Freedium

< Go to the original

Building a Simple RAG System from Scratch: A Comprehensive Guide

Photo by Steve Johnson on Unsplash

Bishal Bose

Follow

a11y-light · April 25, 2025 (Updated: April 25, 2025) · Free: Yes

Introduction

In the era of large language models (LLMs), one of the biggest challenges is ensuring these powerful AI systems can reliably access and utilize specific information. While LLMs possess impressive reasoning capabilities and vast general knowledge, they often struggle with retrieving precise information, accessing up-to-date that enhances LLMs by grounding them in external knowledge sources.

In this comprehensive guide, we'll explore what RAG is, why it matters, and then build a complete RAG system from scratch using Python and popular open-source libraries. By the end, you'll understand not just the concept but also have a functional implementation that you can adapt to your own use cases.

What is RAG?

Photo by Magnet.me on Unsplash

Retrieval-Augmented Generation (RAG) is an architecture that combines information retrieval with text generation. At its core, RAG enhances language models by retrieving relevant information from an external knowledge base before generating a response.

1. **Retrieval:** When a query is received, a retrieval system searches a knowledge base for the most relevant documents or text chunks.
2. **Augmentation:** The retrieved information is injected into the prompt sent to the language model.
3. **Generation:** The language model generates a response that's informed by both its pre-trained knowledge and the specifically retrieved information.

This approach solves several critical problems with traditional LLMs:

- **Knowledge limitations:** Standard LLMs are limited to what they learned during training. RAG allows them to access more recent or specialized information.
- **Hallucinations:** LLMs sometimes generate plausible-sounding but incorrect information. RAG reduces this by grounding responses in verifiable sources.
- **Transparency:** With RAG, the model can cite its sources, making it easier to verify its responses.
- **Adaptability:** RAG systems can be updated with new information without retraining the entire model.

The Development of RAG

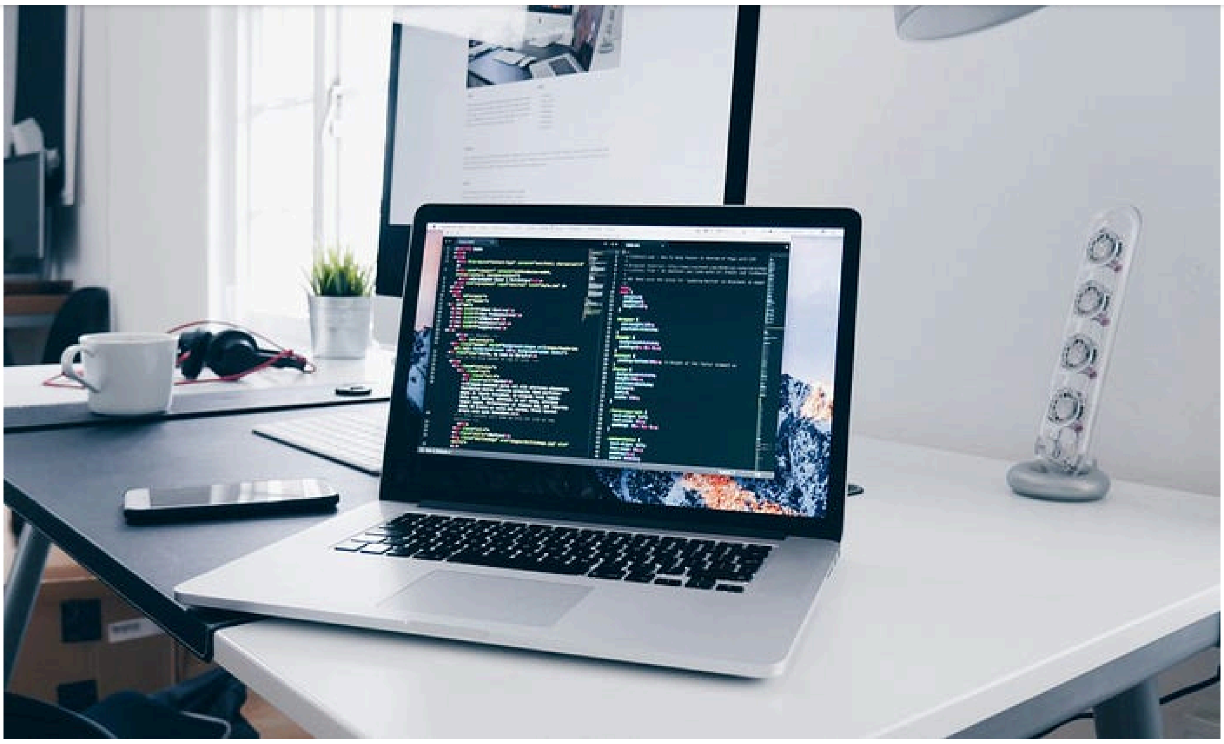
Photo by Christopher Gower on Unsplash

The concept of RAG was formally introduced in a 2020 paper titled "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" by researchers from Facebook AI Research (now Meta AI). This paper proposed combining sparse and dense retrievers with a sequence-to-sequence model for knowledge-intensive tasks.

