Build a Chatbot on Your CSV Data With LangChain and OpenAI

Chat with your CSV files with a memory chatbot 🤖 | Made with Langchain 🦜 and OpenAI 🧠

Yvann · Follow

Published in Better Programming
5 min read · Apr 13, 2023

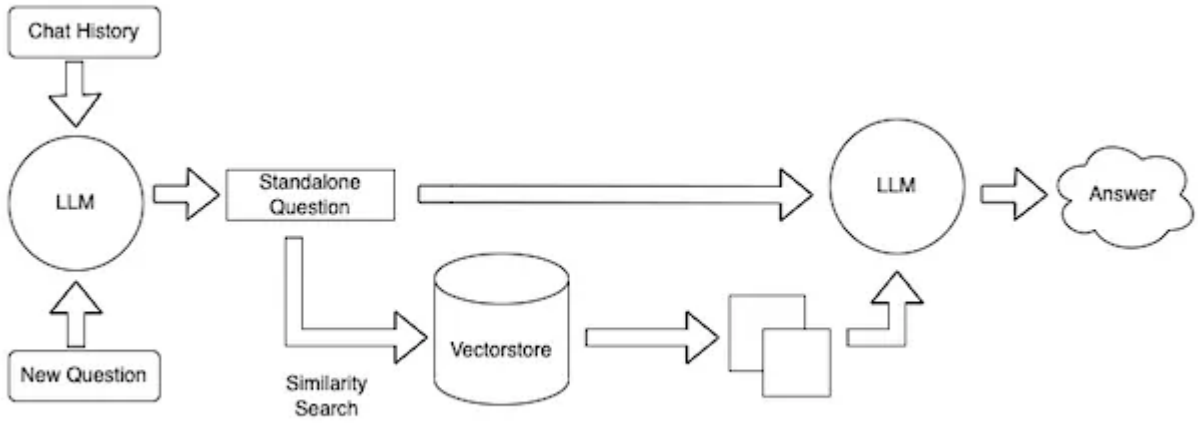
image made with StableDiffusion

> *In this article, we'll see how to build a simple chatbot 🤖 with memory that can answer your questions about your own CSV data.*

Hi everyone! In the past few weeks, I have been experimenting with the fascinating potential of large language models to create all sorts of things, and it's time to share what I've learned!

We'll use LangChain 🦜 to link `gpt-3.5` to our data and Streamlit to create a user interface for our chatbot.

Unlike ChatGPT, which offers limited context on our data (we can only provide a maximum of 4096 tokens), our chatbot will be able to process CSV data and manage a large database thanks to the use of embeddings and a vectorstore.



A diagram of the process used to create a chatbot on your data, from LangChain Blog

The code

Now let's get practical! We'll develop our chatbot on CSV data with very little Python syntax.

> ***Disclaimer**: This code is a simplified version of the chatbot I created, it is not optimized to reduce OpenAI API costs, for a more performant and optimized chatbot, feel free to check out my GitHub project : yvann-hub/Robby-chatbot or just test the app at Robby-chatbot.com 🚀.*

- First, we'll install the necessary libraries:

```
pip install streamlit streamlit_chat langchain openai faiss-cpu tiktoken
```

- Import the libraries needed for our chatbot:

```
import streamlit as st
from streamlit_chat import message
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.chat_models import ChatOpenAI
from langchain.chains import ConversationalRetrievalChain
from langchain.document_loaders.csv_loader import CSVLoader
from langchain.vectorstores import FAISS
import tempfile
```

- We ask the user to enter their OpenAI API key and download the CSV file on which the chatbot will be based.
- To test the chatbot at a lower cost, you can use this lightweight CSV file: `fishfry-locations.csv`

```
user_api_key = st.sidebar.text_input(
    label="#### Your OpenAI API key 👇",
    placeholder="Paste your openAI API key, sk-",
    type="password")

uploaded_file = st.sidebar.file_uploader("upload", type="csv")
```

- If a CSV file is uploaded by the user, we load it using the CSVLoader class from LangChain

```
if uploaded_file :
   #use tempfile because CSVLoader only accepts a file_path
    with tempfile.NamedTemporaryFile(delete=False) as tmp_file:
        tmp_file.write(uploaded_file.getvalue())
        tmp_file_path = tmp_file.name

    loader = CSVLoader(file_path=tmp_file_path, encoding="utf-8", csv_args={
                'delimiter': ','})
    data = loader.load()
```

