Introduction to RAG with Python & LangChain

Joey O'Neill Follow 7 min read · Jan 7, 2025

GitHub - joeyoneill/Medium_RAG_Series: Code & Items for the RAG Series tutorials posted to Medium

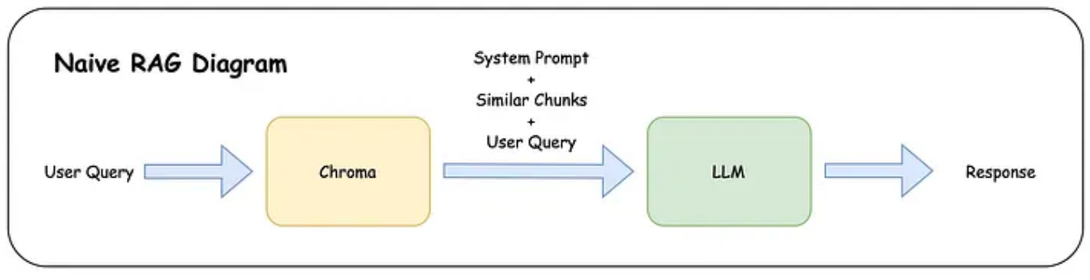
Code & Items for the RAG Series tutorials posted to Medium - joeyoneill/Medium_RAG_Series

github.com

Hello, everyone! I'm Joey O'Neill, a Technical Cloud & Custom Applications Consultant, where I specialize in Azure and have extensive experience working on LLM-based RAG (Retrieval-Augmented Generation) applications. This is an introductory article to an article series where I will guide you through building RAG applications at various levels, progressing toward a comprehensive full-stack solution. This series is intended for readers with a foundational understanding of the relevant technologies, as I'll focus on practical, step-by-step walkthroughs rather than deep dives into the underlying concepts. Be sure to follow me to stay updated with the series! For this article, we will be following along with the 01_Intro_to_RAG folder in the Git repo.

As a quick rundown, RAG (Retrieval-Augmented Generation) is a technique where external content is provided to a model as context, allowing it to respond more accurately and informatively. Think of the LLM as a base "brain" with a general understanding of concepts and limited knowledge of new or proprietary information that it wasn't trained on. By using RAG, we can supply the model with relevant missing information, enabling it to draw on this added context to answer queries more effectively.

Throughout this series, we'll primarily use Python and LangChain, along with other tools and packages introduced as needed in each article. For those unfamiliar, LangChain is a versatile framework designed to help developers easily build applications with LLMs, available in both Python and JavaScript. LangChain will be our main interface for working with LLMs in this series — but that is enough rambling. Let's start with the basics: a straightforward walkthrough of a naive RAG flow in Python.



Overview of a Naive Retrieval-Augmented Generation (RAG) Workflow

A Naive RAG Flow in Python — Environment Set Up

In this tutorial, we'll walk through a basic RAG flow using Python, LangChain, ChromaDB, and OpenAI. A basic RAG flow generally consists of two main components: an index and a large language model (LLM). The index stores data in easily retrievable chunks that can be queried and provided to the LLM as context. This allows the LLM to answer queries using information it wasn't originally trained on, as mentioned earlier. For

indexing, we'll use ChromaDB, a free and open-source vector database, and
Op
wh

For this tutorial, we will be using the following version of python:

```
python==3.11.9
```

Create a virtual python environment with the following command:

```
python -m venv .venv
```

Activate the virtual environment with the following command:

For Windows Users:

```
.\.venv\Scripts\activate
```

For Mac Users:

```
source .venv/bin/activate
```

For the packages, we will mainly be using the following:

```
langchain-chroma==0.1.4
langchain-core==0.3.15
langchain-openai==0.2.6
langchain-text-splitters==0.3.2
python-dotenv==1.0.1
```

Install the packages using *pip* either directly or copy them into a *requirements.txt* file.

Let's create an environment file (*.env*) to securely store environment variables like API keys and other sensitive information. This file keeps your credentials safe by allowing you to separate them from the main codebase. Later, we'll add it to a *.gitignore* file to ensure that any keys or secrets won't be exposed if you upload the project to a public repository.

In the *.env* file, create a variable to store your OpenAI API key and ensure you use the same variable name as below. Please note, you will need to get your own API key from OpenAI and have credits on your account.

```
OPENAI_API_KEY='<YOUR API KEY HERE>'
```

With that created, save it and let's move on to the *.gitignore* file.

Create the *.gitignore* file and include the following and save it:

```
*.env
.env
.venv/
__pycache__/
```

Finally, create the python file that you will be working in. You can do this in a . (.i

Now, let's load the environment variables into the python file and save it:

```
# imports
from dotenv import load_dotenv

# load in the .env variables
load_dotenv()
```

With all these files created, you should have a directory that looks something similar to this:

```
project-folder/
├── .env
├── .gitignore
├── main.ipynb
└── .venv/
```

And our environment is set up and ready to go!

