




Hot deal! Get up to 48% OFF – 8GB GPU Memory/8 Cores/64GB RAM, As Low As \$61.00/Month!>

How to build local RAG App with LangChain, Ollama, Python, and ChromaDB

Discover how to build a local RAG app using LangChain, Ollama, Python, and ChromaDB. Step-by-step guidance for developers seeking innovative solutions.

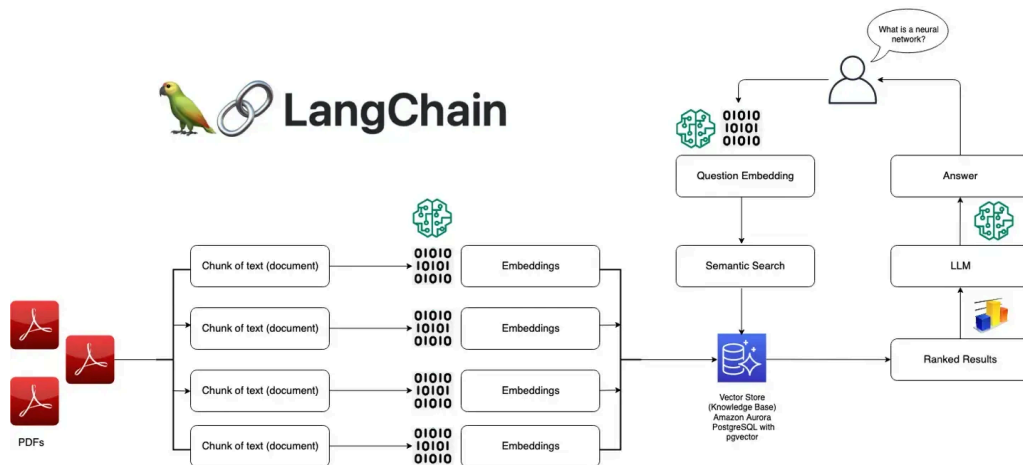
Introduction

In today's world, where data privacy is more important than ever, setting up your own local language model (LLM) offers a key solution for both businesses and individuals. This guide walks you through building a custom chatbot using LangChain, Ollama, Python 3, and ChromaDB, all hosted locally on your system. By following this tutorial, you'll gain the tools to create a powerful and secure local chatbot that meets your specific needs, ensuring full control and privacy every step of the way.



What is LangChain?

LangChain is a framework for developing applications powered by large language models (LLMs). LangChain simplifies every stage of the LLM application lifecycle. LangChain provides tools and abstractions to improve the customization, accuracy, and relevancy of the information the models generate. For example, developers can use LangChain components to build new prompt chains or customize existing templates. LangChain also includes components that allow LLMs to access new data sets without retraining.

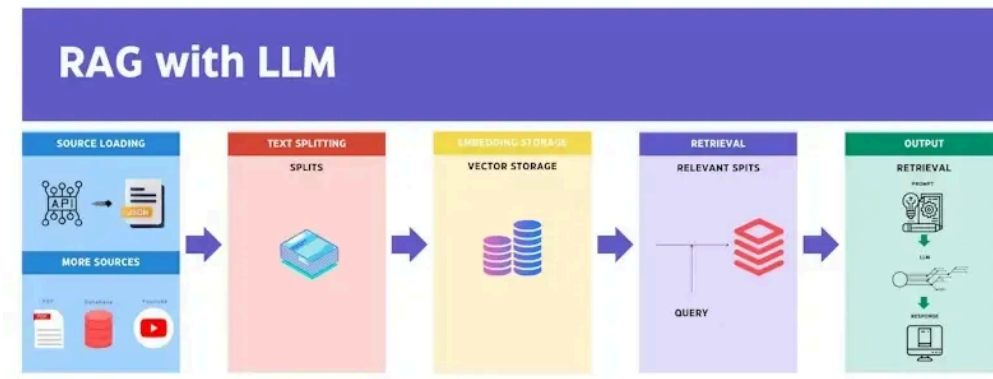


Why Use RAG Locally?

By setting up a local RAG application with tools like Ollama, Python, and ChromaDB, you can enjoy the benefits of advanced language models while maintaining control over your data and customization options.

- **Privacy and Security:** Running a RAG model locally ensures that sensitive data remains secure and private, as it does not need to be sent to external servers.

- **Customization:** You can tailor the retrieval and generation processes to suit your specific needs, including integrating proprietary data sources.
- **Independence:** A local setup ensures that your system remains operational even without internet connectivity, providing consistent and reliable service.



