**Simple RAG Implementation Use Case**

Let me walk you through a simple implementation of a Retrieval Augmented Generation (RAG) system that answers questions based on a document collection.

**Use Case: Document Question Answering**

In this example, we'll build a simple RAG system that allows users to ask questions about a collection of text documents. The system will:

1. Process a collection of documents and store them in a vector database
2. Accept user queries
3. Retrieve relevant document chunks
4. Generate answers based on those chunks

Here's the implementation:

import os

from langchain.document\_loaders import TextLoader

from langchain.text\_splitter import CharacterTextSplitter

from langchain.embeddings import OpenAIEmbeddings

from langchain.vectorstores import FAISS

from langchain.chains import RetrievalQA

from langchain.llms import OpenAI

# 1. Load documents

loader = TextLoader("my\_document.txt")

documents = loader.load()

# 2. Split documents into chunks

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

chunks = text\_splitter.split\_documents(documents)

# 3. Create embeddings and vector store

embeddings = OpenAIEmbeddings()

vectorstore = FAISS.from\_documents(chunks, embeddings)

# 4. Create a RAG chain

llm = OpenAI(temperature=0)

qa\_chain = RetrievalQA.from\_chain\_type(

llm=llm,

chain\_type="stuff",

retriever=vectorstore.as\_retriever()

)

# 5. Ask questions

question = "What are the key points discussed in the document?"

answer = qa\_chain.run(question)

print(answer)

**How This Works - Step by Step Explanation**

**1. Document Loading**

loader = TextLoader("my\_document.txt")

documents = loader.load()

This loads a text document into memory. You can use different loaders for various file types (PDF, HTML, etc.).

**2. Document Chunking**

text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

chunks = text\_splitter.split\_documents(documents)

Large documents are split into manageable chunks. The chunk\_size parameter determines how many characters are in each chunk, and chunk\_overlap allows for some overlap between chunks to prevent cutting sentences in the middle.

**3. Embedding Creation**

embeddings = OpenAIEmbeddings()

vectorstore = FAISS.from\_documents(chunks, embeddings)

Each text chunk is converted into a numerical vector (embedding) that captures its semantic meaning. These vectors are stored in FAISS, a vector database that allows for efficient similarity search.

**4. RAG Chain Setup**

llm = OpenAI(temperature=0)

qa\_chain = RetrievalQA.from\_chain\_type(

llm=llm,

chain\_type="stuff",

retriever=vectorstore.as\_retriever()

)

This creates a question-answering chain that:

* Uses OpenAI's model (like GPT-3.5 or GPT-4) as the language model
* Sets temperature to 0 for deterministic, factual responses
* Uses the "stuff" chain type, which combines all retrieved documents into one prompt
* Connects the vector store as a retriever

**5. Question Answering**

question = "What are the key points discussed in the document?"

answer = qa\_chain.run(question)

print(answer)

When a question is asked:

1. The question is converted to an embedding vector
2. Similar chunks are retrieved from the vector store
3. The chunks and question are combined into a prompt
4. The LLM generates an answer based on the provided context

**Key Benefits of This Approach**

* **Contextual relevance**: Answers are based on the actual content of your documents
* **Reduced hallucinations**: The LLM has specific reference material rather than relying solely on trained knowledge
* **Scalability**: Can handle large document collections through efficient vector search
* **Customization**: Can be tailored to specific domains or knowledge bases

Would you like me to elaborate on any specific part of this implementation or explain other RAG variations?