**Building a Multi-Document Reader and Chatbot With LangChain and ChatGPT**

The best part? The chatbot will remember your chat history

[[](https://medium.com/@ssmaameri?source=post_page-----d1864d47e339--------------------------------)](https://medium.com/@ssmaameri?source=post_page-----d1864d47e339--------------------------------)

[[Better Programming](https://betterprogramming.pub/?source=post_page-----d1864d47e339--------------------------------)](https://betterprogramming.pub/?source=post_page-----d1864d47e339--------------------------------)

[Sami Maameri](https://medium.com/@ssmaameri?source=post_page-----d1864d47e339--------------------------------)

·

Follow

Published in

[Better Programming](https://betterprogramming.pub/?source=post_page-----d1864d47e339--------------------------------)

·

17 min read

·

May 20

1.2K

11

Many AI products are coming out these days that allow you to interact with your own private PDFs and documents. But how do they work? And how do you build one? Behind the scenes, it’s actually pretty easy.

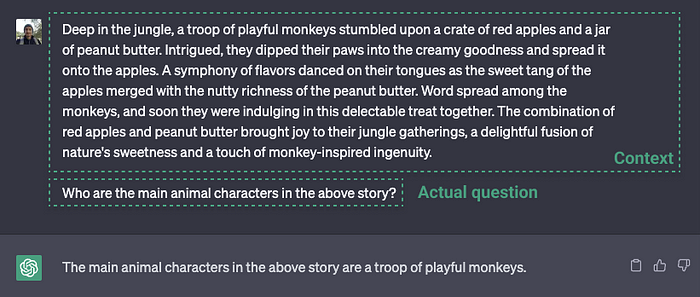
Let’s dive in!

We’ll start with a simple chatbot that can interact with just one document and finish up with a more advanced chatbot that can interact with multiple different documents and document types, as well as maintain a record of the chat history, so you can ask it things in the context of recent conversations.

**Contents**  
  
[How Does It Work?](https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339#e39f)  
  
[Interacting With a Single PDF](https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339#8b1b)  
  
[Interacting With a Single PDF Using Embeddings and Vector Stores](https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339#5f0b)  
  
[Adding Chat History](https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339#fafe)  
  
[Interacting With Multiple Documents](https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339#305d)  
  
[Improvements](https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339#8a4c)  
  
[Summary](https://betterprogramming.pub/building-a-multi-document-reader-and-chatbot-with-langchain-and-chatgpt-d1864d47e339#3538)

**How Does It Work?**

At a basic level, how does a document chatbot work? At its core, it’s just the same as ChatGPT. On ChatGPT, you can copy a bunch of text into the prompt, and then ask ChatGPT to summarise the text for you or generate some answers based on the text.



Example of passing in some context and a question to ChatGPT

Interacting with a single document, such as a PDF, Microsoft Word, or text file, works similarly. We extract all of the text from the document, pass it into an LLM prompt, such as ChatGPT, and then ask questions about the text. This is the same way the ChatGPT example above works.

**Interacting with multiple documents**

Where it gets a little more interesting is when the document is very large, or there are many documents we want to interact with. Passing in all of the information from these documents into a request to an LLM (Large Language Model) is impossible since these requests usually have size (token) limits, and so would only succeed if we tried to pass in too much information.

We can only send the relevant information to the LLM prompt to overcome this. But how do we get only the relevant information from our documents? This is where embeddings and vector stores come in.

**Embeddings and Vector Stores**

We want a way to send only relevant bits of information from our documents to the LLM prompt. Embeddings and vector stores can help us with this.

Embeddings are probably a little confusing if you have not heard of them before, so don’t worry if they seem a little foreign at first. A bit of explanation, and using them as part of our setup, should help make their use a little more clear.

An embedding allows us to organise and categorise a text based on its semantic meaning. So we split our documents into lots of little text chunks and use embeddings to characterise each bit of text by its semantic meaning. An embedding transformer is used to convert a bit of text into an embedding.

An embedding categorises a piece of text by giving it a vector (coordinate) representation. That means that vectors (coordinates) that are close to each other represent pieces of information that have a similar meaning to each other. The embedding vectors are stored inside a vector store, along with the chunks of text corresponding to each embedding.

Once we have a prompt, we can use the embeddings transformer to match it with the bits of text that are most semantically relevance to it, so we know how a way to match our prompt with other related bits of text from the vector store. In our case, we use the OpenAI embeddings transformer, which employs the cosine similarity method to calculate the similarity between documents and a question.