- The LangChain `CSVLoader` class allows us to split a CSV file into unique rows. This can be seen by displaying the content of the data:

```
st.write(data)
```

```
0:"Document(page_content='venue_name: McGinnis Sisters\nvenue_type: Market\nver
1:"Document(page_content='venue_name: Holy Cross (Reilly Center)\nvenue_type: C
```

- Cutting the CSV file now allows us to provide it to our `vectorstore` (FAISS) using OpenAI embeddings.
- Embeddings allow transforming the parts cut by CSVLoader into vectors, which then represent an index based on the content of each row of the given file.
- In practice, when the user makes a query, a search will be performed in the vectorstore, and the best matching index(es) will be returned to the LLM, which will rephrase the content of the found index to provide a formatted response to the user.
- I recommend deepening your understanding of vectorstore and embeddings concepts for better comprehension.

```
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(data, embeddings)
```

- We then add the `ConversationalRetrievalChain` by providing it with the desired chat model `gpt-3.5-turbo` (or gpt-4) and the FAISS vectorstore storing our file transformed into vectors by `OpenAIEmbeddings()`.
- This chain allows us to have a chatbot with memory while relying on a `vectorstore` to find relevant information from our document.

```
chain = ConversationalRetrievalChain.from_llm(
llm = ChatOpenAI(temperature=0.0,model_name='gpt-3.5-turbo'),
retriever=vectorstore.as_retriever())
```

- This function allows us to provide the user's question and conversation history to `ConversationalRetrievalChain` to generate the chatbot's response.
- `st.session_state['history']` stores the user's conversation history when they are on the Streamlit site.

> *If you want to add improvements to this chatbot you can check my GitHub* 👀

```
def conversational_chat(query):

    result = chain({"question": query,
    "chat_history": st.session_state['history']})
    st.session_state['history'].append((query, result["answer"]))

    return result["answer"]
```

- We initialize the chatbot session by creating st.session_state['history'] and the first messages displayed in the chat.
- `['generated']` corresponds to the chatbot's responses.
- `['past']` corresponds to the messages provided by the user.
- Containers are not essential but help improve the UI by placing the user's question area below the chat messages.

```
if 'history' not in st.session_state:
    st.session_state['history'] = []

if 'generated' not in st.session_state:
    st.session_state['generated'] = ["Hello ! Ask me anything about " + upl

if 'past' not in st.session_state:
    st.session_state['past'] = ["Hey ! 👋"]
```

```
#container for the chat history
response_container = st.container()
#container for the user's text input
container = st.container()
```

- Now that the `session.state` and containers are configured.
- We can set up the UI part that allows the user to enter and send their question to our `conversational_chat` function with the user's question as an argument.