However, the ideas behind RAG have deeper roots in various fields:

- **Question Answering Systems:** Early QA systems used document retrieval to find relevant information before attempting to answer questions.
- **Information Retrieval:** Decades of research in search engines provided the foundation for efficient document retrieval.
- **Neural Information Retrieval:** The development of neural networks for information retrieval enabled more semantically meaningful searches.

retrieval.

The explosion in popularity of RAG coincided with the rise of powerful LLMs like GPT-3, GPT-4, Claude, and open-source alternatives like LLaMA. Organizations quickly realized that while these models were powerful, they needed to be grounded in trusted information sources to be reliable for business applications.

Today, RAG has become a cornerstone of applied LLM development, with frameworks like LangChain, LlamaIndex, and others providing tools to simplify implementation.

Why RAG Matters

Photo by Oyemike Princewill on Unsplash

RAG offers several compelling advantages:

1. **Up-to-date information:** RAG systems can access the latest information, overcoming the knowledge cutoff limitations of LLMs.
2. **Domain specialization:** By providing domain-specific knowledge bases, RAG can make general LLMs perform like specialized models.
3. **Reduced hallucinations:** By grounding responses in retrieved documents, RAG significantly reduces the tendency of LLMs to generate incorrect information.
4. **Lower costs:** Instead of fine-tuning or retraining large models, RAG allows adaptation to new domains by simply changing the knowledge base.

6. **Privacy and security:** Sensitive information can remain in a controlled knowledge base rather than being included in model training data.

Building a RAG System: Core Components

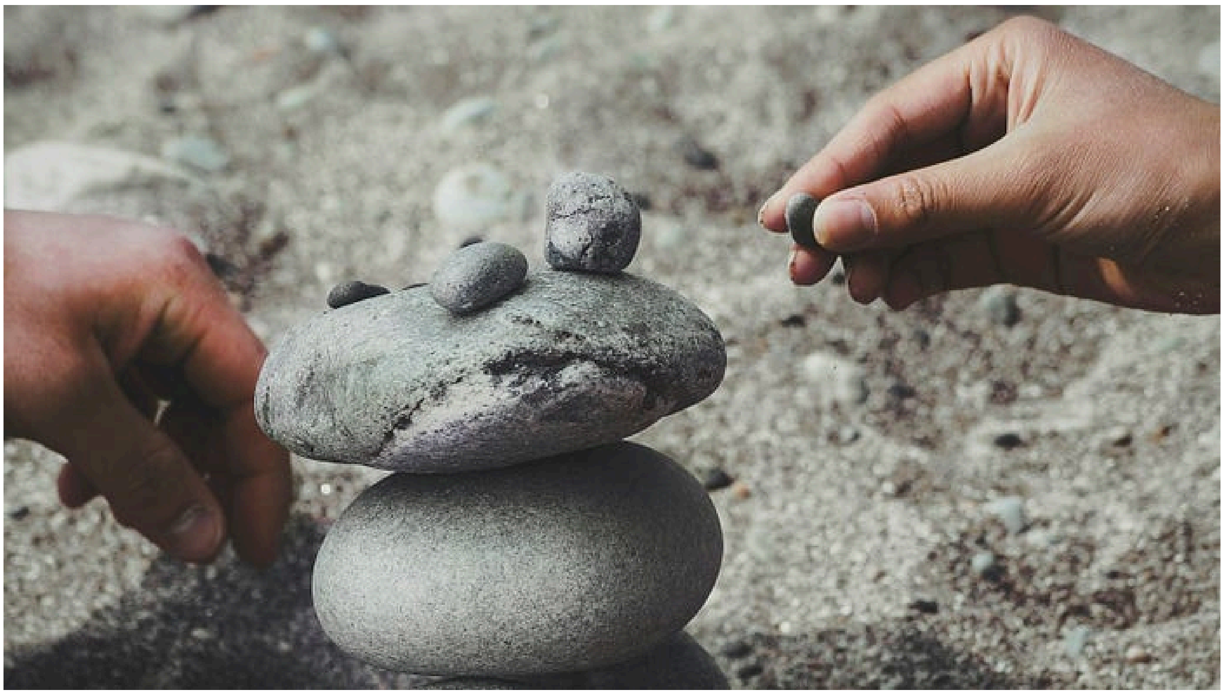
Photo by Andrik Langfield on Unsplash

A typical RAG system consists of several key components:

1. **Document Loader:** Imports documents from various sources (PDFs, web pages, databases, etc.)
2. **Text Chunker:** Splits documents into manageable pieces for indexing and retrieval
3. **Embedding Model:** Converts text chunks into numerical vectors that capture semantic meaning
4. **Vector Store:** Indexes and stores the vectors for efficient retrieval

6. **Language Model:** Generates responses based on the query and retrieved information
7. **Prompt Templates:** Instructs the model on how to use the retrieved information

Now, let's build a complete RAG system using your code as a foundation.

Implementation: Building a RAG System Step by Step

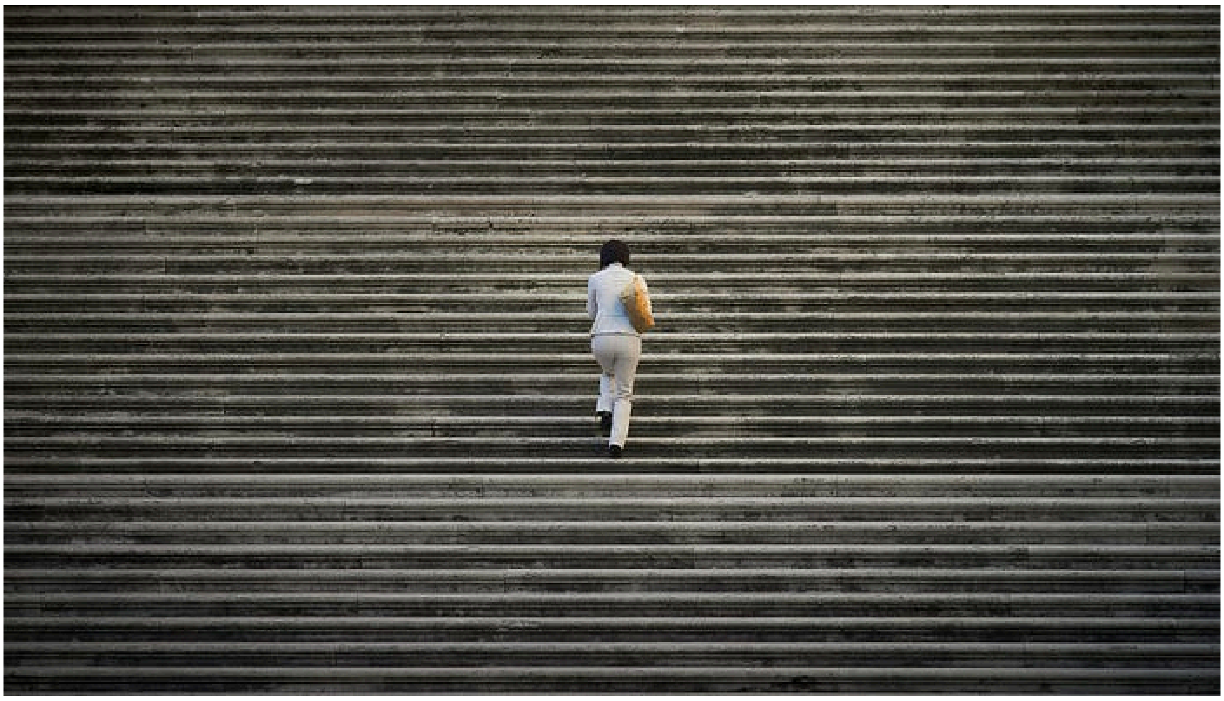

Photo by Vidar Nordli-Mathisen on Unsplash

Setting Up the Environment

First, let's understand the requirements for our RAG system. We'll need several Python libraries:

Copy

```
langchain
langchain-core
langchain-community
```

```
faiss-cpu
langchain-ollama
langchain-openai
```

These libraries provide the foundation for our RAG implementation:

- LangChain provides the overall framework and components for building LLM applications
- PyMuPDF enables us to extract text from PDF documents
- FAISS offers efficient similarity search capabilities for our vector database
- The Ollama and OpenAI integrations allow us to use different language models

You can install these requirements using pip:

Copy

```
pip install langchain langchain-core langchain-community langchain
```

Now, let's dive into the implementation of each component.