Now that we have a smaller subset of the information which is relevant to our prompt, we can query the LLM with our initial prompt, while passing in only the relevant information as the context to our prompt.

This is what allows us to overcome the size limitation on LLM prompts. We use embeddings and a vector store to pass in only the relevant information related to our query and let it get back to us based on that.

So, how do we do this in LangChain? Fortunately, LangChain provides this functionality out of the box, and with a few short method calls, we are good to go. Let’s get started!

**Coding Time!**



Photo by [Radek Grzybowski](https://unsplash.com/@rgrzybowski?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

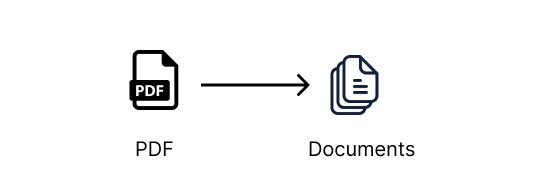
**Interacting with a single pdf**

Let’s start with processing a single pdf, and we will move on to processing multiple documents later on.

The first step is to create a Document from the pdf. A Document is the base class in LangChain, which chains use to interact with information. If we look at the class definition of a Document, it is a very simple class, just with a page\_content method, that allows us to access the text content of the Document.

class Document(BaseModel):  
 """Interface for interacting with a document."""  
 page\_content: str  
 metadata: dict = Field(default\_factory=dict)

We use the [DocumentLoaders](https://python.langchain.com/en/latest/modules/indexes/document_loaders.html) that LangChain provides to convert a content source into a list of Documents, with one Document per page.



For example, there are DocumentLoaders that can be used to convert pdfs, word docs, text files, CSVs, Reddit, Twitter, Discord sources, and much more, into a list of Document's which the LangChain chains are then able to work. Those are some cool sources, so lots to play around with once you have these basics set up.

First, let’s create a directory for our project. You can create all this as we go along or clone the GitHub repository with all the examples and sample docs using the below command. If you do clone the repository, make sure to follow the instructions in the README.md file to set up your OpenAI API key correctly.

git clone git@github.com:smaameri/multi-doc-chatbot.git

Otherwise, if you want to follow along step by step:

mkdir multi-doc-chatbot  
cd multi-doc-chatbot  
touch single-doc.py  
mkdir docs  
# lets create a virtual environement also to install all packages locally only  
python3 -m venv .venv  
. .venv/bin/activate

Then download the sample CV [RachelGreenCV.pdf from here](https://github.com/smaameri/multi-doc-chatbot/blob/master/docs/RachelGreenCV.pdf), and store it in the docs folder.

Let's install all the packages we will need for our setup:

pip install langchain pypdf openai chromadb tiktoken docx2txt

Now that our project folders are set up, let’s convert our PDF into a document. We will use the PyPDFLoader class. Also, let's set up our OpenAI API key now. We will need it later.

import os  
from langchain.document\_loaders import PyPDFLoader  
  
os.environ["OPENAI\_API\_KEY"] = "sk-"  
  
pdf\_loader = PyPDFLoader('./docs/RachelGreenCV.pdf')  
documents = pdf\_loader.load()

This returns a list of Document’s, one Document for each page of the pdf. In terms of Python types, it will return a List[Document]. So the index of the list will correspond to the page of the document, e.g., documents[0] for the first page, documents[1] for the second page, and so on.

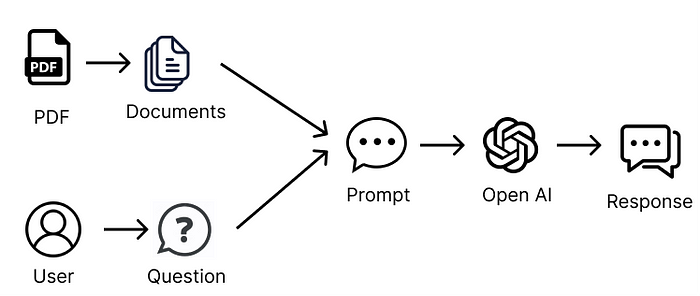
The simplest Q&A chain implementation we can use is the load\_qa\_chain. It loads a chain that allows you to pass in all of the documents you would like to query against.

from langchain.llms import OpenAI  
from langchain.chains.question\_answering import load\_qa\_chain  
  
# we are specifying that OpenAI is the LLM that we want to use in our chain  
chain = load\_qa\_chain(llm=OpenAI())  
query = 'Who is the CV about?'  
response = chain.run(input\_documents=documents, question=query)  
print(response)

Now, run this script to get the response:

➜ multi-doc-chatbot: python3 single-doc.py  
 The CV is about Rachel Green.

