG system requires quantitative measurement of its performance. For the project report, the inclusion of an evaluation blueprint demonstrates technical rigor and reflection.[1] Metrics should focus on the quality of the entire pipeline, including:

- **Context Recall:** Whether the retriever successfully fetched all relevant documents needed to answer the query.
- **Faithfulness:** Whether the generated LLM response is supported strictly by the retrieved context.
- **Answer Relevance:** Whether the final output directly addresses the user's question.

Tools such as LangSmith can be employed to trace, debug, and systematically test the pipeline against a defined set of questions and expected answers, providing empirical evidence of the system's reliability and guiding future improvements.[26]

Works cited

1. EONVERSE AI Intern – Screening Challenge.pdf
2. Retrieval - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langchain/retrieval
3. RAG Pipeline: Example, Tools & How to Build It - lakeFS, accessed on November 13, 2025, https://lakefs.io/blog/what-is-rag-pipeline/
4. Build a RAG agent with LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langchain/rag
5. Best Open-Source Embedding Models Benchmarked and Ranked - Supermemory, accessed on November 13, 2025, https://supermemory.ai/blog/best-open-source-embedding-models-benchmarked-and-ranked/
6. Combining conversation history with source citations in LangChain RAG implementation, accessed on November 13, 2025, https://community.latenode.com/t/combining-conversation-history-with-source-citations-in-langchain-rag-implementation/34471
7. Best PDF Parsers for RAG Applications - DEV Community, accessed on November 13, 2025, https://dev.to/biomathcode/best-pdf-parsers-for-rag-applications-26j8
8. Retrieval augmented generation (RAG) | 🦜 Langchain, accessed on November 13, 2025, https://js.langchain.com/v0.1/docs/expression_language/cookbook/retrieval/
9. Build an LLM app using LangChain - Streamlit Docs, accessed on November 13, 2025, https://docs.streamlit.io/develop/tutorials/chat-and-llm-apps/llm-quickstart
10. Build a basic LLM chat app - Streamlit Docs, accessed on November 13, 2025, https://docs.streamlit.io/develop/tutorials/chat-and-llm-apps/build-conversational-apps
11. Chunk Tastic! Chunking Strategies for RAG - Advancing Analytics, accessed on November 13, 2025,

https://www.advancinganalytics.co.uk/blog/chunk-tastic-chunking-strategies-for-rag
12. ParentDocumentRetriever — LangChain documentation, accessed on November 13, 2025, https://api.python.langchain.com/en/latest/langchain/retrievers/langchain.retrievers.parent_document_retriever.ParentDocumentRetriever.html
13. Create a Smart RAG App with LangChain and Streamlit - DEV Community, accessed on November 13, 2025, https://dev.to/ngonidzashe/doc-sage-create-a-smart-rag-app-with-langchain-and-streamlit-4lin
14. RAG and Conversational Patterns: Building Intelligent Knowledge Systems | by Lijo Jose, accessed on November 13, 2025, https://medium.com/@lijojose/rag-and-conversational-patterns-building-intelligent-knowledge-systems-6e070bd7bd03
15. Document loaders - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/integrations/document_loaders
16. Best PDF Parser for RAG? : r/LangChain - Reddit, accessed on November 13, 2025, https://www.reddit.com/r/LangChain/comments/1dzj5qx/best_pdf_parser_for_rag/
17. Best Small LLM For Rag - Models - Hugging Face Forums, accessed on November 13, 2025, https://discuss.huggingface.co/t/best-small-llm-for-rag/143971
18. The 11 best open-source LLMs for 2025 - n8n Blog, accessed on November 13, 2025, https://blog.n8n.io/open-source-llm/
19. Implement RAG chunking strategies with LangChain and watsonx.ai - IBM, accessed on November 13, 2025, https://www.ibm.com/think/tutorials/chunking-strategies-for-rag-with-langchain-watsonx-ai
20. Beyond Basic RAG: A Practical Guide to Advanced Indexing Techniques - Towards AI, accessed on November 13, 2025, https://pub.towardsai.net/beyond-basic-rag-a-practical-guide-to-advanced-indexing-techniques-a3efe7c6d78c
21. RAG: Parent Document Retriever in LangChain - Kaggle, accessed on November 13, 2025, https://www.kaggle.com/code/marcinrutecki/rag-parent-document-retriever-in-langchain
22. RAG Tutorial 2025 #13: Advanced Document Retrieval Techniques - YouTube, accessed on November 13, 2025, https://www.youtube.com/watch?v=kNU-J4NNNhk
23. Application structure - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/langsmith/application-structure
24. Prep and deploy your app on Community Cloud - Streamlit Docs, accessed on November 13, 2025, https://docs.streamlit.io/deploy/streamlit-community-cloud/deploy-your-app
25. streamlit/example-app-langchain-rag - GitHub, accessed on November 13, 2025, https://github.com/streamlit/example-app-langchain-rag

26. Build a custom RAG agent - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langgraph/agentic-rag
27. Build an LLM RAG Chatbot With LangChain - Real Python, accessed on November 13, 2025, https://realpython.com/build-llm-rag-chatbot-with-langchain/