Building a Multi-Document RAG System

AMAN KHARWAL · JANUARY 6, 2026

RAG connects the powerful reasoning of an LLM with the unique information in your own documents. Today, I'll teach you how to build a Multi-Document RAG System using Python. By the end, you'll have an app that reads a folder of documents and answers your questions accurately.

Multi-Document RAG System: Getting Started

We are going to build a Multi-Document RAG system from scratch using Python, LangChain, and Ollama. It sounds complex, but I promise you, it's just a series of logical steps.

We'll use LangChain for orchestration, Chroma for storage, and Ollama to run the Llama 3 model locally.

First, install these libraries. In your terminal, run:

```
pip install langchain langchain-community langchain-huggingface langchain-chroma langchain-ollama pypdf
```

You'll also need **Ollama** running locally with the Llama 3 model. After installing Ollama, **run ollama pull llama3.**

Step 1: Loading the Raw Knowledge

First, gather your source materials. We need to extract text from PDF files. PyPDFLoader is a good choice because it handles the tricky formatting of PDFs well:

```python
import os
from langchain_community.document_loaders import PyPDFLoader

def load_documents(folder_path: str):
    if not os.path.exists(folder_path):
        raise FileNotFoundError(f"Folder '{folder_path}' does not exis

    documents = []
    for filename in os.listdir(folder_path):
        if filename.endswith(".pdf"):
            file_path = os.path.join(folder_path, filename)
            print(f"📄 Loading: {filename}")
            try:
                loader = PyPDFLoader(file_path)
                documents.extend(loader.load())
            except Exception as e:
                print(f"❌ Error loading {filename}: {e}")
    return documents
```

- Data is rarely perfect. Make sure your loading logic skips non-PDFs and handles errors, so your pipeline keeps running even if one file is bad.

Step 2: Chunking

You can't give a 100-page document to an LLM all at once because it goes over the memory limit. So, we need to break it into smaller parts.

We use RecursiveCharacterTextSplitter, which is a smart tool. It tries to split text by paragraphs first, then by sentences, so related text stays together:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

def split_text(documents):
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,
        chunk_overlap=200,
    )
    chunks = splitter.split_documents(documents)
    print(f"✂️ Created {len(chunks)} chunks")
```

```
    return chunks
```

Pay attention to **chunk_overlap=200**. This setting is important because it creates a sliding window, making sure you don't lose context if a sentence is split between chunks.

Step 3: Embeddings

Computers understand numbers, not words. So, we need to turn our text chunks into lists of numbers, called **vectors** or **embeddings**.

This means that if two chunks have similar meanings, like "Dog" and "Puppy," their numbers will be close to each other:

```
from langchain_huggingface import HuggingFaceEmbeddings

embedding_function = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)
```

- We'll use all-MiniLM-L6-v2, a lightweight, open-source model that runs quickly on your CPU.

Step 4: The Vector Store

Now we need somewhere to store these numbers for fast searching. A regular SQL database isn't good for this, so we'll use a **Vector Database**. We'll use **Chroma**:

```
from langchain_chroma import Chroma

def create_vector_store(chunks):
    vector_store = Chroma.from_documents(
        documents=chunks,
        embedding=embedding_function,
        persist_directory="./chroma_db",
        collection_name="rag_docs"
    )
    return vector_store
```

This function saves the database in a folder called **./chroma_db**. That way, you don't have to rebuild the database every time you restart the app; it stays saved.

Step 5: The Brain

This is the most important part. This function links the user, the database, and the LLM:

```
from langchain_ollama import ChatOllama
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.output_parsers import StrOutputParser


def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)


def query_rag_system(query_text, vector_store):
    llm = ChatOllama(model="llama3") # Make sure you have Ollama insta

    retriever = vector_store.as_retriever(search_kwargs={"k": 3})

    prompt = ChatPromptTemplate.from_template(
        """
        You are a helpful assistant.
```

```
        Answer ONLY using the context below.
        If the answer is not present, say "I don't know."

        Context:
        {context}

        Question:
        {question}
        """
    )

    chain = (
        {
            "context": retriever | format_docs,
            "question": RunnablePassthrough(),
        }
        | prompt
        | llm
        | StrOutputParser()
    )

    return chain.invoke(query_text)
```

First, it looks at the user's question and finds the top 3 most relevant chunks **(k=3)**. Then, it puts those chunks into a strict prompt: **"Answer ONLY using the context below."** This helps stop the AI from making things up.

Step 6: Putting It All Together

Finally, the main loop checks if a database already exists. If it doesn't, it processes the PDFs. Then, it starts a chat loop so you can ask questions:

```
def main():
    folder_path = "/Users/amankharwal/aiagent/data" # CHANGE THIS to y

    if not os.path.exists("./chroma_db"):
        print("📦 No vector DB found. Creating one...")
        docs = load_documents(folder_path)
        chunks = split_text(docs)
        vector_store = create_vector_store(chunks)
        print("Vector database created")
    else:
        print("📦 Loading existing vector DB...")
```