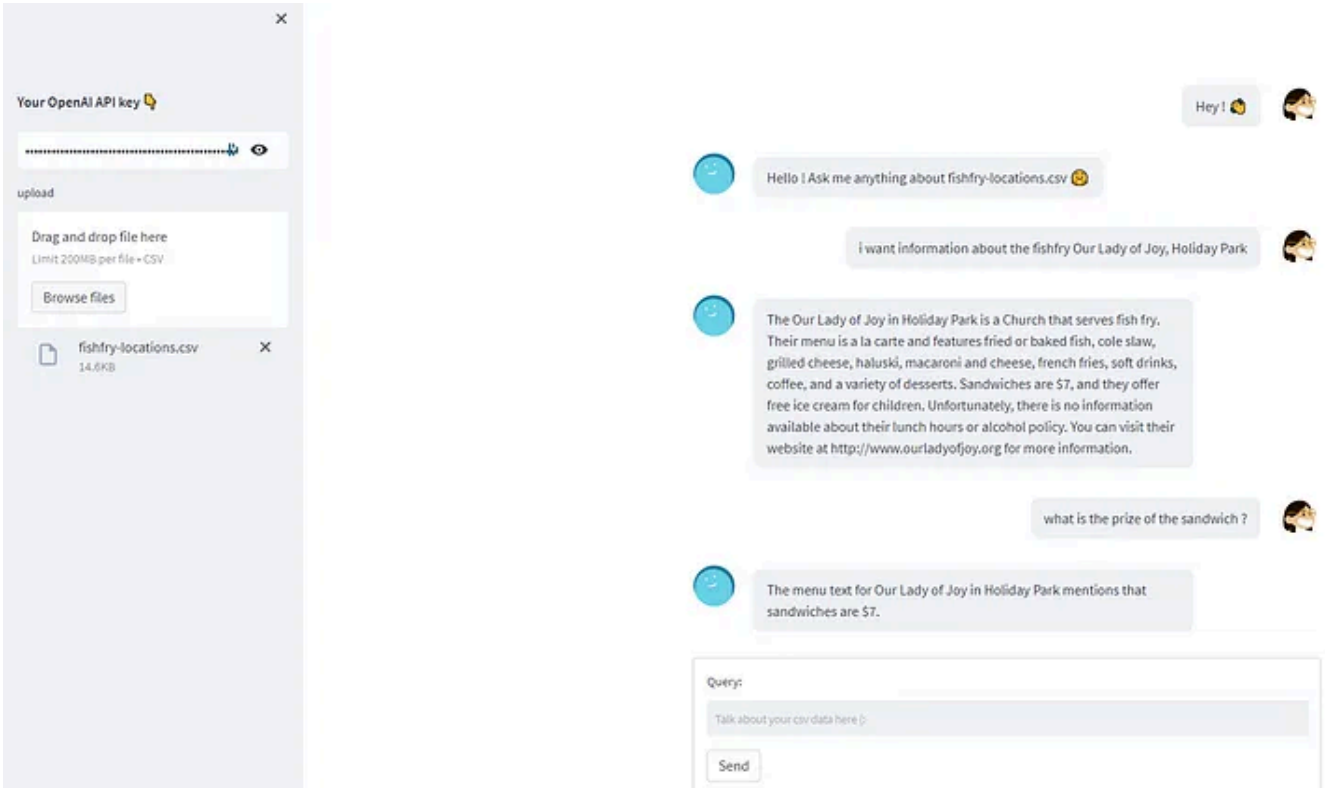
```
with container:
    with st.form(key='my_form', clear_on_submit=True):

        user_input = st.text_input("Query:", placeholder="Talk about your c
        submit_button = st.form_submit_button(label='Send')

    if submit_button and user_input:
        output = conversational_chat(user_input)

        st.session_state['past'].append(user_input)
        st.session_state['generated'].append(output)
```

- This last part allows displaying the user's and chatbot's messages on the Streamlit site using the `streamlit_chat` module.

```
if st.session_state['generated']:
    with response_container:
        for i in range(len(st.session_state['generated'])):
            message(st.session_state["past"][i], is_user=True, key=str(i) +
            message(st.session_state["generated"][i], key=str(i), avatar_st
```

- All that's left is to launch the script:

```
streamlit run name_of_your_chatbot.py #run with the name of your file
```



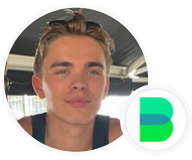
The result after launch the last command

Et voilà! You now have a beautiful chatbot running with LangChain, OpenAI, and Streamlit, capable of answering your questions based on your CSV file!

I hope this article will help you to create nice things, do not hesitate to contact me on **Twitter** or at barbot.yvann@gmail.com if you need.

You also can find the full project on my GitHub.