Component 1: PDF Loader

Freedium

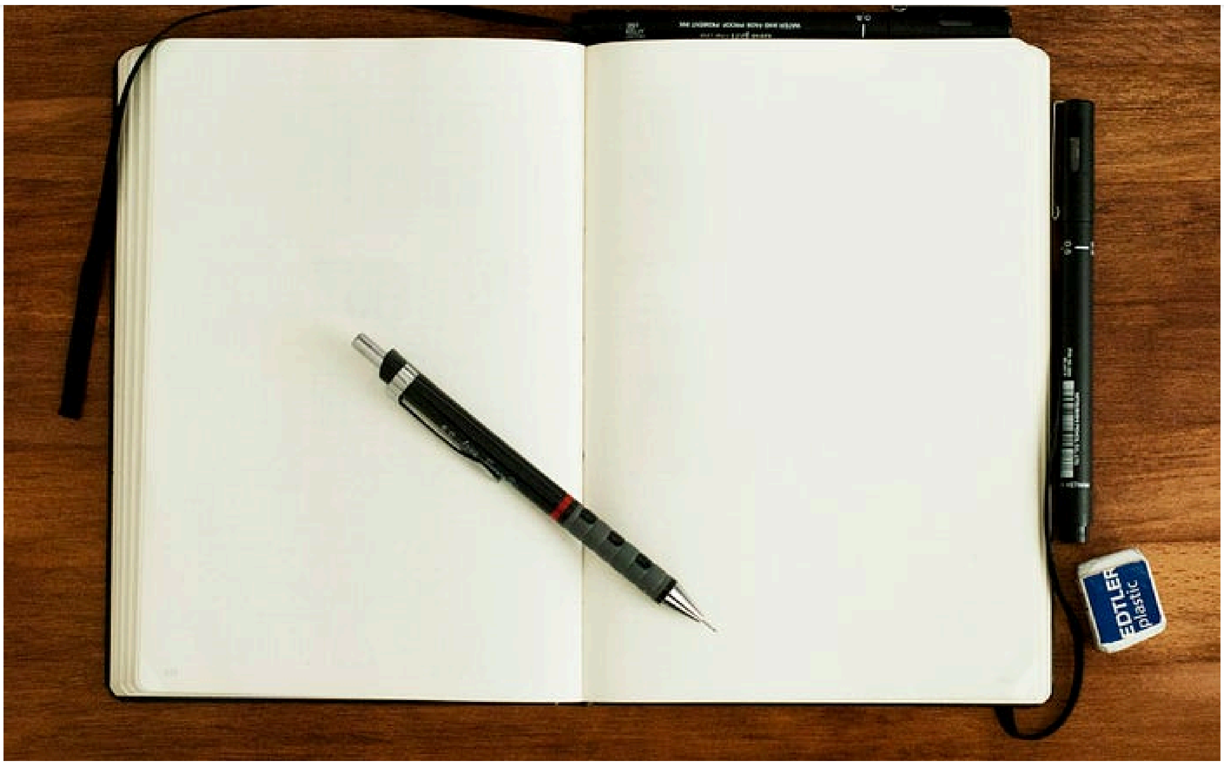

Photo by Mike Tinnion on Unsplash

First, we need a way to ingest PDF documents into our system:

Copy

```
from langchain_community.document_loaders import PyMuPDFLoader

class PdfLoader:
  def __init__(self, ):
    pass

  def read_file(self, file_path):
    loader = PyMuPDFLoader(file_path)
    docs = loader.load()
    return docs
  pass
```

Let's break down what this code does:

1. We import the `PyMuPDFLoader` from LangChain's community components. This loader uses the PyMuPDF library (also known as fitz) to extract text from PDF files.

3. When `read_file` is called, it:

- Creates a new `PyMuPDFLoader` instance for the specified file
- Calls the loader's `load()` method to extract text from the PDF
- Returns the loaded documents

The `load()` method returns a list of `Document` objects, where each document represents a page from the PDF. Each `Document` contains:

- `page_content` : The extracted text from the page
- `metadata` : Additional information about the document, including the source file path and page number

This loader is particularly powerful because PyMuPDF efficiently handles various PDF features including text, tables, and even some images through OCR. For a production system, you might extend this class to handle other document types like Word documents, web pages, or even database records.

