Medium Search

Building a Knowledge base for custom LLMs using Langchain, Chroma, and GPT4All

Anindyadeep · Follow

14 min read · Jul 30, 2023

Listen Share

Hello everyone, Hope you are having an awesome weekend. In my previous blo learned about knowledge bases, embeddings, and how LLMs use knowledge bases for giving factually better and enhanced results. Now it is time to get our hands dirty and implement the same using langchain.

The plot

But before making starting with our recipe, we have to think of our plot. By plot, I mean what we are going to build at the end of the day. Well, this whole series of blogs is dedicated to making a simple application named **DocChat.** The concept is very simple, an LLM-powered app where we can upload and chat with our document.

Simple right? Well no. I mean yes, it is simple when you just use a few lines of code and let various services like HuggingFace serve or other services handle the core backend. But when you are at a company building capabilities, then you can never use those 10 lines of code. Then that time you have to build most of the things from the start and may use some part of the code. And hence comes my blog series. I am treating this LLM app like another end-to-end ML application with things like

- serving using API

- connecting with other backend workflows
- making things serverless and turning them to cloud-native LLM-powered apps etc.

The journey might not be perfect. However, I am sure that this will help me and you guys to learn and deep dive more into building real-world ML applications.

What is covered so far in the series

- Integrating custom LLM using langchain (A GPT4ALL example)
- Configuration management for LLM-powered applications
- Knowledge Bases and retrieval augmented LLMs, A primer.
- Connecting LLM to the knowledge base
- Serving LLM using Fast API (coming soon)
- Fine-tuning an LLM using transformers and integrating it into the existing pipeline for domain-specific use cases (coming soon)
- Putting all together into a simple make cloud-native LLM-powered application (coming soon)

Hence in this blog, we are covering the topic, Connecting LLMs to the knowledge bases. Just for the newcomers here, who directly hopped into this blog, we have covered these things so far.

- We built our custom gpt4all-powered LLM with custom functions wrapped around the langchain.
- We then discussed how to structure a project and maintain current standards using cookie clutter and how to practice configuration management using Hydra to manage complex app configurations

Today on top of these two, we will add a few lines of code, to support the functionalities of adding docs and injecting those docs to our vector database

(Chroma becomes our choice here) and connecting it to our LLM. We use gpt4all embeddings to get embed the text for a query search.

Let's get started

This blog is going to be very simple. For starters, we have to select a folder where we will dump all the documents. In my case, I have a folder called `source_documents/` where I will dump all my PDFs.

Then we will first set some configurations for chroma db, which will be our vector db eventually. Do not worry, for folks who are not familiar with hydra can check out my previous blog. But here I am not using Hydra for setting up the settings. You can see that in my GitHub repo implementation.

```
# specify the directory where Chroma will build the vector db
CHROMA_DB_DIRECTORY='db'

# specify where the source documents are
DOCUMENT_SOURCE_DIRECTORY='/path/to/source/documents'

# settings include things like which database backend chroma will use
# here we will be using duck db
# and document will be stored under db
# with no telemetry (i.e. nothing will be tracked)

CHROMA_SETTINGS = Settings(
    chroma_db_impl='duckdb+parquet',
    persist_directory=CHROMA_DB_DIRECTORY,
    anonymized_telemetry=False
)

# target number of relevant chunks to return
TARGET_SOURCE_CHUNKS=4

# the number of characters that will make up a chunk
CHUNK_SIZE=500

# the number of overlapping characters to maintain the chunk
# continuity
CHUNK_OVERLAP=50

# whether to show or hide specific documents while LLM giving response
# i.e. whether we need to show the document sources that our LLM
```

```
# referred while giving the answer

HIDE_SOURCE_DOCUMENTS=False
```

These settings are very important and can be changed based on the LLM we choose (some LLM supports better huge context length and in that case, we can increase the chunk size to 1000 or more and the top-n chunks are set to 4, we can add more, but then retrieval might take more time)

Now let's import the necessary imports

```
import os
from typing import Optional
from chromadb.config import Settings
from langchain.vectorstores import Chroma
from langchain.document_loaders import DirectoryLoader
from langchain.embeddings import GPT4AllEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

I hope we do not need much explanation of what is getting imported. It is pretty intuitive. We are loading our Chroma vector db python client wrapped around langchain, Langchain's directory loader, which loads all the documents when we pass the directory, GPT4AllEmebeddings by Nomic AI's GPT4All and RecursiveCharacterTextSplitter, which will be responsible to create our document chunks before giving it to our embeddings.

Please NOTE: There is something that is needed to be installed (if in Linux) before using the Directory loader. Because Langchain's `DirectoryLoader` loads any kind of document like `.pdf/.txt/.ppt` etc, and hence it requires some additional dependencies to get installed.

Install Linux packages

```
# Update package lists
sudo apt update

# Install tesseract-ocr and libtesseract-dev
sudo apt install tesseract-ocr libtesseract-dev

# Install more dependencies
sudo apt-get install \
    libleptonica-dev \
    tesseract-ocr-dev \
    python3-pil \
    tesseract-ocr-eng