Prerequisites

- **CUDA-capable GPU:** Ensure you have an NVIDIA GPU installed. When setting up a local LLM, the choice of GPU can significantly impact performance.
- **Python 3:** Python is a versatile programming language that you'll use to write the code for your RAG app.
- **ChromaDB:** A vector database that will store and manage the embeddings of our data.
- **Ollama:** To download and serve custom LLMs in our local machine.

Steps to Build a RAG App

Now that you've set up your environment with Python, Ollama, ChromaDB and other dependencies, it's time to build your custom local RAG app. In this section, we'll walk through the hands-on Python code and provide an overview of how to structure your application.

Step 1. Install Python and other dependencies

To install and setup our Python 3 environment, follow these steps: Download and setup Python 3 on your machine. Then make sure your Python 3 installed and run successfully:

```
$ python3 --version
# Python 3.12.2
```

Create a folder for your project, for example, local-rag:

```
$ mkdir local-rag
$ cd local-rag
```

Create a virtual environment named venv and activate it:

```
$ python3 -m venv venv
$ source venv/bin/activate
# Windows
# venv\Scripts\activate
```

Install Flask to serve your app as a HTTP service:

```
$ pip install --q flask
```

Step 2. Install and run Ollama server

Ollama provides the backend infrastructure needed to run LLMs locally. To get started, head to [Ollama's website](#) and download the application. Follow the instructions to set it up on your local machine. By default, it's running on <http://localhost:11434>.

```
$ ollama --version
# ollama version is 0.3.11

$ ollama pull llama3.2

$ ollama pull nomic-embed-text

$ ollama serve
```

Step 3. Install Chroma and LangChain

Install ChromaDB using pip:

```
$ pip install --q chromadb
```

Install LangChain tools to work seamlessly with your model:

```
$ pip install --q unstructured langchain
langchain-text-splitters
$ pip install --q "unstructured[all-docs]"
```

Step 4. Create an entry file - app.py

This is the main Flask application file. It defines routes for embedding files to the vector database, and retrieving the response from the model.

```
import os
from dotenv import load_dotenv
```

```

load_dotenv()

from flask import Flask, request, jsonify
from embed import embed
from query import query
from get_vector_db import get_vector_db

TEMP_FOLDER = os.getenv('TEMP_FOLDER', './_temp')
os.makedirs(TEMP_FOLDER, exist_ok=True)

app = Flask(__name__)

@app.route('/embed', methods=['POST'])
def route_embed():
    if 'file' not in request.files:
        return jsonify({"error": "No file part"}), 400

    file = request.files['file']

    if file.filename == '':
        return jsonify({"error": "No selected file"}), 400

    embedded = embed(file)

    if embedded:
        return jsonify({"message": "File embedded successfully"}), 200

    return jsonify({"error": "File embedded unsuccessfully"}), 400

@app.route('/query', methods=['POST'])
def route_query():
    data = request.get_json()

```

```

        response = query(data.get('query'))

    if response:
        return jsonify({"message": response}),
200

    return jsonify({"error": "Something went
wrong"}), 400

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8080,
debug=True)

```

Step 5. Create files to process private documents - embed.py

This module handles the embedding process, including saving uploaded files, loading and splitting data, and adding documents to the vector database.

```

import os
from datetime import datetime
from werkzeug.utils import secure_filename
from langchain_community.document_loaders import
UnstructuredPDFLoader
from langchain_text_splitters import
RecursiveCharacterTextSplitter
from get_vector_db import get_vector_db

TEMP_FOLDER = os.getenv('TEMP_FOLDER', './_temp')

# Function to check if the uploaded file is
allowed (only PDF files)
def allowed_file(filename):
    return '.' in filename and
filename.rsplit('.', 1)[1].lower() in {'pdf'}

```

```

# Function to save the uploaded file to the
temporary folder
def save_file(file):
    # Save the uploaded file with a secure
filename and return the file path
    ct = datetime.now()
    ts = ct.timestamp()
    filename = str(ts) + "_" +
secure_filename(file.filename)
    file_path = os.path.join(TEMP_FOLDER,
filename)
    file.save(file_path)

    return file_path

# Function to load and split the data from the
PDF file
def load_and_split_data(file_path):
    # Load the PDF file and split the data into
chunks
    loader =
UnstructuredPDFLoader(file_path=file_path)
    data = loader.load()
    text_splitter =
RecursiveCharacterTextSplitter(chunk_size=7500,
chunk_overlap=100)
    chunks = text_splitter.split_documents(data)

    return chunks

# Main function to handle the embedding process
def embed(file):
    # Check if the file is valid, save it, load
and split the data, add to the database, and
remove the temporary file
    if file.filename != '' and file and