What is actually happening in the background here is that the document’s text (i.e., the PDF text) is being sent to the OpenAPI Chat API, along with the query, all in a single request.



passing all of the text from our source documents into the LLM prompt

Also, the load\_qa\_chain actually wraps the entire prompt in some text, instructing the LLM to use only the information from the provided context. So the prompt being sent to OpenAI looks something like the following:

Use the following pieces of context to answer the question at the end.  
If you don't know the answer, just say that you don't know, don't try to  
make up an answer.  
  
{context} // i.e the pdf text content  
  
Question: {query} // i.e our actualy query, 'Who is the CV about?'  
Helpful Answer:

That is why if you try asking random questions, like “Where is Paris?”, the chatbot will respond saying it does not know.

To display the entire prompt that is sent to the LLM, you can set the verbose=True flag on the load\_qa\_chain() method, which will print to the console all the information that is actually being sent in the prompt. This can help to understand how it is working in the working in background, and what prompt is actually being sent to the OpenAI API.

chain = load\_qa\_chain(llm=OpenAI(), verbose=True)

As we mentioned at the start, this method is all good when we only have a short amount of information to send in the context. Most LLMs will have a limit on the amount of information that can be sent in a single request. So we will not be able to send all the information in our documents within a single request.

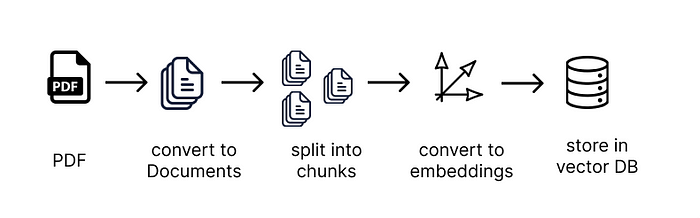
To overcome this, we need a smart way to send only the information we think will be relevant to our question/prompt.

**Interacting With a Single PDF Using Embeddings**

**Embeddings to the rescue!**

As explained earlier, we can use embeddings and vector stores to send only relevant information to our prompt. The steps we will need to follow are:

* Split all the documents into small chunks of text
* Pass each chunk of text into an embedding transformer to turn it into an embedding
* Store the embeddings and related pieces of text in a vector store



Let’s get to it!

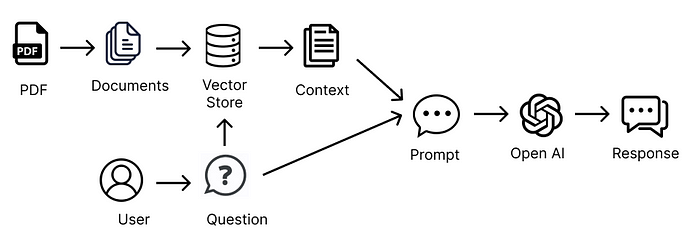
To get started, let’s create a new file called single-long-doc.py, to symbolise this script can be used for handling PDFs that are too long to pass in as context to a prompt

touch single-long-doc.py

Now, add the following code to the file. The steps are explained in the comments in the code. Remember to add your API key.

import os  
from langchain.document\_loaders import PyPDFLoader  
from langchain.text\_splitter import CharacterTextSplitter  
from langchain.vectorstores import Chroma  
from langchain.embeddings import OpenAIEmbeddings  
  
os.environ["OPENAI\_API\_KEY"] = "sk-"  
  
# load the document as before  
loader = PyPDFLoader('./docs/RachelGreenCV.pdf')  
documents = loader.load()  
  
# we split the data into chunks of 1,000 characters, with an overlap  
# of 200 characters between the chunks, which helps to give better results  
# and contain the context of the information between chunks  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)  
documents = text\_splitter.split\_documents(documents)  
  
# we create our vectorDB, using the OpenAIEmbeddings tranformer to create  
# embeddings from our text chunks. We set all the db information to be stored  
# inside the ./data directory, so it doesn't clutter up our source files  
vectordb = Chroma.from\_documents(  
 documents,  
 embedding=OpenAIEmbeddings(),  
 persist\_directory='./data'  
)  
vectordb.persist()

Once we have loaded our content as embeddings into the vector store, we are back to a similar situation as to when we only had one PDF to interact with. As in, we are now ready to pass information into the LLM prompt. However, instead of passing in all the documents as a source for our context to the chain, as we did initially, we will pass in our vector store as a source, which the chain will use to retrieve only the relevant text based on our question and send that information only inside the LLM prompt.



building a prompt using only relevant information from our document sources

We will use the RetrievalQA chain this time, which can use our vector store as a source for the context information.