# Install the python depedencies

pip install pytesseract
```

Now let's build a simple call `MyKnowledgeBase` where we will be adding these methods

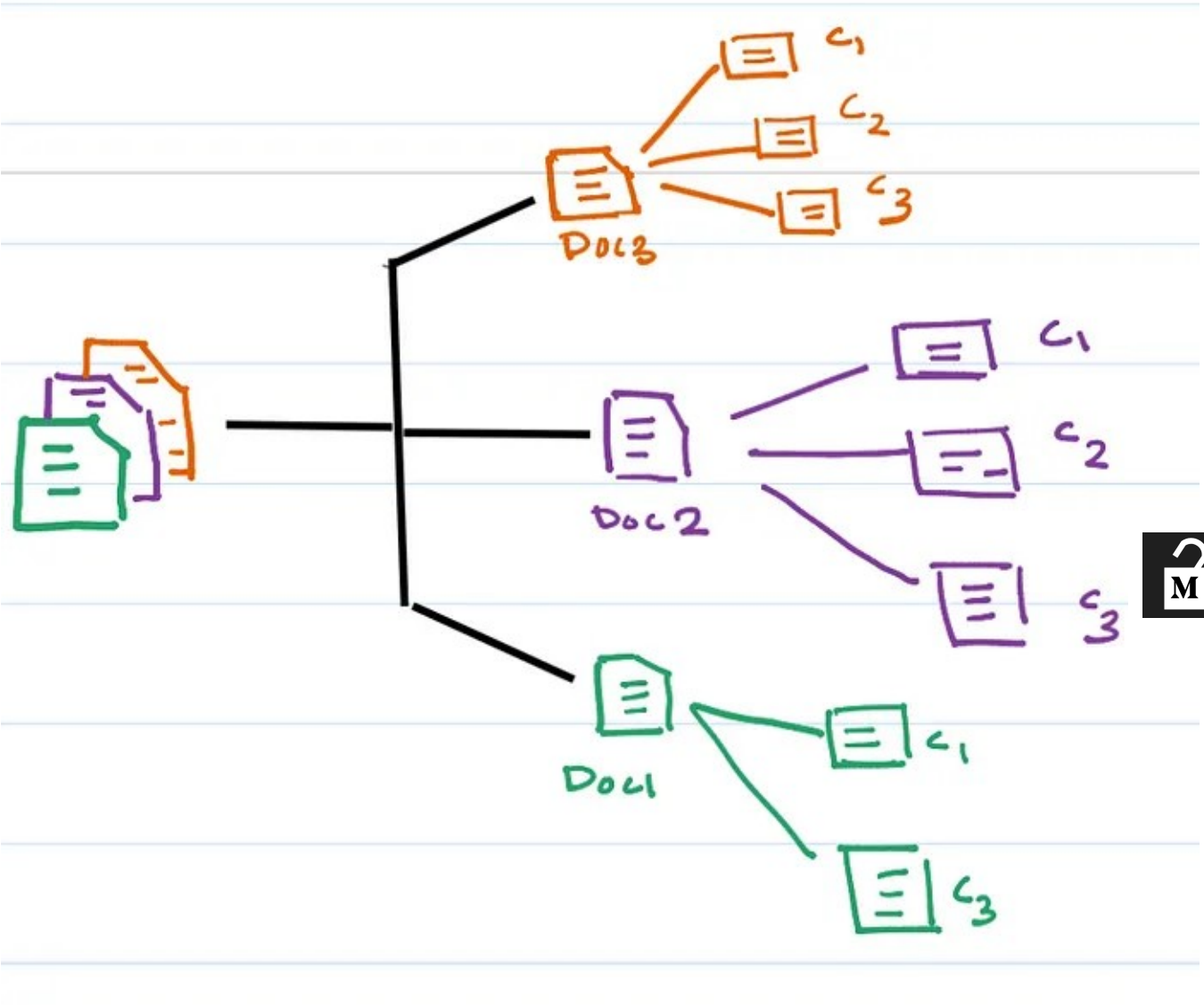
```
class PDFKnowledgeBase:
    def __init__(self, pdf_source_folder_path: str) -> None:
        """
        Loads pdf and creates a Knowledge base using the Chroma
        vector DB.
        Args:
            pdf_source_folder_path (str): The source folder containing
            all the pdf documents
        """
        self.pdf_source_folder_path = pdf_source_folder_path

    def load_pdfs(self):
        # method to load all the pdf's inside the directory
        # using DirectoryLoader
        pass

    def split_documents(self, loaded_docs, chunk_size=1000):
        # split the documents into chunks and return the
        # chunked documents
        pass
```

```
    def convert_document_to_embeddings(
        self, chunked_docs, embedder
    ):
        # convert the chunked docs to embeddings and add that
        # to our vector db
        pass

    def return_retriever_from_persistant_vector_db(
        self, embedding_function
    ):
        # return a retriever object which will retrieve the
        # relevant chunks
        pass
```

Now let's start writing each function one by one. Starting with our `load_pdf` functions. Well, all we have to do is instantiate the `DirectoryLoader` class and provide the source document folders inside the constructor.

```
def load_pdfs(self):
    # instantiate the DirectoryLoader class
    # load the pdfs using loader.load() function