Component 2: Text Chunking

Photo by Megan Watson on Unsplash

Once we've loaded our documents, we need to split them into smaller chunks for effective retrieval:

Copy

```python
from langchain_text_splitters import RecursiveCharacterTextSplitte
from langchain_core.documents import Document

class Chunker:
  def __init__(self, chunk_size=1000, chunk_overlap=100):
    self.text_splitter = RecursiveCharacterTextSplitter(
      # Set a really small chunk size, just to show.
      separators=[
          "\n\n",
          "\n",
          " ",
          ".",
          ",",
          "\u200b",  # Zero-width space
          "\uff0c",  # Fullwidth comma
          "\u3001",  # Ideographic comma
          "\uff0e",  # Fullwidth full stop
          "\u3002",  # Ideographic full stop
```

```
        chunk_overlap=chunk_overlap,
        length_function=len,
        is_separator_regex=False,
    )
    pass
    def chunk_docs(self, docs):
      list_of_docs = []
      for doc in docs:
        tmp = self.text_splitter.split_text(doc.page_content)
        for chunk in tmp:
          list_of_docs.append(
              Document(
              page_content=chunk,
              metadata=doc.metadata,
            )
          )
      return list_of_docs
```

This chunking component is crucial for effective retrieval. Let's examine it line by line:

1. We import the `RecursiveCharacterTextSplitter` from LangChain's text splitters library and the `Document` class from LangChain core.
2. We define a `Chunker` class that takes optional parameters for chunk size and overlap:

- `chunk_size` : The target size for each text chunk (1000 characters by default)
- `chunk_overlap` : How much adjacent chunks should overlap (100 characters by default)

3. In the constructor, we create a `RecursiveCharacterTextSplitter` with specific settings:

- `separators` : A prioritized list of delimiters used to split the text. The splitter tries each separator in order until chunks of

- `chunk_overlap` : Overlap between consecutive chunks to maintain context
- `length_function` : Function used to measure chunk size (here, Python's built-in `len`)
- `is_separator_regex` : Flag indicating whether separators are regular expressions

4. The `chunk_docs` method processes a list of documents:

- For each document, it extracts the page content
- Splits the content into chunks using the text splitter
- Creates new `Document` objects for each chunk, preserving the original metadata
- Returns a list of these chunked documents

The choice of chunking strategy is crucial for RAG performance. If chunks are too large, they might contain too much irrelevant information. If they're too small, they might lose important context. The overlap between chunks helps maintain coherence when information spans chunk boundaries.

The separator list is particularly interesting here. It starts with paragraph breaks, then line breaks, then spaces, and finally punctuation marks. This hierarchy ensures the system tries to break text at natural boundaries first before resorting to splitting sentences or words. The inclusion of Unicode characters like zero-width spaces and ideographic punctuation makes this chunker more international, handling texts in languages like Chinese and Japanese.