Again, the chain will wrap our prompt with some text instructing it to only use the information provided for answering the questions. So the prompt we end up sending to the LLM something that looks like this:

Use the following pieces of context to answer the question at the end.  
If you don't know the answer, just say that you don't know, don't try to  
make up an answer.  
  
{context} // i.e the chunks of text retrieved deemed to be moset semantically  
 // relevant to our question  
  
Question: {query} // i.e our actualy query  
Helpful Answer:

So, let’s create the RetrievalQA chain, and make some queries to the LLM. We create the RetrievalQA chain, passing in the vector store as our source of information. Behind the scenes, this will only retrieve the relevant data from the vector store based on the semantic similarity between the prompt and the stored.

Notice we set search\_kwargs={‘k’: 7} on our retriever, which means we want to send seven chunks of text from our vector store to our prompt. Any more than this, and we will overuse the OpenAI prompt token limit. But the more info we have, the more accurate our answers will be, so we do want to send in as much as possible. This [article provides](https://www.activeloop.ai/resources/data-chad-an-ai-app-with-lang-chain-deep-lake-to-chat-with-any-data/) some nice information on tuning parameters for doc chatbot LLMs.

from langchain.chains import RetrievalQA  
from langchain.llms import OpenAI  
  
qa\_chain = RetrievalQA.from\_chain\_type(  
 llm=OpenAI(),  
 retriever=vectordb.as\_retriever(search\_kwargs={'k': 7}),  
 return\_source\_documents=True  
)  
  
# we can now execute queries against our Q&A chain  
result = qa\_chain({'query': 'Who is the CV about?'})  
print(result['result'])

Now, run the script, and you should see the result.

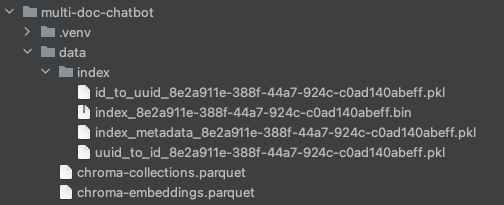
➜ multi-doc-chatbot python3 single-long-doc.py  
Rachel Green.

Awesome! We now have our document reader and chatbot working using embeddings and vector stores!

Notice we set the persist\_directory for the vector store to be ./data . So this is the location where the vector database will store all of its information, including the embedding vectors it generates and the chunks of text associated with each of the embeddings.

If you open the data directory inside the project root folder, you will see all of the DB files inside there. Cool right! Like a MySQL or Mongo database, it has its own directories that store all of the information.

If you change the code or documents stored, and the chatbot responses started looking strange, try deleting this directory, and it will recreate on the next script run. That can sometimes help with strange responses.



**Adding Chat History**

Now, if we want to take things one step further, we can also make it so that our chatbot will remember any previous questions.

Implementation-wise, all that happens is that on each interaction with the chatbot, all of our previous conversation history, including the questions and answers, needs to be passed into the prompt. That is because the LLM does not have a way to store information about our previous requests, so we must pass in all the information on every call to the LLM.

Fortunately, LangChain also has a set of classes that let us do this out of the box. This is called the ConversationalRetrievalChain, which allows us to pass in an extra parameter called chat\_history , which contains a list of our previous conversations with the LLM.

Let’s create a new script for this, called multi-doc-chatbot.py (we will add multi-doc support a bit later on 😉).

touch multi-doc-chatbot.py

Set up the PDF loader, text splitter, embeddings, and vector store as before. Now, let’s initiate the Q&A chain.

from langchain.chains import ConversationalRetrievalChain  
from langchain.chat\_models import ChatOpenAI  
  
qa\_chain = ConversationalRetrievalChain.from\_llm(  
 ChatOpenAI(),  
 vectordb.as\_retriever(search\_kwargs={'k': 6}),  
 return\_source\_documents=True  
)

The chain run command accepts the chat\_history as a parameter. So first of all, let’s enable a continuous conversation via the terminal by nesting the stdin and stout commands inside a While loop. Next, we must manually build up this list based on our conversation with the LLM. The chain does not do this out of the box. So for each question and answer, we will build up a list called chat\_history , which we will pass back into the chain run command each time.