    loader = DirectoryLoader(
        self.pdf_source_folder_path
    )
    loaded_pdfs = loader.load()
    return loaded_pdfs
```

Now we have our `split_documents` function. This will split our documents into the some number of chunks where each chunk will have a size of characters.

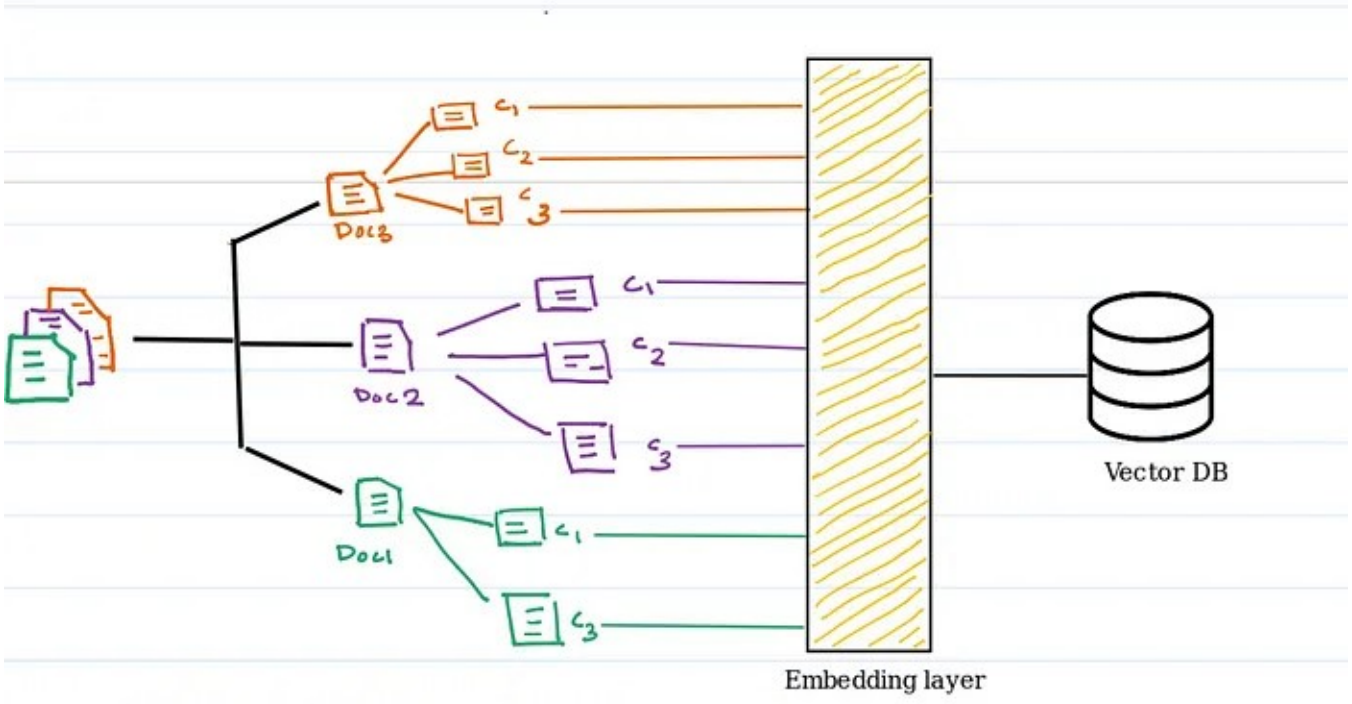


Here we have set that in our settings as 500, also we have to fill in an additional parameter called `chunk_overlap` that helps to ensure that adjacent chunks share some common characters, **preventing** potential **information loss** at the boundary between the chunks. Here is the sample code on how to split all the loaded documents into chunks

```
def split_documents(
    self,
    loaded_docs,
    chunk_size: Optional[int] = 500,
    chunk_overlap: Optional[int] = 20,
```

```
):
    # instantiate the RecursiveCharacterTextSplitter class
    # by providing the chunk_size and chunk_overlap

    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
    )

    # Now split the documents into chunks and return
    chunked_docs = splitter.split_documents(loaded_docs)
    return chunked_docs
```

Here comes the fun part. Now it is time to create embeddings for our chunked documents and register those embeddings into our knowledge base, which will be our vector database (here Chroma)




```
def convert_document_to_embeddings(
    self, chunked_docs, embedder
):
    # instantiate the Chroma db python client
    # embedder will be our embedding function that will map our chunked
```

```
    # documents to embeddings

    vector_db = Chroma(
        persist_directory=CHROMA_DB_DIRECTORY,
        embedding_function=embedder,
        client_settings=CHROMA_SETTINGS,
    )

    # now once instantiated we tell our db to inject the chunks
    # and save all inside the db directory
    vector_db.add_documents(chunked_docs)
    vector_db.persist()

    # finally return the vector db client object
    return vector_db
```

Now our last function is left, which is essentially to provide a retriever object wh has only one object, given my `query` I can simply retrieve the relevant documen chunks from a vector similarity search by our provided vector database.

```
def return_retriever_from_persistant_vector_db(
    self, embedder
):
    # first check whether the database is created or not
    # if not then throw error
    # because if the database is not instantiated then
    # we can not get the retriever

    if not os.path.isdir(CHROMA_DB_DIRECTORY):
        raise NotADirectoryError(
            "Please load your vector database first."
        )

    vector_db = Chroma(
        persist_directory=CHROMA_DB_DIRECTORY,
        embedding_function=embedder,
        client_settings=CHROMA_SETTINGS,
    )

    # used the returned embedding function to provide the retriver object
    # with number of relevant chunks to return will be = 4
    # based on the one we set inside our settings
```

```
    return vector_db.as_retriever(
        search_kwargs={"k": TARGET_SOURCE_CHUNKS}
    )
```

Note, this is just a simple implementation, there could be different checks and edge cases before providing the retriever. The goal is to learn that we need to be aware of those scenarios and how we can write better object-oriented programs so that it is loosely coupled and highly modular i.e. can be used in other programs very easily. Awesome now let's club all the functions together so that during the time of calling these functions we can call all of them at once and just call the retriever.

Putting everything together

You did an awesome job till here 💪, Now it's time to put everything all together see how it got turned up.