Setting up the Index

As mentioned, we'll use ChromaDB as our index. Chroma enables us to store documents and their embeddings, making them easily retrievable through its built-in semantic similarity search. In short, semantic similarity compares the embedding matrix of a search query to those stored in the database, identifying and returning the closest matches within the vector space. This approach allows us to retrieve the most relevant and contextually similar data chunks efficiently which we can then feed to the LLM as context.

In this example, we'll use the 2024 State of the Union transcript, which you can retrieve from the GitHub repository for this tutorial or save directly as a text file in your working directory. We'll read in the transcript and split the text into smaller chunks to enable more accurate retrieval.

```
# imports
from langchain_text_splitters import CharacterTextSplitter

# Read in State of the Union Address File
with open("2024_state_of_the_union.txt") as f:
    state_of_the_union = f.read()

# Initialize Text Splitter
text_splitter = CharacterTextSplitter(
    chunk_size=1000,
    chunk_overlap=200,
    length_function=len
)

# Create Documents (Chunks) From File
texts = text_splitter.create_documents([state_of_the_union])
```

With the transcript now split into smaller text chunks — formatted as LangChain documents — we can create the vector store index. First, we'll initialize the embeddings model function, ensuring that any document uploaded to the vector store is embedded and its embeddings are stored for efficient retrieval. Once again, for this we will be using OpenAI's `text-`

embedding-3-large model — for this to work, your OpenAI API key must be re


```
# imports
from langchain_chroma import Chroma
from langchain_openai import OpenAIEmbeddings

# Get Embeddings Model
embeddings = OpenAIEmbeddings(model="text-embedding-3-large")

# Initialize ChromaDB as Vector Store
vector_store = Chroma(
    collection_name="test_collection",
    embedding_function=embeddings
)
```


the documents uploaded into the vector store.

```
# Save Document Chunks to Vector Store
ids = vector_store.add_documents(texts)
```

Finally, to test it we can run the following code and manually check the results. They should include parts of the speech that specifically mentioned the invasion of Ukraine.

```
# Query the Vector Store
results = vector_store.similarity_search(
    'Who invaded Ukraine?',
    k=2
)

# Print Resulting Chunks
for res in results:
    print(f"* {res.page_content} [{res.metadata}]\n\n")
```

If the content returned makes sense, then you are good to move on to the next section — the actual naive RAG flow.

The Naive RAG Flow

Firstly, we will need to instantiate the remaining parts of the chain: the retriever and the LLM. For the LLM, we will be using OpenAI's `gpt-4o-mini` model, but any model will do.

```
# imports
from langchain_openai import ChatOpenAI

# Set Chroma Vector Store as the Retriever
retriever = vector_store.as_retriever()

# Initialize the LLM instance
llm = ChatOpenAI(model="gpt-4o-mini")
```

After this, we will need to create a function to format the retrieve content as a string to insert into the prompt. The below function grabs the documents' content and formats them with a space in between.

```
# Create Document Parsing Function to String
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)
```

Next, we create the prompt template. To show that the LLM is answering ba e ad prompt.


```
# imports
from langchain_core.prompts import PromptTemplate

# Create the Prompt Template
prompt_template = """Use the context provided to answer
the user's question below. If you do not know the answer
based on the context provided, tell the user that you do
not know the answer to their question based on the context
provided and that you are sorry.

context: {context}

question: {query}

answer: """

# Create Prompt Instance from template
custom_rag_prompt = PromptTemplate.from_template(prompt_template)
```

Finally, we create the RAG chain and can ask questions based on our individual documents:

```
# imports
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough

# Create the RAG Chain
rag_chain = (
    {"context": retriever | format_docs, "query": RunnablePassthrough()}
    | custom_rag_prompt
    | llm
    | StrOutputParser()
)

# Query the RAG Chain
rag_chain.invoke(
  "According to the 2024 state of the union address, Who invaded Ukraine?"
)
```

Here is the RAG chain response from the invoke above:

> *According to the 2024 State of the Union address, Putin of Russia invaded Ukraine.*

To test if it is only answering based on the content provided, we can run the following code as well:

```
# Get an I don't know from the Model
rag_chain.invoke("What is the purpose of life?")
```

Here is the response from the invoke above:

> *I'm sorry, but I do not know the answer to your question based on the context provided.*

This concludes our walkthrough of a basic RAG flow using Python and LangChain. In the next article, we'll dive into more advanced RAG techniques, exploring ways to optimize and scale the pipeline. If you're interested in following along with the series and staying updated on new releases, feel free to follow me here on Medium and connect on LinkedIn. Your support is appreciated, and I look forward to sharing more insights with you soon!