import sys  
  
chat\_history = []  
while True:  
 # this prints to the terminal, and waits to accept an input from the user  
 query = input('Prompt: ')  
 # give us a way to exit the script  
 if query == "exit" or query == "quit" or query == "q":  
 print('Exiting')  
 sys.exit()  
 # we pass in the query to the LLM, and print out the response. As well as  
 # our query, the context of semantically relevant information from our  
 # vector store will be passed in, as well as list of our chat history  
 result = qa\_chain({'question': query, 'chat\_history': chat\_history})  
 print('Answer: ' + result['answer'])  
 # we build up the chat\_history list, based on our question and response  
 # from the LLM, and the script then returns to the start of the loop  
 # and is again ready to accept user input.  
 chat\_history.append((query, result['answer']))

Delete the data directory so that it recreates on the next run. The responses can act strange sometimes when you change the chains and code setup without deleting the data created from the previous setups.

Run the script,

python3 multi-doc-chatbot.py

and start interacting with the document. Notice that it can recognise context from previous questions and answers. You can submit exit or q to leave the script.

multi-doc-chatbot python3 multi-doc-chatbot.py  
Prompt: Who is the CV about?  
Answer: The CV is about Rachel Green.  
Prompt: And their surname only?  
Answer: Rachel Greens surname is Green.  
Prompt: And first?  
Answer: Rachel.

So, that’s it! We have now built a chatbot that can interact with multiple of our own documents, as well as maintain a chat history. But wait, we are still only interacting with a single PDF, right?

**Interacting With Multiple Documents**

Interacting with multiple documents is easy. If you remember, the Documents created from our PDF Document Loader is just a list of Documents, i.e., a List[Document]. So to increase our base of documents to interact with, we can just add more Documents to this list.

Let’s add some more files to our docs folder. You can copy the remaining sample docs from the [GitHub repository docs folder](https://github.com/smaameri/multi-doc-chatbot/tree/master/docs). Now there should be a .pdf, .docx, and .txt file in our docs folder.

Now we can simply iterate over all of the files in that folder, and convert the information in them into Documents. From then onwards, the process is the same as before. We just pass our list of documents to the text splitter, which passes the chunked information to the embeddings transformer and vector store.

So, in our case, we want to be able to handle pdfs, Microsoft Word documents, and text files. We will iterate over the docs folder, handle files based on their extensions, use the appropriate loaders for them, and add them to the documentslist, which we then pass on to the text splitter.

from langchain.document\_loaders import Docx2txtLoader  
from langchain.document\_loaders import TextLoader  
  
documents = []  
for file in os.listdir('docs'):  
 if file.endswith('.pdf'):  
 pdf\_path = './docs/' + file  
 loader = PyPDFLoader(pdf\_path)  
 documents.extend(loader.load())  
 elif file.endswith('.docx') or file.endswith('.doc'):  
 doc\_path = './docs/' + file  
 loader = Docx2txtLoader(doc\_path)  
 documents.extend(loader.load())  
 elif file.endswith('.txt'):  
 text\_path = './docs/' + file  
 loader = TextLoader(text\_path)  
 documents.extend(loader.load())  
  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=10)  
chunked\_documents = text\_splitter.split\_documents(documents)  
  
# we now proceed as earlier, passing in the chunked\_documents to the  
# to the vectorstore  
# ...

Now you can run the script again and ask questions about all the candidates. It seems to help if you delete the /data folder after adding new files to the docs folder. The chatbot doesn’t seem to pick up the new information otherwise.

python3 multi-doc-chatbot.py

So there we have it, a chatbot that is able to interact with information from multiple documents, as well as maintain a chat history. We can jazz things up by adding some colour to the terminal outputs and handling empty string inputs. Here is a full copy of the script below:

import sys  
import os  
from langchain.document\_loaders import PyPDFLoader  
from langchain.document\_loaders import Docx2txtLoader  
from langchain.document\_loaders import TextLoader  
from langchain.embeddings import OpenAIEmbeddings  
from langchain.vectorstores import Chroma  
from langchain.llms import OpenAI  
from langchain.chat\_models import ChatOpenAI  
from langchain.chains import ConversationalRetrievalChain  
from langchain.memory import ConversationBufferMemory  
from langchain.text\_splitter import CharacterTextSplitter  
from langchain.prompts import PromptTemplate  
  
os.environ["OPENAI\_API\_KEY"] = "sk-XXX"  
  