```



```

allowed_file(file.filename):
    file_path = save_file(file)
    chunks = load_and_split_data(file_path)
    db = get_vector_db()
    db.add_documents(chunks)
    db.persist()
    os.remove(file_path)

    return True

return False

```

Step 6. Create files that handle user queries - query.py

This module processes user queries by generating multiple versions of the query, retrieving relevant documents, and providing answers based on the context.

```

import os
from langchain_community.chat_models import
ChatOllama
from langchain.prompts import ChatPromptTemplate,
PromptTemplate
from langchain_core.output_parsers import
StrOutputParser
from langchain_core.runnables import
RunnablePassthrough
from langchain.retrievers.multi_query import
MultiQueryRetriever
from get_vector_db import get_vector_db

LLM_MODEL = os.getenv('LLM_MODEL', 'mistral')

# Function to get the prompt templates for
generating alternative questions and answering

```

```

based on context
def get_prompt():
    QUERY_PROMPT = PromptTemplate(
        input_variables=["question"],
        template="""You are an AI language model
assistant. Your task is to generate five
        different versions of the given user
question to retrieve relevant documents from
        a vector database. By generating multiple
perspectives on the user question, your
        goal is to help the user overcome some of
the limitations of the distance-based
        similarity search. Provide these
alternative questions separated by newlines.
        Original question: {question}""",
    )

    template = """Answer the question based ONLY
on the following context:
    {context}
    Question: {question}
    """

    prompt =
ChatPromptTemplate.from_template(template)

    return QUERY_PROMPT, prompt

# Main function to handle the query process
def query(input):
    if input:
        # Initialize the language model with the
specified model name
        llm = ChatOllama(model=LLM_MODEL)
        # Get the vector database instance
        db = get_vector_db()
        # Get the prompt templates

```

```

        QUERY_PROMPT, prompt = get_prompt()

        # Set up the retriever to generate
        multiple queries using the language model and the
        query prompt
        retriever = MultiQueryRetriever.from_llm(
            db.as_retriever(),
            llm,
            prompt=QUERY_PROMPT
        )

        # Define the processing chain to retrieve
        context, generate the answer, and parse the
        output
        chain = (
            {"context": retriever, "question":
RunnablePassthrough()}}
            | prompt
            | llm
            | StrOutputParser()
        )

        response = chain.invoke(input)

        return response

    return None

```

Step 7. Create a file to obtain a vector database instance - `get_vector_db.py`

This module initializes and returns the vector database instance used for storing and retrieving document embeddings.

```

import os
from langchain_community.embeddings import

```

```

OllamaEmbeddings
from langchain_community.vectorstores.chroma
import Chroma

CHROMA_PATH = os.getenv('CHROMA_PATH', 'chroma')
COLLECTION_NAME = os.getenv('COLLECTION_NAME',
'local-rag')
TEXT_EMBEDDING_MODEL =
os.getenv('TEXT_EMBEDDING_MODEL', 'nomic-embed-
text')

def get_vector_db():
    embedding =
OllamaEmbeddings(model=TEXT_EMBEDDING_MODEL, show_
progress=True)

    db = Chroma(
        collection_name=COLLECTION_NAME,
        persist_directory=CHROMA_PATH,
        embedding_function=embedding
    )

    return db

```

Step 8. Run your app!

Create .env file to store your environment variables:

```

TEMP_FOLDER = './_temp'
CHROMA_PATH = 'chroma'
COLLECTION_NAME = 'local-rag'
LLM_MODEL = 'mistral'
TEXT_EMBEDDING_MODEL = 'nomic-embed-text'

```

Run the app.py file to start your app server:

```
$ python3 app.py
```

Once the server is running, you can start making requests to the following endpoints:

- Example command to embed a PDF file (e.g., NatureDeepReview.pdf):

```
$ curl --request POST \
  --url http://localhost:8080/embed \
  --header 'Content-Type: multipart/form-data' \
  --form
file=@/Users/fen/Documents/NatureDeepReview.pdf
```

```
# Response
```

```
{
  "message": "File embedded successfully"
}
```