Langchain | OpenAI | AI | Large Language Models | Streamlit



Follow

Written by Yvann

291 Followers · Writer for Better Programming

42 School Student 🎓 | France | Build AI Tools 🤖 | GitHub : https://github.com/yvann-ba

More from Yvann and Better Programming

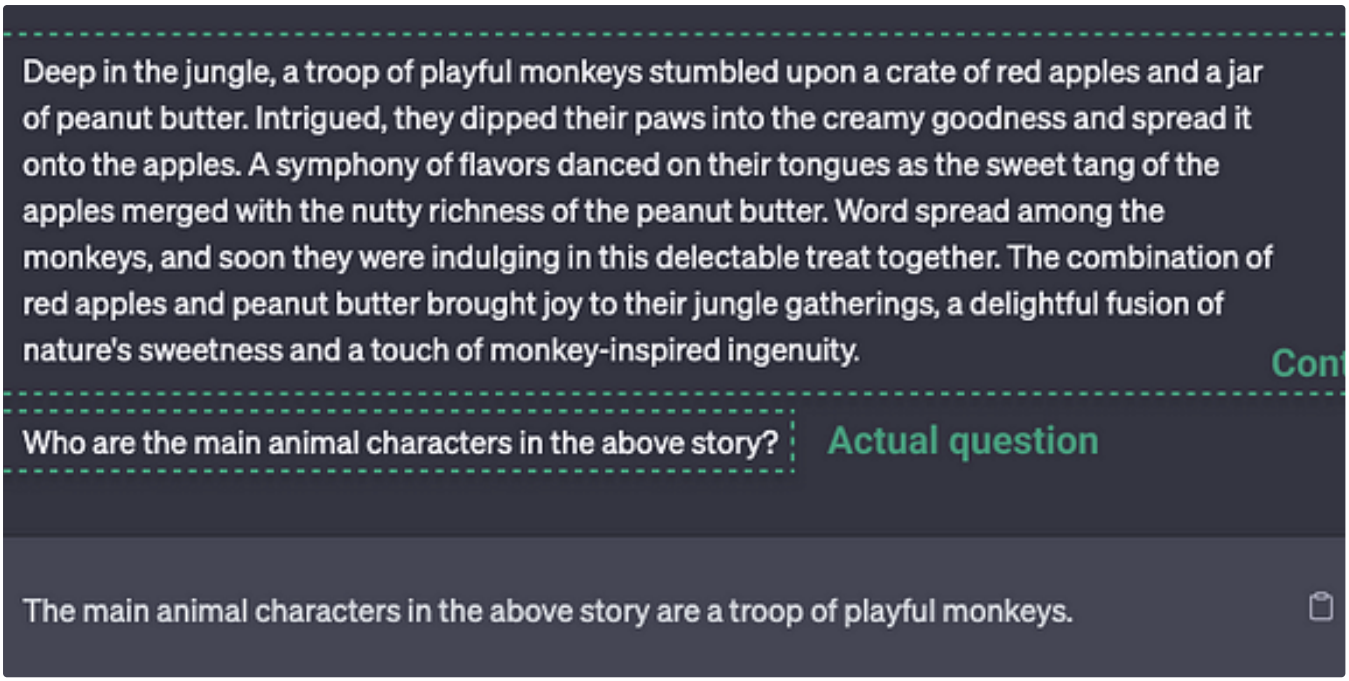
Benoit Ruiz in Better Programming

Advice From a Software Engineer With 8 Years of Experience

Practical tips for those who want to advance in their careers

22 min read · Mar 20, 2023

14.7K 273



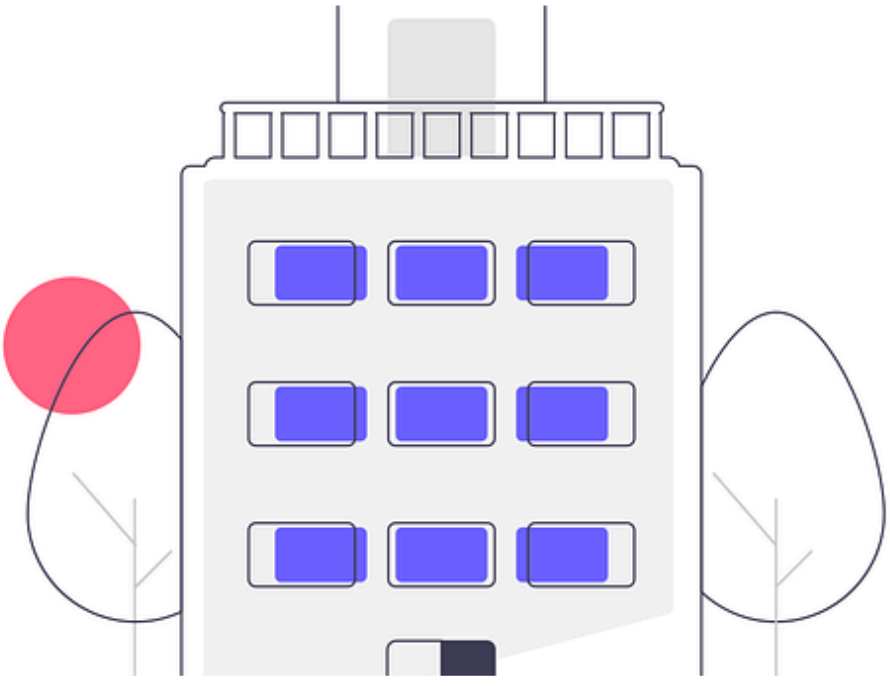
Sami Maameri in Better Programming