```
import os
from typing import Optional

from chromadb.config import Settings
from langchain.vectorstores import Chroma
from langchain.document_loaders import DirectoryLoader
from langchain.embeddings import GPT4AllEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter


CHROMA_DB_DIRECTORY='db'
DOCUMENT_SOURCE_DIRECTORY='/path/to/source/documents'
CHROMA_SETTINGS = Settings(
    chroma_db_impl='duckdb+parquet',
    persist_directory=CHROMA_DB_DIRECTORY,
    anonymized_telemetry=False
)
TARGET_SOURCE_CHUNKS=4
CHUNK_SIZE=500
CHUNK_OVERLAP=50
HIDE_SOURCE_DOCUMENTS=False

class MyKnowledgeBase:
    def __init__(self, pdf_source_folder_path: str) -> None:
        """
```

```
        Loads pdf and creates a Knowledge base using the Chroma
        vector DB.
        Args:
            pdf_source_folder_path (str): The source folder containing
            all the pdf documents
        """
        self.pdf_source_folder_path = pdf_source_folder_path

    def load_pdfs(self):
        loader = DirectoryLoader(
            self.pdf_source_folder_path
        )
        loaded_pdfs = loader.load()
        return loaded_pdfs

    def split_documents(
        self,
        loaded_docs,
    ):
        splitter = RecursiveCharacterTextSplitter(
            chunk_size=CHUNK_SIZE,
            chunk_overlap=CHUNK_OVERLAP,
        )
        chunked_docs = splitter.split_documents(loaded_docs)
        return chunked_docs

    def convert_document_to_embeddings(
        self, chunked_docs, embedder
    ):
        vector_db = Chroma(
            persist_directory=CHROMA_DB_DIRECTORY,
            embedding_function=embedder,
            client_settings=CHROMA_SETTINGS,
        )

        vector_db.add_documents(chunked_docs)
        vector_db.persist()
        return vector_db

    def return_retriever_from_persistant_vector_db(
        self, embedder
    ):
        if not os.path.isdir(CHROMA_DB_DIRECTORY):
            raise NotADirectoryError(
                "Please load your vector database first."
            )

        vector_db = Chroma(
```

```
            persist_directory=CHROMA_DB_DIRECTORY,
            embedding_function=embedder,
            client_settings=CHROMA_SETTINGS,
        )

        return vector_db.as_retriever(
            search_kwargs={"k": TARGET_SOURCE_CHUNKS}
        )

    def initiate_document_injetion_pipeline(self):
        loaded_pdfs = self.load_pdfs()
        chunked_documents = self.split_documents(loaded_docs=loaded_pdfs)

        print("=> PDF loading and chunking done.")

        embeddings = GPT4AllEmbeddings()
        vector_db = self.convert_document_to_embeddings(
            chunked_docs=chunked_documents, embedder=embeddings
        )

        print("=> vector db initialised and created.")
        print("All done")
```

Great job 🔥. Cool, now let's assume we have a file named called `knowledgebase.py` where all these codes are written. Inside that file only let's use this function first to load and create our database. Because in our main file, we will just load the DB and will attach it with our LLM. We do not want to build the documents every time we run our main file. Hence the best practices here would be to either run the `knowledgebase.py` file then and there inside a `if __name__ == '__main__'` or import the `MyKnowledgeBase` module to any other file to call it separately (maybe when you are frequently adding files)

Now then we have two more things left.

1. First using our awesome class to make our knowledge base
2. Inside our main folder where our LLM chat resides, we will attach our retriever from our vector db to make a retrieval chain using Langchain to complete our full pipeline.

Document ingestion using our knowledgebase module

Let's assume we have a file called `injestion.py` which will be responsible for document ingestion. Here is a sample code of how we import our knowledge base module and do our document ingestion followed by the vector db instantiation task.

```
from knowledgebase import MyKnowledgeBase
from knowledgebase import (
    DOCUMENT_SOURCE_DIRECTORY
)

# kb is here knowledge base
kb = MyKnowledgeBase(
        pdf_source_folder_path=DOCUMENT_SOURCE_DIRECTORY
)

kb.initiate_document_injetion_pipeline()
```

And once done our directory will be having showing something like this

```
├── db
│   ├── chroma-collections.parquet
│   ├── chroma-embeddings.parquet
│   └── index
│       ├── id_to_uuid_bb6c59a0-db4d-4bcf-bc0d-e8c4fc78ee34.pkl
│       ├── index_bb6c59a0-db4d-4bcf-bc0d-e8c4fc78ee34.bin
│       ├── index_metadata_bb6c59a0-db4d-4bcf-bc0d-e8c4fc78ee34.pkl
│       └── uuid_to_id_bb6c59a0-db4d-4bcf-bc0d-e8c4fc78ee34.pkl
└── source_documents
    ├── diff_lm.pdf
    └── llama2.pdf
```

Inside `source_documents` , I kept two research paper PDFs, and after invoking our ingestion pipeline this new folder `db` get's created which becomes our vector db.

Using Langchain to connect our vector db and LLM

Now comes the fun part. Here we will be connecting our vector database with our LLM, here we are using our custom LLM class wrapped around Langchain and we

are using gpt4all for our LLM provider. You can check this link out to see how we have done that.

Start with importing our libraries and module

```
# import our MyGPT4ALL class from mode module
# import MyKnowledgeBase class from our knowledgebase module

from model import MyGPT4ALL
from knowledgebase import MyKnowledgeBase
from knowledgebase import (
    DOCUMENT_SOURCE_DIRECTORY
)

# import all the langchain modules
from langchain.chains import RetrievalQA
from langchain.embeddings import GPT4AllEmbeddings
```

Please note: Before going further, I want to make you guys aware that some parts of the incoming code are heavily dependent on my previous blogs. So either you can check out that blog to see how things are happening or you can do another just see how things are done and instead of gpt4all, you can replace your LLM provider of choice, like Open AI or HuggingFace, etc.

Let's start by setting up some variables

```
GPT4ALL_MODEL_NAME='ggml-gpt4all-j-v1.3-groovy.bin'
GPT4ALL_MODEL_FOLDER_PATH='/home/anindya/.local/share/nomic.ai/GPT4All/'
GPT4ALL_BACKEND='llama'
GPT4ALL_ALLOW_STREAMING=True
GPT4ALL_ALLOW_DOWNLOAD=False
```

This configuration tells us where our model path is present, what will be the backend, whether to stream results or not, etc. Now let's start with defining our

custom LLM model.