For additional help, here are some links to documentation that was used in creating this article:
https://python.langchain.com/docs/tutorials/rag/
https://python.langchain.com/docs/integrations/vectorstores/chroma/

https://python.langchain.com/v0.1/docs/modules/data_connection/document_l

My Personal Links:
GitHub: https://github.com/joeyoneill/
LinkedIn: https://www.linkedin.com/in/joseph-o-neill-131592164/
Portfolio: https://joeyoneill.neocities.org/

Langchain | Python | AI | Programming | Vector Database

Written by Joey O'Neill

149 followers · 3 following

Follow

Developer of Things | GenAI | Python Enthusiast | AI Consulting
joeyoneill.neocities.org https://www.linkedin.com/in/joseph-o-neill-131592164/

No responses yet

Write a response

What are your thoughts?

More from Joey O'Neill

Joey O'Neill

A Comprehensive Guide to LangGraph: Managing Agent Stat...

How to build custom tool nodes that handle both regular tool returns and Command...

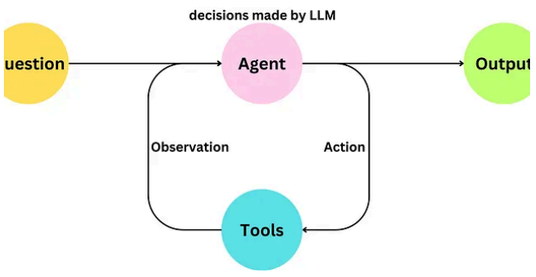
Aug 7 · 1

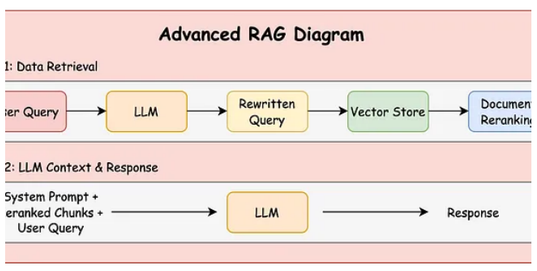
Joey O'Neill

Full-stack RAG: FastAPI Backend (Part 1)

All of my articles are 100% free to read. Non-members can read for free by clicking this...

Jan 28 · 133 · 1





Joey O'Neill

Agentic RAG with Python & LangGraph

All of my articles are 100% free to read. Non-members can read for free by clicking this...

Jan 21 · 217 · 2

Joey O'Neill

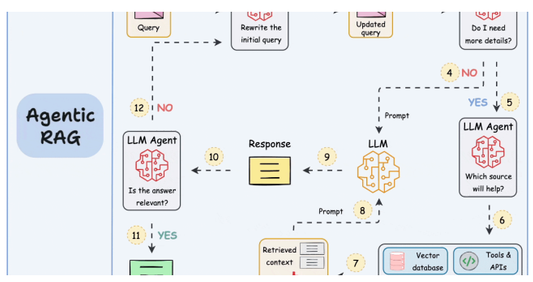
Advanced RAG with Python & LangChain

In this article, we incorporate pre-retrieval query rewriting and post-retrieval document...

Jan 14 · 61

See all from Joey O'Neill

Recommended from Medium

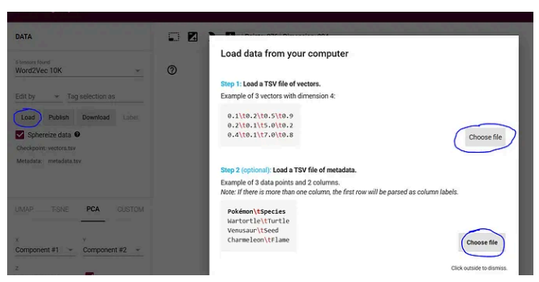


In Artificial Intelligence in Plain ... by Piyush Agni...

Building Agentic RAG with LangGraph: Mastering Adaptive...

Build intelligent RAG systems that know when to retrieve documents, search the web, or...

Jul 20 1.91K 27



Anil Goyal

Building a Simple RAG System with LangChain, FAISS, and Ollama...

In my blog https://medium.com/@anil.goyal0057/unloc...

May 16 7

In Towards AI by Harshit Kandoi

Building Smarter LLMs with LangChain and RAG: A Beginner'...

Retrieval-Augmented Generation (RAG), show you how LangChain fits into the puzzle...

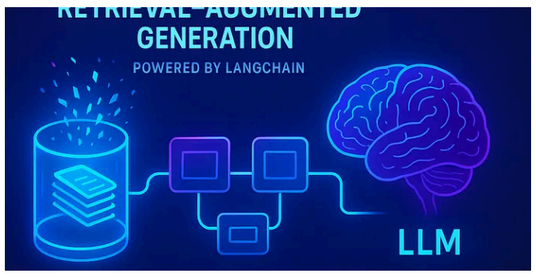
Apr 30 107 1

Bishal Bose

Building a Simple RAG System from Scratch: A Comprehensive...

Photo by Steve Johnson on Unsplash

Apr 25 677



Nikulsinh Rajput

LangChain for Real-World Retrieval-Augmented Generation

Building production-ready RAG pipelines with Python, LangChain, and vector databases.

Sep 1 14

In Data And Beyond by Shreyansh Jain

Building with Langchain — RAG Application

Retrieval is a technique where user can asked for a specific data that might not be present...

Mar 27 201

See more recommendations