Building a Multi-document Reader and Chatbot With LangChain and ChatGPT

The best part? The chatbot will remember your chat history

17 min read · May 20, 2023

1.8K 17



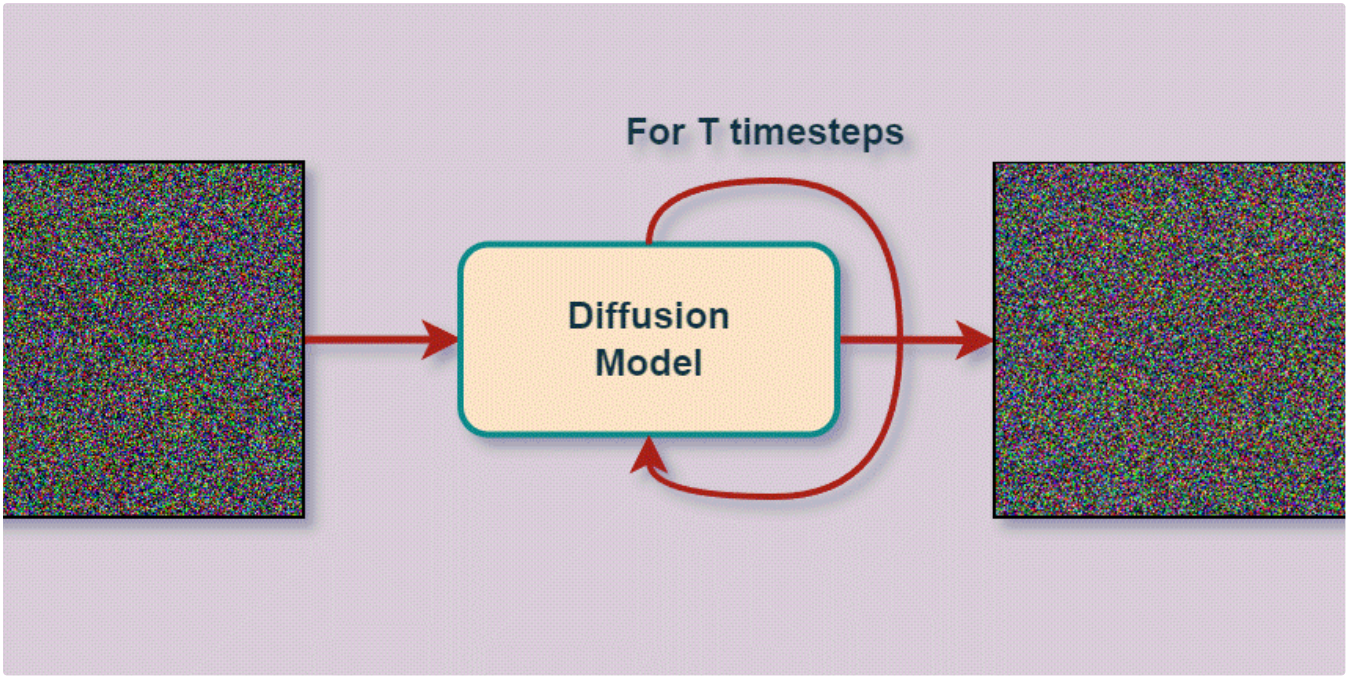
Bharath in Better Programming

The Clean Architecture — Beginner's Guide

As explained with visual illustrations

6 min read · Jan 4, 2022

3.1K 23



Gabriel Mongaras in Better Programming

Diffusion Models—DDPMs, DDIMs, and Classifier Free Guidance

A guide to the evolution of diffusion models from DDPMs to Classifier Free guidance