```
llm = MyGPT4ALL(
    model_folder_path=GPT4ALL_MODEL_FOLDER_PATH,
    model_name=GPT4ALL_MODEL_NAME,
    allow_streaming=True,
    allow_download=False
)
```

Instead of `MyGPT4ALL`, just replace the LLM provider of your choice. Now let's define our knowledge base. Here we are doing a strong assumption that we are calling our knowledge base along with our retriever after we have done the ingestion process otherwise it will throw an error.

```
embeddings = GPT4AllEmbeddings()

kb = MyKnowledgeBase(
    pdf_source_folder_path=DOCUMENT_SOURCE_DIRECTORY
)

# get the retriver object from the vector db

retriever = kb.return_retriever_from_persistant_vector_db()
```

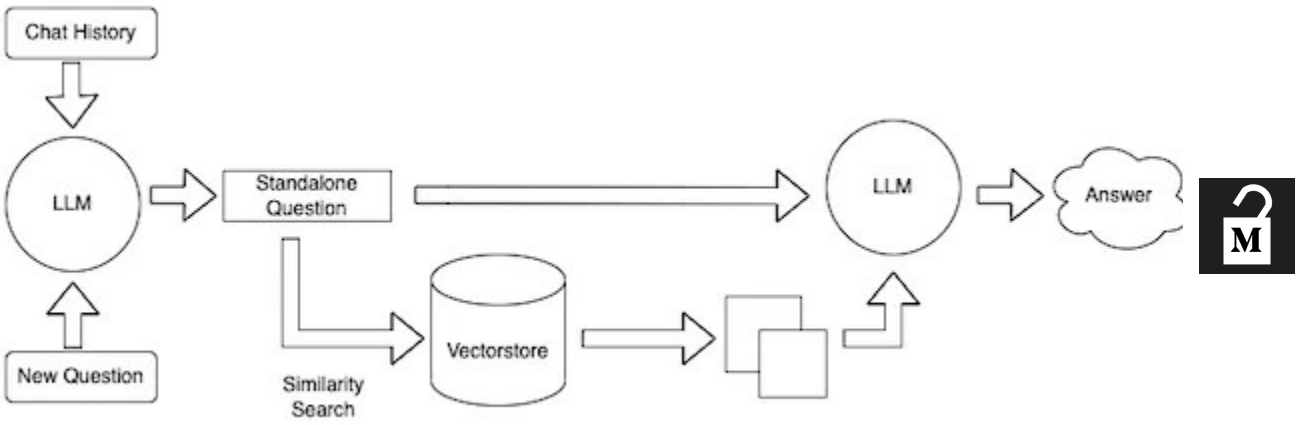
So, now we have the retriever, we have our LLM, how we can connect those two? Here comes our **RetrievalQA** by Langchain, which connects our LLM to the retriever (or we can say chain them), and hence we can do all everything with just a function call.

Here is the sample example

```
qa_chain = RetrievalQA.from_chain_type(
    llm = llm,
    chain_type='stuff',
```

```
        retriever=retriever,
        return_source_documents=True, verbose=True
    )
```

Here `chain_type='stuff'` means that whatever gets retrieved from the chain, we will stuff them (or say stack upon each other) to build the final prompt to provide to the LLM. Here is an awesome diagram showing how the chain works



https://blog.langchain.dev/retrieval/

Now then we have everything set up, it's time to roll the engine. We will create a simple infinite loop, where we will take the input from the user if the input is `exit` then we will come out of the loop or we will start our `qa_chain` to get the results.

Results come as a tuple of two things (answer, relevant_docs). As we have set `return_source_documents=True` , hence we length of relevant_docs will not be 0 other wise 0. And as it is not 0, so we traverse through each doc and show after the answers. This tells us the relevant documents which were retrieved that provided context to our LLM.

Here is a sample code for this

```
# main.py file
```

```
while True:
    query = input("What's on your mind: ")
    if query == 'exit':
        break
    result = qa_chain(query)
    answer, docs = result['result'], result['source_documents']

    print(answer)

    print("#"* 30, "Sources", "#"* 30)
    for document in docs:
        print("\n> SOURCE: " + document.metadata["source"] + ":")
        print(document.page_content)
    print("#"* 30, "Sources", "#"* 30)
```

So now when I do `python3 main.py`

It gives this awesome output

```
What's on your mind: What is difference between Llama2 and Llama1


> Entering new RetrievalQA chain...
The main differences between LLAma1 (Lila) and LLaMa2 are in their architecture

* Architecture: The primary difference is that LLaMa2 has an additional layer f
* Training data: LLaMa2 has access to a larger training dataset compared to its
* Performance: LLaMa2 has shown significant improvements compared to Lila when
> Finished chain.
The main differences between LLAma1 (Lila) and LLaMa2 are in their architecture

* Architecture: The primary difference is that LLaMa2 has an additional layer f
* Training data: LLaMa2 has access to a larger training dataset compared to its
* Performance: LLaMa2 has shown significant improvements compared to Lila when

############################## Sources ##############################

> SOURCE: /home/anindya/workspace/repos/end-to-end-llm/source_documents/llama2.
Liama 2 is a new technology that carries risks with use. Testing conducted to d