- Example command to ask a question to your model:

```
$ curl --request POST \
  --url http://localhost:8080/query \
  --header 'Content-Type: application/json' \
  --data '{ "query": "What are the advantages of
deep learning over traditional machine learning?"
}'
```

```
# Response
```

```
{
  "message": "Deep learning allows computational
models composed of multiple processing layers to
learn representations of data with multiple
levels of abstraction. These methods have greatly
improved the state of the art in speech
recognition, visual object recognition, object
detection, and many other fields such as drug
discovery and genomics. Deep learning discovers
```

complex structures in large data sets by using a backpropagation algorithm to indicate how the machine should change its internal parameters used to compute representations at each layer, which are derived from the representations of the previous layer. Deep convolutional networks have made breakthroughs in processing images, video, speech, and audio, while recurrent networks have brought light to sequence data such as text and speech."

}

Conclusion

By following this guide, you'll be able to run and interact with your custom local RAG (Retrieval-Augmented Generation) app using Python, Ollama, LangChain, and ChromaDB, all tailored to your specific needs. Feel free to modify and expand its functionality to push the boundaries of what your application can achieve.

A local deployment offers more than just privacy—it gives you the power to fully optimize performance and responsiveness. Whether you're aiming to boost customer engagement or streamline internal workflows, your locally hosted RAG app provides the flexibility to grow and evolve with your needs, all while maintaining tight control over sensitive data.

Additional - Reference Links

<https://dev.to/nassermaronie/build-your-own-rag-app-a-step-by-step-guide-to-setup-llm-locally-using-ollama-python-and-chromadb-b12>

<https://jws.news/2024/how-to-build-a-rag-system-using-python-ollama-langchain-and-chroma-db/>

<https://medium.com/@vnndee.huynh/build-your-own-rag-and-run-it-locally-langchain-ollama-streamlit-181d42805895>



GPU Mart offers professional GPU hosting services that are optimized for high-performance computing projects. We support a wide variety of GPU cards, providing fast processing speeds and reliable uptime for complex applications such as deep learning algorithms and simulations. Additionally, our expert support team is available 24/7 to assist with any technical challenges that may arise.



📍 257 Westwood Dr, League City, TX 77573
Copyright © 2005 - 2025 Database Mart LLC

GPU Hosting

GeForce GT 710 Hosting

GeForce GT 730 Hosting

GeForce GTX 1650 Hosting

GeForce GTX 1660 Hosting

GeForce RTX 2060 Hosting

GeForce RTX 3060Ti Hosting

GeForce RTX 4060 Hosting

GeForce RTX 4090 Hosting

GeForce RTX 5060 Hosting

GeForce RTX 5090 Hosting

Quadro P600 Hosting

Quadro k620 Hosting

Quadro P620 Hosting

Quadro P1000 Hosting

Quadro T1000 Hosting

Quadro RTX A4000 Hosting

Quadro RTX A5000 Hosting

Quadro RTX A6000 Hosting

Nvidia Tesla K40 Hosting

Nvidia Tesla K80 Hosting

Nvidia Tesla P100 Hosting

Nvidia A40 Hosting

Nvidia V100 Hosting

Nvidia A100 Hosting

Nvidia H100 Hosting

GPU Solutions

Cloud GPU Server
Cheap GPU Server
TensorFlow GPU Server
GPU Emulation Hosting
Android Emulator Hosting
BlueStacks GPU Hosting
Keras GPU Hosting
MXNet GPU Hosting
PyTorch GPU Hosting
Deep Learning GPU Hosting
Streaming GPU Hosting
LDPlayer GPU Hosting
Rendering GPU Hosting
XGBoost GPU Hosting
OBS GPU Hosting
Octane Render GPU Hosting
Nvidia GeForce Server
Nvidia Quadro RTX Server
RTX Server Hosting
Tesla GPU Hosting
Nox Player GPU Hosting
MEmu Play GPU Hosting
GameLoop GPU Hosting
Streamlabs GPU Hosting
Discord GPU Hosting
Stable Diffusion GPU Servers

Company

Pricing
About Us
Data Centers
Affiliate
Backlink Exchange
Partner

Legal

[Terms of Service](#)

[Privacy Policy](#)

[Service Level Agreement](#)

Help Center

[Submit Ticket](#)

[Send Email](#)

[Online Chat](#)

[Blog](#)

[FAQ](#)

[Knowledgebase](#)

[Web Hosting](#)

[Cloud Hosting](#)

[VPS Hosting](#)

[RDP Hosting](#)

[Bare Metal Server](#)