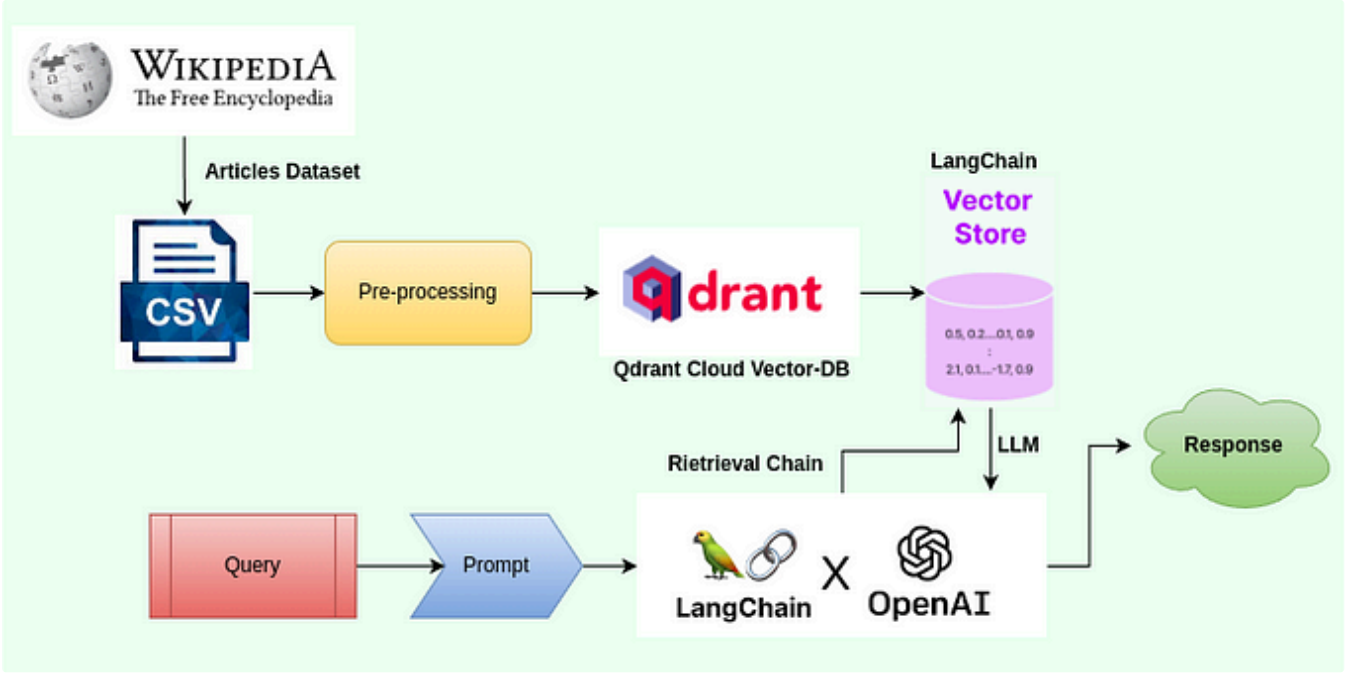
28 min read · Mar 13, 2023

718 8

See all from Yvann

See all from Better Programming

Recommended from Medium



Sidgraph in GoPenAI

Chat with Large CSV Data Using Qdrant, Langchain, and OpenAI

Today, chatbots are at the forefront of every organization. Due to the exponential increase in industry-scale Large Language Models (LLMs)...

8 min read · Dec 20, 2023

223 2

Writers@Tintash

Talking to your CSV using OpenAI and LangChain

Ever since OpenAI released ChatGPT, the world of Large Language Models (LLM) has been advancing at a breakneck pace. These LLMs are a...