Table 52: Model card for LLAMA 2.
```

That's dope right 😎😎. Whooahh, Congratulations on coming this far. This part of the blog post intentionally does not cover the config management side of the code and is hence simplfied. To see how we do the same but with best practices, check out the GitHub repository.

So, in this blog, you have learned how we can use Langchain, Chroma, and GPT4All to build in-house LLM capabilities without using Open AI-like API.

In the next series, we are going to cover, how we can now make APIs and serve the applications. Hence please tighten your seat belts, because from here the curve is gonna be steep. Until next time.

References

- Knowledge Bases and retrieval augmented LLMs, A primer.
- How to integrate custom LLM using langchain. A GPT4ALL example.
- Retrieval by LangChain AI

Llm | Llmops | Machine Learning | Transformers | Deep Learning

Follow

Written by Anindyadeep

299 Followers

Engineering @PremAI | Ex ML @CorridorPlatforms, @Voxela Inc. I like to talk about my journey and learnings.

More from Anindyadeep

Anindyadeep

How to integrate custom LLM using langchain. A GPT4ALL example.

This is part 1 of my mini-series: Building end to end LLM powered applications without Open AI's API

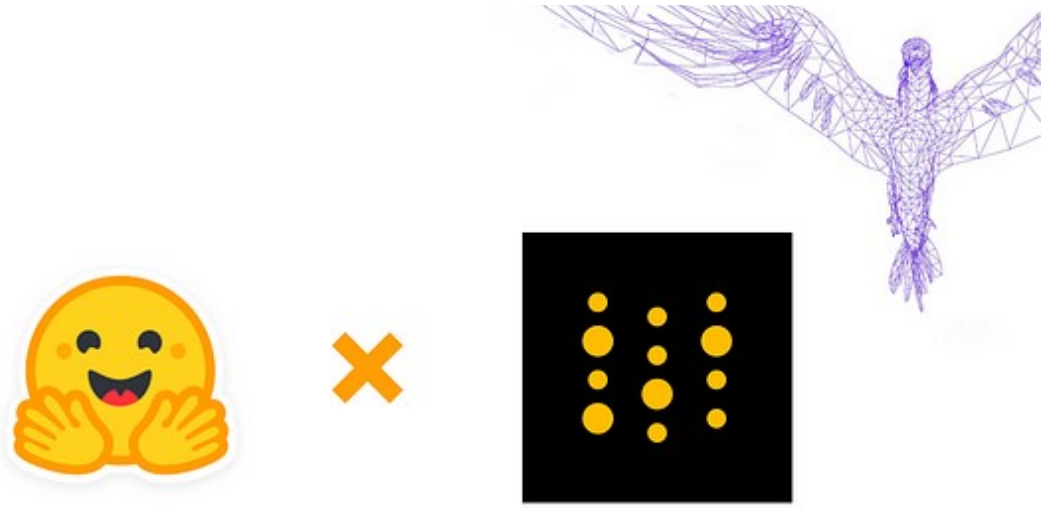
Jul 16, 2023 286 5

Anindyadeep

Structuring projects and configuration management for LLM-powered Apps.

You can say that this is part 1.5 😅 of my mini-series Building end-to-end LLM-powered applications without Open AI's API.

Jul 23, 2023 92 1



Anindyadeep

Practitioners guide to fine-tune LLMs for domain-specific use case

with Aditya Khandekar CorridorPlatforms

Aug 4, 2023 285 3

Anindyadeep

Knowledge Bases and retrieval augmented LLMs, A primer.

So far we have learned how to connect custom Large Language Models using Langchain and there I showed an example of connecting a LLM...

Jul 30, 2023 184 2

See all from Anindyadeep

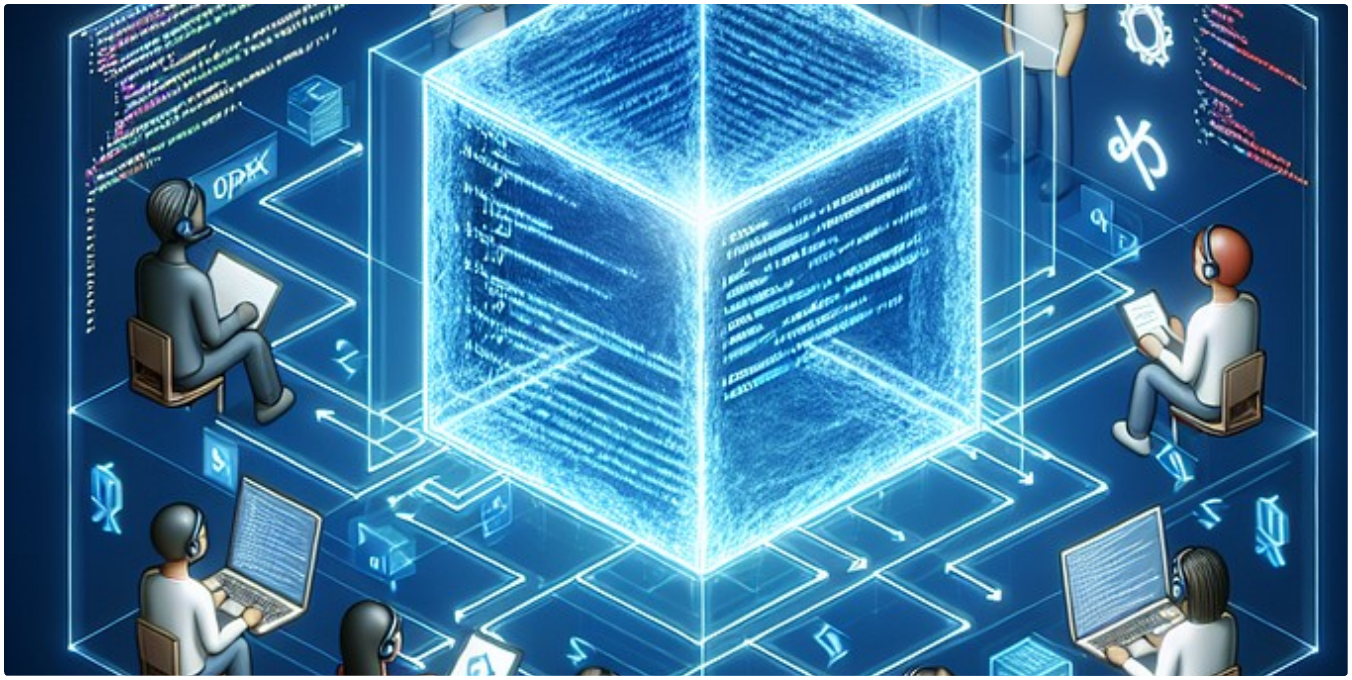
Recommended from Medium

Devvrat Rana

Understanding LangChain Agents: A Beginner's Guide to How LangChain Agents Work

Introduction:

Jun 2 66



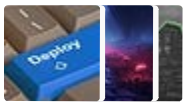
Subhrajit Mohanty

How to Build an OpenAI-Compatible API using Any FastAPI Application with LLM Integration: A...