Component 3: Vector Store

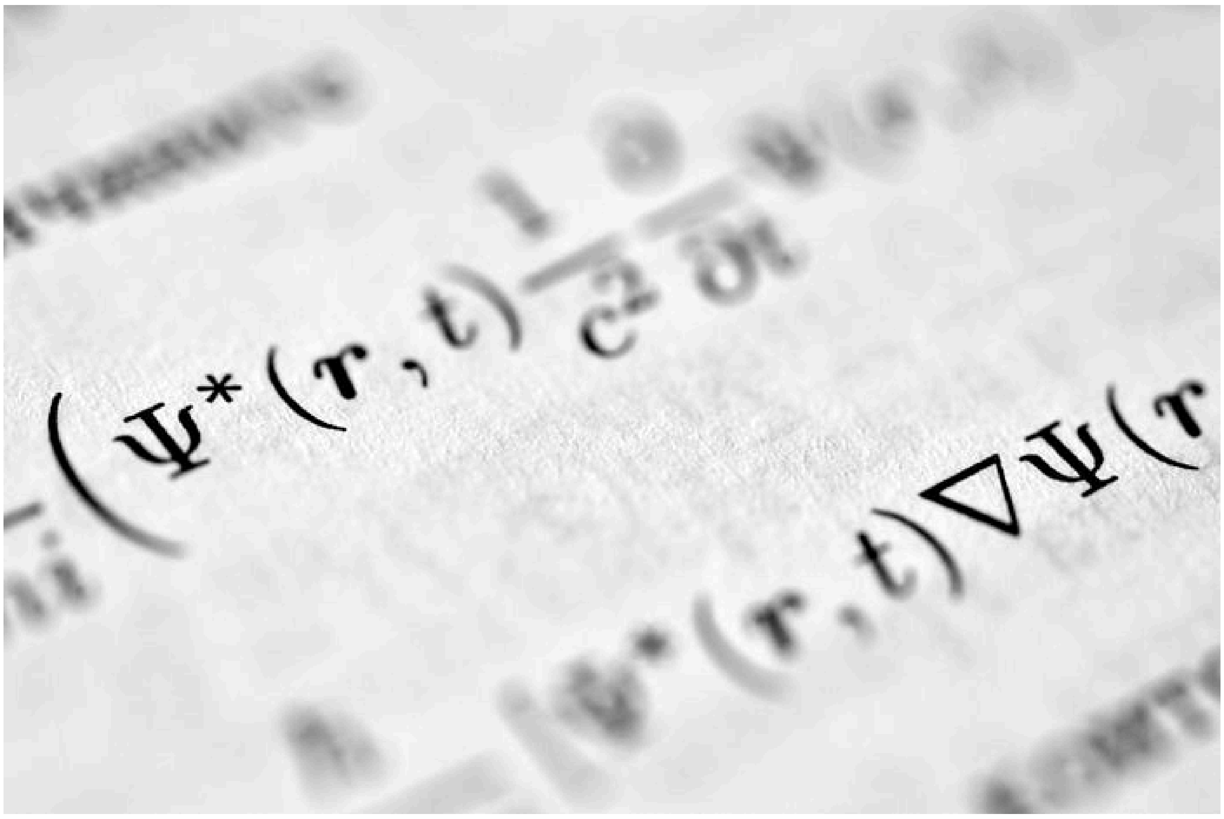

Photo by Bozhin Karaivanov on Unsplash

Now that we have our document chunks, we need to convert them to vector embeddings and store them for efficient retrieval:

Copy

```
import faiss
from langchain_community.docstore.in_memory import InMemoryDocstor
from langchain_community.vectorstores import FAISS
from langchain_ollama import OllamaEmbeddings
from uuid import uuid4


class VectorStore:
  def __init__(self, ):
    self.embeddings = OllamaEmbeddings(model="llama3.2:3b")
    self.index = faiss.IndexFlatL2(len(self.embeddings.embed_query
    self.vector_store = FAISS(
        embedding_function=self.embeddings,
        index=self.index,
    )
    pass

  def add_docs(self, list_of_docs):
    uuids = [str(uuid4()) for _ in range(len(list_of_docs))]
    self.vector_store.add_documents(documents=list_of_docs, ids=uu
    pass

  def search_docs(self, query, k=5):
    results = self.vector_store.similarity_search(
        query,
        k=k,
    )
    return results
    pass
  pass
```

This component is the heart of our retrieval system. Let's dissect how it works:

1. We import necessary libraries:

- `faiss` : Facebook AI Similarity Search, a library for efficient similarity search and clustering of dense vectors
- `InMemoryDocstore` : A simple in-memory document store from LangChain
- `FAISS` : LangChain's wrapper for the FAISS library
- `OllamaEmbeddings` : Integration with Ollama's embedding models
- `uuid4` : For generating unique identifiers for our documents

2. In the constructor, we:

- Initialize an embedding model (here using Llama 3.2 through Ollama)
- Create a FAISS index configured for L2 (Euclidean) distance with appropriate dimensions

3. The `add_docs` method:

- Generates unique IDs for each document
- Adds the documents to the vector store, which:
- Computes embeddings for each document's content
- Indexes these embeddings in FAISS
- Stores the original documents with their IDs

4. The `search_docs` method:

- Takes a query and a parameter `k` (number of results to return)
- Converts the query to an embedding using the same model
- Performs a similarity search in the vector store
- Returns the `k` most similar documents

The code shows a flexible approach where you can choose between different embedding models. While OpenAI's embedding models (commented out in the code) generally provide excellent performance, using Ollama with local LLaMA models offers a completely self-hosted alternative.

The L2 (Euclidean) distance measure is used here, but other options exist in FAISS, like cosine similarity or inner product. The choice depends on how your embeddings are normalized and what similarity concept makes most sense for your application.