7 min read · Nov 17, 2023

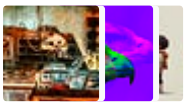
14

Lists

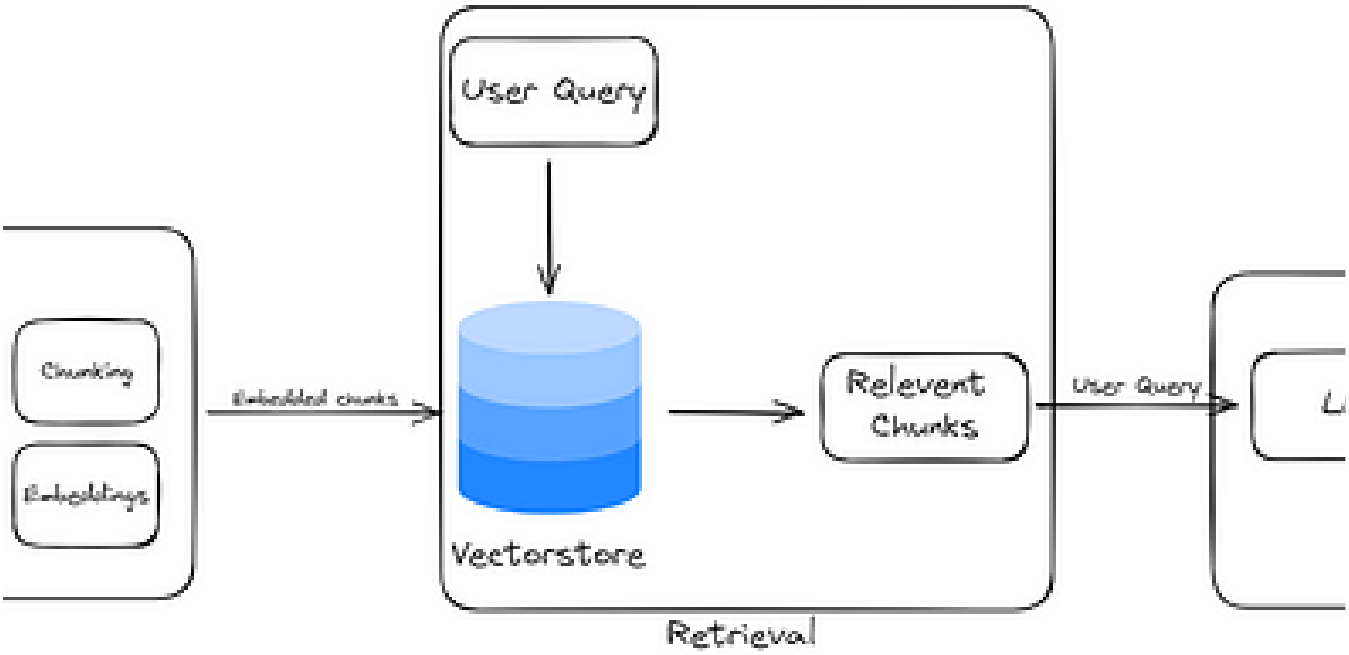
Natural Language Processing
1340 stories · 821 saves