OpenAI Wrapper is a versatile tool that simplifies interactions with OpenAI's API. It streamlines the integration process, allowing...

Jun 19 96

Lists

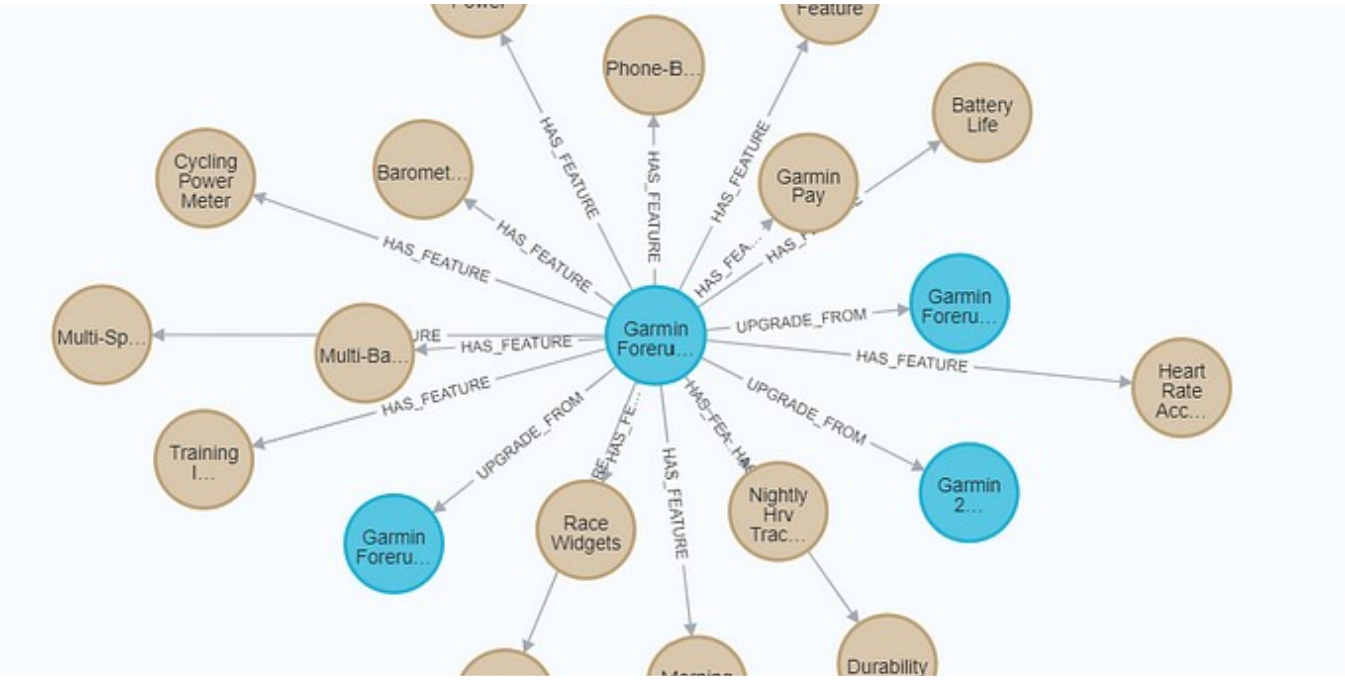


Predictive Modeling w/ Python
20 stories · 1614 saves

Practical Guides to Machine Learning
10 stories · 1968 saves

Natural Language Processing
1772 stories · 1374 saves

data science and AI
40 stories · 271 saves

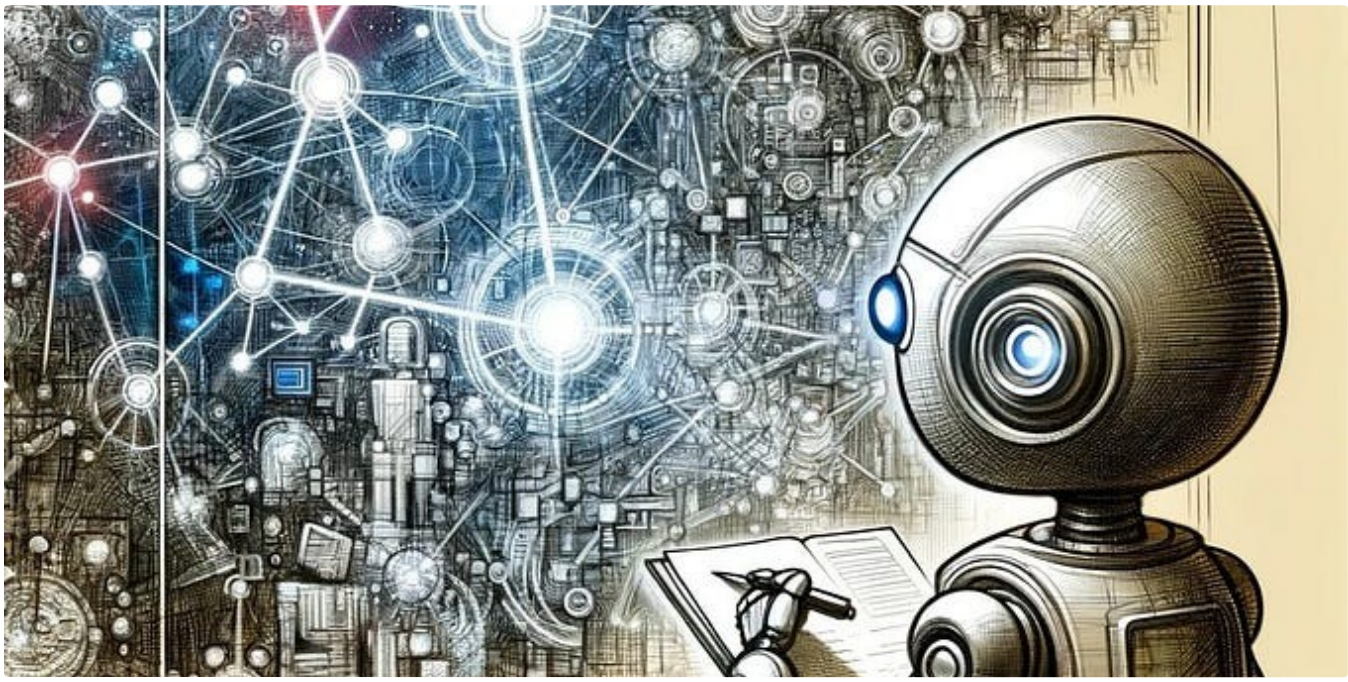


Mark OBrien

Building Knowledge Graphs with LangChain & Neo4J

Knowledge Graphs

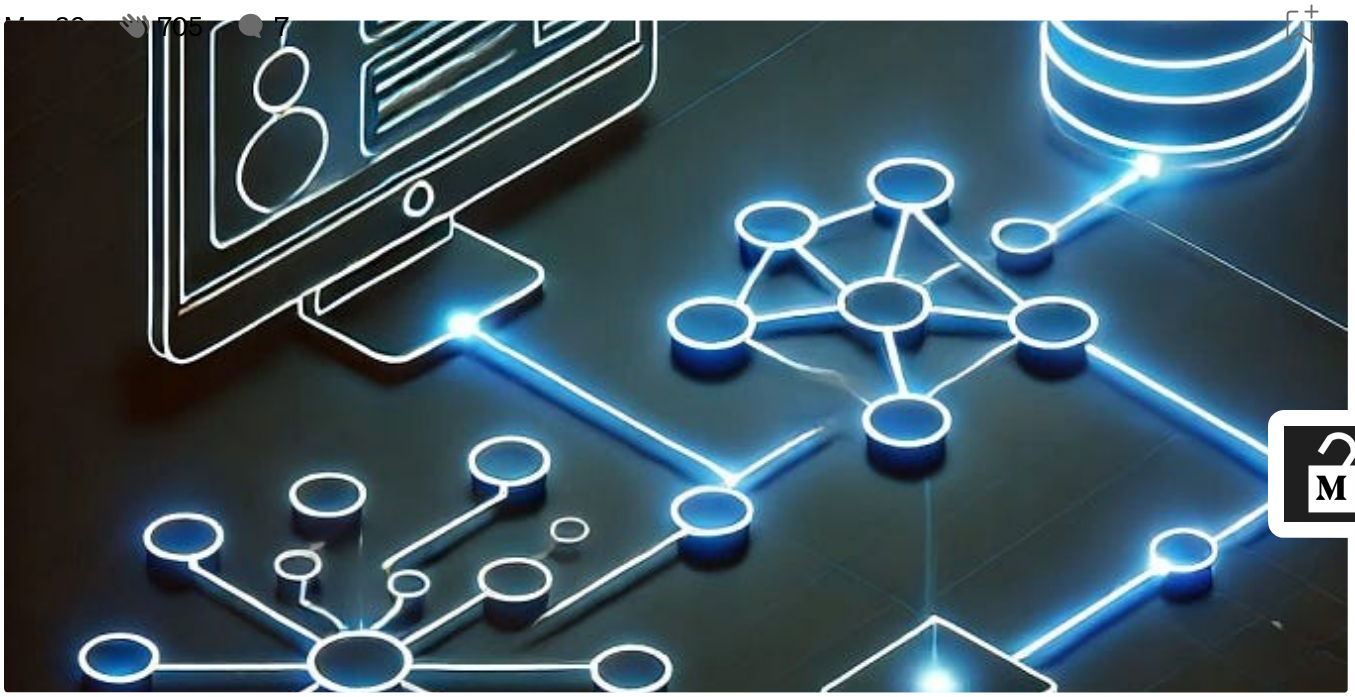
Jun 13 38



Tomaz Bratanic in Neo4j Developer Blog

Enhancing the Accuracy of RAG Applications With Knowledge Graphs

A practical guide to constructing and retrieving information from knowledge graphs in RAG applications with Neo4j and LangChain



Omotolani Kehinde

Building an Advanced RAG Chatbot with Knowledge Graphs Using Llamaindex, Neo4j, and Llama 3.

A step by step guide on how to build an advanced Retrieval-Augmented Generation (RAG) chatbot by integrating knowledge graphs.

Jun 17 166 1

Rahatara Ferdousi

Build and Deploy A Custom LLM Chatbot in 5 Minutes

Ever wanted to create a custom chatbot with LLM?

Jun 24 14

See more recommendations