For production use, you might consider:

- Using a persistent vector store instead of an in-memory one
- Adding metadata filtering capabilities

Component 4: The RAG System

Photo by Markus Spiske on Unsplash

Finally, we can put all these components together into a complete RAG system:

Copy

```
from langchain_core.prompts import PromptTemplate
from langchain_openai import OpenAI
from langchain_ollama import OllamaLLM
from pdf_loader import PdfLoader
from vector_store import VectorStore
from chunk_text import Chunker

class RAG:

  def __init__(self, ):
    self.instructor_prompt = """Instruction: You're an expert prob
    Answer context: {answer_context}
    """
    self.prompt = PromptTemplate.from_template(self.instructor_pro
    self.llm = OllamaLLM(model="llama3.2:3b") #OpenAI()
    self.vectorStore = VectorStore()
    self.pdfloader = PdfLoader()
    self.chunker = Chunker()
    pass

  def run(self, filePath, query):
    docs = self.pdfloader.read_file(filePath)
    list_of_docs = self.chunker.chunk_docs(docs)
    self.vectorStore.add_docs(list_of_docs)
    results = self.vectorStore.search_docs(query)
    answer_context = "\n\n"
    for res in results:
      answer_context = answer_context + "\n\n" + res.page_content
    chain = self.prompt | self.llm
    response = chain.invoke(
        {
            "user_query": query,
            "answer_context": answer_context,
        }
    )
    return response
  pass

if __name__ == "__main__":
  rag = RAG()
  filePath="investment.pdf"
  query="How to invest?"
  response = rag.run(filePath, query)
  print(response)
  pass
```

This main class brings together all the components we've built into a complete RAG pipeline. Let's analyze it step by step:

1. We import our custom components and LangChain utilities:

- `PromptTemplate` : For structuring prompts to the language model
- `OllamaLLM` and `OpenAI` : Alternative language model integrations

2. In the constructor, we:

- Define an instructor prompt template that will guide the LLM
- Create a PromptTemplate object from this template
- Initialize the language model (here using Llama 3.2 through Ollama)
- Initialize our vector store, PDF loader, and chunker components

3. The `run` method implements the complete RAG workflow:

- Load a PDF document using our PDF loader
- Chunk the document into smaller pieces
- Add the chunks to our vector store
- Search for relevant chunks based on the user's query
- Combine the retrieved chunks into a context string
- Create a chain that combines our prompt template with the language model
- Invoke this chain with the user query and retrieved context
- Return the generated response

4. The main block demonstrates basic usage:

- Create a RAG instance
- Specify a PDF file and a query
- Run the RAG system and print the response

The instructor prompt is particularly important here. It guides the language model on how to use the retrieved context:

- Provides an honesty policy ("if you do not find the answer... say 'I Don't know!'")
- Clearly demarcates the user question from the retrieved context

This structured approach helps prevent the model from hallucinating or drawing too heavily on its pre-trained knowledge when it should be focusing on the retrieved information.

Advanced Considerations and Improvements

Photo by Andy Kelly on Unsplash

While our implementation provides a solid foundation, there are several ways to enhance it for production use:

1. Document Processing Enhancements

- **Document metadata extraction:** Capture and use metadata like creation date, author, and title
- **OCR integration:** Add optical character recognition for scanned documents or images
- **Table extraction:** Specialized handling for tabular data within documents

2. Chunking Strategies

- **Semantic chunking:** Split documents based on semantic meaning rather than character count
- **Hierarchical chunking:** Maintain document structure with parent-child relationships between chunks
- **Metadata-enhanced chunking:** Include section titles or document structure in chunk metadata

3. Embedding and Retrieval Improvements

- **Reranking:** Add a secondary ranking step to refine initial retrieval results
- **Hybrid search:** Combine vector similarity with keyword-based (BM25) search
- **Query expansion:** Automatically enhance queries to improve retrieval performance
- **Cross-encoder reranking:** Use a more expensive but accurate model to rerank initial results

4. LLM Integration

- **Chain-of-thought reasoning:** Modify prompts to encourage step-by-step reasoning
- **Self-critique:** Have the model evaluate and refine its own responses
- **Multi-step reasoning:** Break complex queries into sub-questions

5. Evaluation and Monitoring

- **Relevance evaluation:** Measure how relevant retrieved documents are to queries
- **Answer accuracy:** Compare generated answers against ground truth when available
- **Hallucination detection:** Techniques to identify when the model is fabricating information
- **User feedback loops:** Incorporate user feedback to improve system performance