Generative AI Recommended Reading
52 stories · 892 saves

AI Regulation
6 stories · 394 saves



What is ChatGPT?
9 stories · 332 saves



Vivas.AI

Overcoming the Hurdles: Navigating RAG Failure When Handling Large CSV Files

Introduction

4 min read · Jan 29, 2024

16

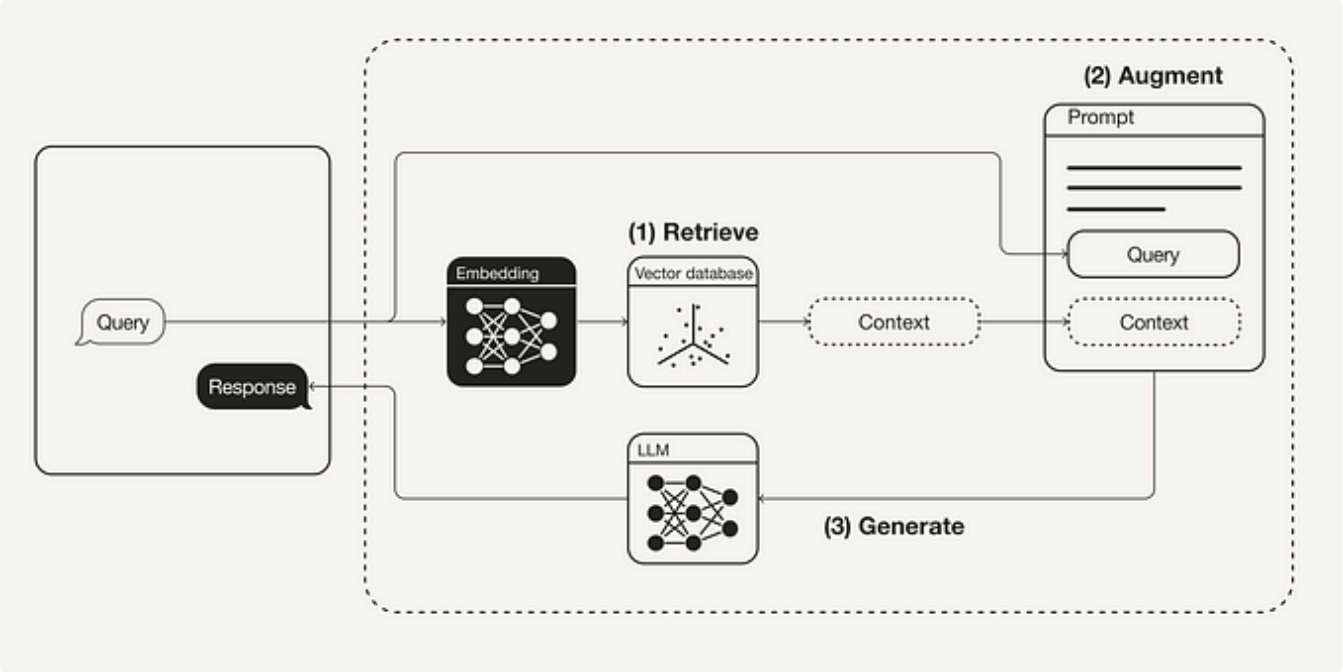
Carlo C. in **AI monks.io**

Multiple-PDF Chatbot using Langchain

The goal of the project is to create a question answering system based on information retrieval, which is able to answer questions posed by...

6 min read · Oct 22, 2023

181 1



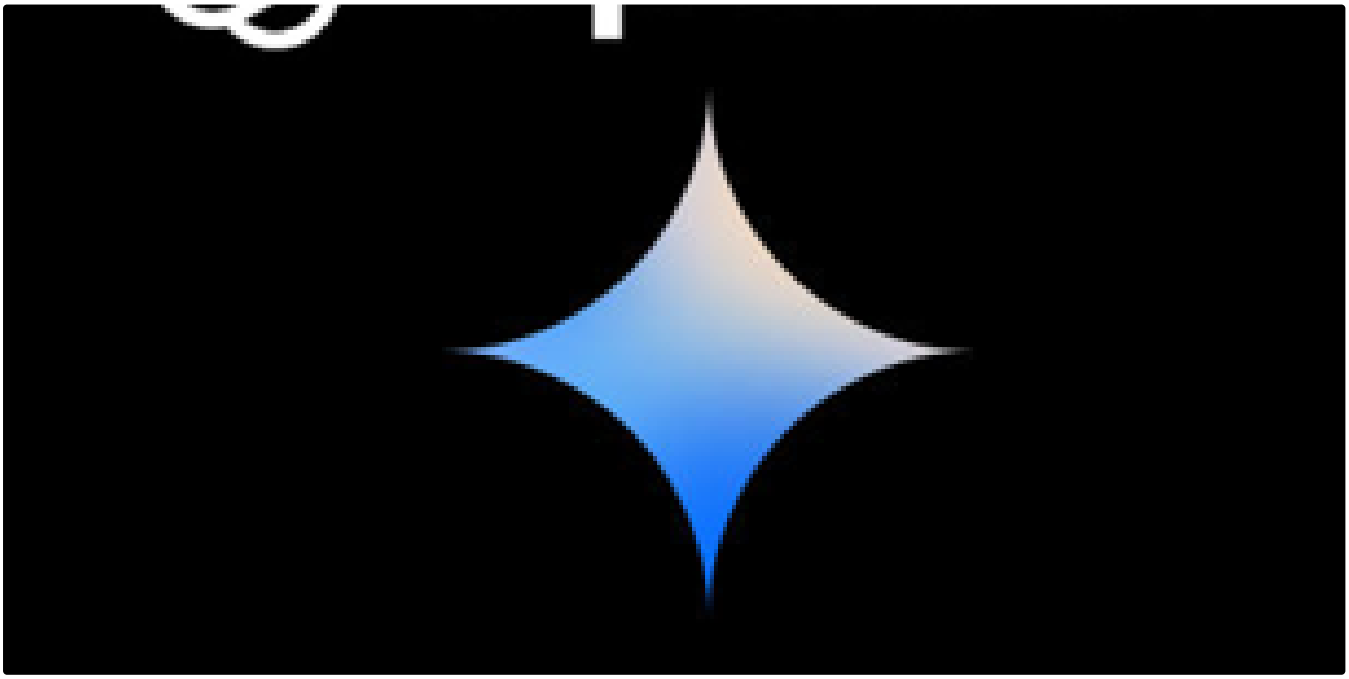
Leonie Monigatti in Towards Data Science

Retrieval-Augmented Generation (RAG): From Theory to LangChain Implementation

From the theory of the original academic paper to its Python implementation with OpenAI, Weaviate, and LangChain

7 min read · Nov 14, 2023

1.4K 11



Ala Eddine GRINE in The Deep Hub

RAG chatbot powered by Langchain, OpenAI, Google Generative AI and Hugging Face APIs

Introduction

15 min read · Feb 14, 2024

 76  1