```
        vector_store = Chroma(
            persist_directory="./chroma_db",
            embedding_function=embedding_function,
            collection_name="rag_docs"
        )

    while True:
        query = input("\n❓ Ask a question (or type 'exit'): ")
        if query.lower() == "exit":
            break

        print("🤔 Thinking...")
        answer = query_rag_system(query, vector_store)
        print("\n🧠 Answer:\n", answer)

if __name__ == "__main__":
    main()
```

Here’s the answer I got for my and my friends’ resumes:

```
(env) (base) amankharwal@Amans-MacBook-Pro aiagent % python multirag.py
📦 Loading existing vector DB...

❓ Ask a question (or type 'exit'): Compare Tushar and Aman for the role of a Data Analyst
🤔 Thinking...

🧠 Answer:
 Based on the provided context, here's a comparison between Tushar and Aman for the role of a Data Analyst:

**Similarities:**

* Both Tushar and Aman have experience with data-related tools like SQL, Power BI, and Excel.
* They both have a strong foundation in analytical thinking.

**Differences:**

* **Experience:** Aman has 5+ years of experience building intelligent systems using Machine Learning, NLP, and Generative AI,
 whereas Tushar is a recent graduate with limited work experience (February 2024 – September 2024).
* **Specialization:** Aman specializes in developing LLM-based systems, RAG chatbots, and Multi-Agent architectures that solve
 real-world business challenges. Tushar has not mentioned any specific areas of specialization.
* **Background:** Aman is the founder of Statso.io and has led data-driven projects for startups and enterprises, whereas Tush
ar is a recent graduate with limited work experience.

**Conclusion:**

Based on the provided context, Aman appears to be a more suitable candidate for the role of a Data Analyst due to his extensiv
e experience in building intelligent systems using Machine Learning, NLP, and Generative AI. However, it's essential to consid
er Tushar's skills and background as well. If you're looking for a recent graduate with strong foundations in SQL, Power BI, a
nd analytical thinking, Tushar might be a good fit.

Please note that this comparison is based solely on the provided context, and additional information may be necessary to make
a more informed decision.
```

Closing Thoughts

Building systems like this shows me that AI isn’t meant to replace our curiosity; it helps fuel it. When it’s easier to find answers, we can ask better, deeper, and more creative questions.

Don’t be afraid to experiment with this code. Try changing the chunk size, swap llama3 for Mistral, or use a different embedding model. That’s the best way to learn.

If you found this article useful, you can follow me on Instagram for daily AI tips and practical resources. You might also like my latest book, Hands-On GenAI, LLMs & AI Agents. It’s a step-by-step guide to help you get ready for jobs in today’s AI field.

Aman Kharwal

AI/ML Engineer | Published Author. My aim is to decode data science for the real world in the most simple words.

PREVIOUS POST

The Agentic AI Engineer Roadmap for 2026

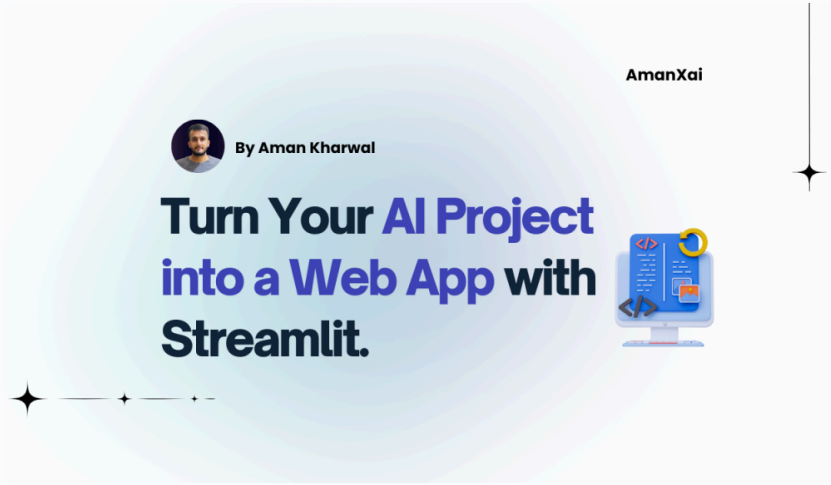
NEXT POST

Top Companies Hiring for GenAI Roles

Recommended For You

4 End-to-End AI/ML Projects to Get Job-Ready

January 10, 2026



Build Your First AI App UI with Streamlit

January 8, 2026 / 3 Comments

Top Companies Hiring for GenAI Roles

January 7, 2026

The Agentic AI Engineer Roadmap for 2026

January 5, 2026 / 2 Comments

Leave a Reply