Scaling RAG for Production

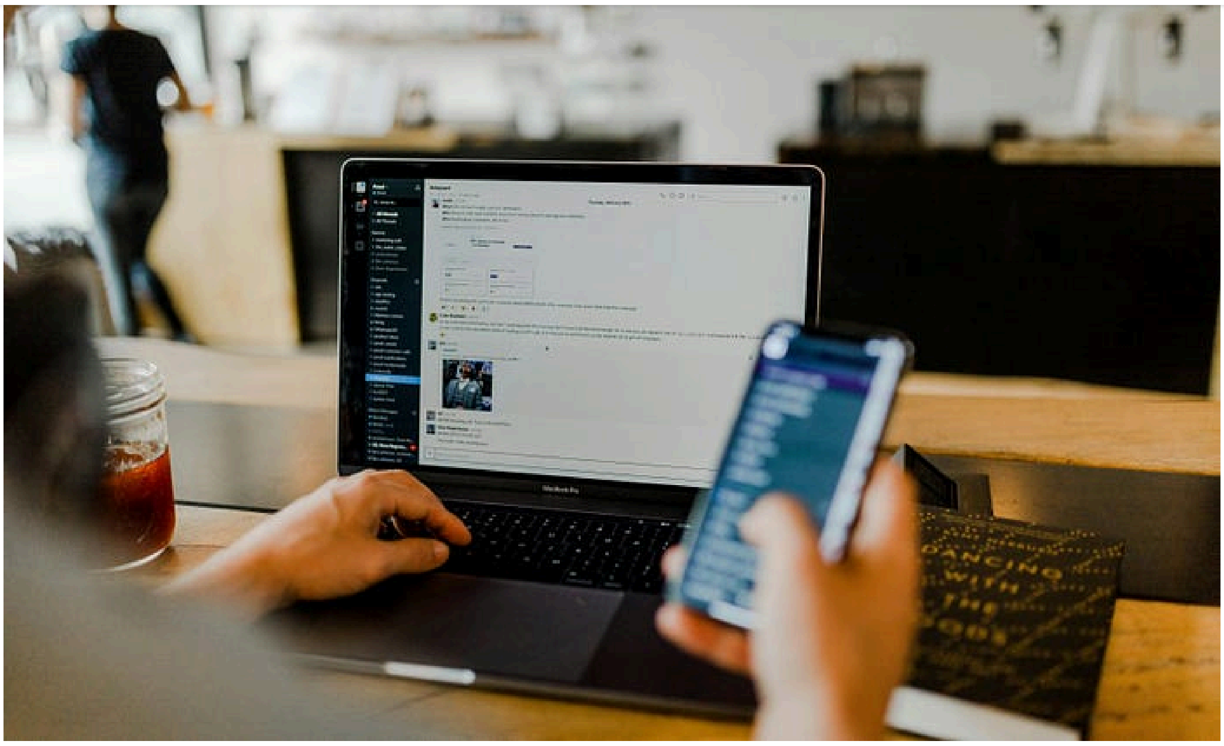

Photo by Austin Distel on Unsplash

To deploy RAG in production environments, consider:

1. Persistence

- Store embeddings and documents in a persistent database (Pinecone, Weaviate, Chroma, etc.)
- Implement incremental updates to avoid reprocessing entire document collections

2. Performance Optimization

- Pre-compute embeddings for document collections
- Implement caching at various levels (query, embeddings, responses)
- Use quantization to reduce embedding size and improve search speed

3. Infrastructure

- Containerize components for easier deployment
- Implement microservices architecture for scalability
- Use queue systems for asynchronous processing of large document batches

4. Security and Compliance

- Implement access controls for sensitive documents
- Add logging for audit trails
- Ensure proper handling of personally identifiable information (PII)

Conclusion

Photo by bruce mars on Unsplash

trustworthy. By grounding LLM responses in specific knowledge sources, RAG systems can provide more accurate, up-to-date, and verifiable information than standalone language models.

Our implementation demonstrates the core components of a RAG system:

1. Document loading with PyMuPDF
2. Text chunking with RecursiveCharacterTextSplitter
3. Vector storage and retrieval with FAISS
4. Response generation with a language model guided by carefully crafted prompts

While this implementation provides a solid foundation, RAG is an evolving field with ongoing research and improvements. By understanding the principles and components outlined in this guide, you can build increasingly sophisticated RAG systems tailored to your specific needs.

Whether you're creating a customer support system, a research assistant, or a domain-specific expert system, RAG provides a powerful framework for combining the reasoning capabilities of LLMs with the precision of information retrieval.

Additional Resources

For those interested in diving deeper into RAG, consider exploring:

- LangChain's RAG documentation
- LlamaIndex's guides on RAG systems
- The original RAG paper from Meta AI Research

- Self-RAG for improved retrieval and generation

By continuing to build on these foundations, you can create increasingly powerful, accurate, and helpful AI systems that combine the best of both retrieval and generation.

Photo by Markus Spiske on Unsplash

#artificial-intelligence #rags #ai #artificial-neural-network #advanced-ai