documents = []  
for file in os.listdir("docs"):  
 if file.endswith(".pdf"):  
 pdf\_path = "./docs/" + file  
 loader = PyPDFLoader(pdf\_path)  
 documents.extend(loader.load())  
 elif file.endswith('.docx') or file.endswith('.doc'):  
 doc\_path = "./docs/" + file  
 loader = Docx2txtLoader(doc\_path)  
 documents.extend(loader.load())  
 elif file.endswith('.txt'):  
 text\_path = "./docs/" + file  
 loader = TextLoader(text\_path)  
 documents.extend(loader.load())  
  
text\_splitter = CharacterTextSplitter(chunk\_size=1000, chunk\_overlap=10)  
documents = text\_splitter.split\_documents(documents)  
  
vectordb = Chroma.from\_documents(documents, embedding=OpenAIEmbeddings(), persist\_directory="./data")  
vectordb.persist()  
  
pdf\_qa = ConversationalRetrievalChain.from\_llm(  
 ChatOpenAI(temperature=0.9, model\_name="gpt-3.5-turbo"),  
 vectordb.as\_retriever(search\_kwargs={'k': 6}),  
 return\_source\_documents=True,  
 verbose=False  
)  
  
yellow = "\033[0;33m"  
green = "\033[0;32m"  
white = "\033[0;39m"  
  
chat\_history = []  
print(f"{yellow}---------------------------------------------------------------------------------")  
print('Welcome to the DocBot. You are now ready to start interacting with your documents')  
print('---------------------------------------------------------------------------------')  
while True:  
 query = input(f"{green}Prompt: ")  
 if query == "exit" or query == "quit" or query == "q" or query == "f":  
 print('Exiting')  
 sys.exit()  
 if query == '':  
 continue  
 result = pdf\_qa(  
 {"question": query, "chat\_history": chat\_history})  
 print(f"{white}Answer: " + result["answer"])  
 chat\_history.append((query, result["answer"]))

**The LangChain repository**

To understand what is happening behind the scenes a bit more, I would encourage you to download the LangChain source code and poke around to see how it works.

git clone https://github.com/hwchase17/langchain

If you browse the source code using an IDE like PyCharm (I think the [community edition](https://www.jetbrains.com/pycharm/download/#section=mac) is free), you can hot-click (CMD + click) into each of the method and class calls, and it will take you straight to where they are written, which is super useful for clicking around the code base to see how things work.

**Improvements**

When you start playing around with the chatbot, and see how it responds to different questions, you will notice is only sometimes gives the right answers.

Our current method does have limitations. For example, the OpenAI token limit is 4,096 tokens, which means we cannot send more than around 6–7 chunks of text from the vector DB. This means we may not even be sending information from all our documents, which would be important if we wanted to know, for example, the names of all the people in our CV documents. For example, one of the documents could be totally missed out, and so we would miss a crucial piece of information.

And we have only three documents here. Imagine having 100s. At some point, a token limit of just 4096 is just not going to be enough to give us an accurate answer. Maybe you would need to use a different LLM, for example, other than the OpenAI one, where you can have a higher token limit, so you can send more context in it. One of the features you are hearing about these days is LLMs with greater and greater token limits.

If the document source size is too large, maybe training an LLM on your data is the way to go, instead of sending the information via prompt contexts. Or maybe some other smart tweaks could be made to chain parameters or vector store retrieval techniques. Maybe smart prompt engineering or some kind of agents for recursive lookups would be the way to go. [This article](https://www.obytes.com/blog/langchain-guide) gives some idea of how you can tweak prompts to give you better responses.

Likely it would be a combination of all these, and the answer might also vary depending on the types of documents you would want to parse. For example, if you choose to focus on a specific document type, for example, CVs, user manuals, or website scrapes, there might be certain optimisations more suited for specific types of content.

Overall, to get a multidocument reader that works well, I think you need to go a little beyond the surface parts of just getting it to work and start figuring out some of these enhancements that could make it a much more capable and useful chatbot.

**Summary**

So, that’s it. We built a single-document chatbot and finished with a multi-document chatbot that remembers our chat history. Hopefully, the article helped to take some of the mystery out of embeddings, vector stores, and parameter tuning on the chains and vector store retrievers.

I wrote another article [here](https://medium.com/@ssmaameri/youtube-chatbot-using-langchain-and-openai-f8faa8f34929) on building a YouTube video chatbot. The concepts are very similar to the article here, except that instead of using text documents (e.g pdfs) as the source for our content, we use the YouTube video’s transcripts, which then allows us to interact with and ask questions about the videos.

Hope that was useful. Cheers!

*To keep in touch with my latest posts, you can give me a follow on Medium and*[*Twitter*](https://twitter.com/smaameri)*.*