# LangChain Basics: A Beginner's Guide

---

## □ What is LangChain?

**LangChain** is an open-source framework that makes it easier to build **applications powered by large language models (LLMs)** like GPT-4, Claude, or LLaMA.

It helps you:

- Connect LLMs with **external data (like PDFs, APIs, or databases)**
- Create **chatbots**, **agents**, **search tools**, and **automation workflows**
- Chain multiple steps or prompts together in a **structured way**

---

## □ Why Use LangChain?

Large language models are great—but on their own, they:

- **Don’t have memory**
- **Can’t use tools or access real-time data**
- **Struggle to work with structured logic**

LangChain solves these problems by:

- Adding **memory**
- Letting LLMs **interact with tools** (like calculators, search engines, or APIs)
- Enabling **complex workflows** (like summarizing a PDF → asking questions → storing answers)

LangChain supports many models:

- **OpenAI** (e.g., GPT-4)
- **Anthropic** (Claude)
- **Hugging Face Transformers**
- **Local models** via Ollama or Llama.cpp

Here's a **minimal LangChain example** that shows how to use a **local LLaMA model** (like LLaMA 2 or LLaMA 3) **without any document retrieval, chat UI, or external tools** — just the core language model running locally.

---

## □ Goal:

Use **LangChain** + **`llama-cpp-python`** to run a basic prompt with a **local `.gguf` model**.

---

## □ Step 1: Install Required Libraries

```
pip install langchain llama-cpp-python
```

□ You also need a local `.gguf` model like `llama-3-8b.Q4_K_M.gguf`

Download from TheBloke on Hugging Face.

## Step 2: Minimal Code (LLaMA only)

```
from langchain.llms import LlamaCpp

# Load your local LLaMA model
llm = LlamaCpp(
    model_path="model/llama-3-8b.Q4_K_M.gguf",  # Change to your actual path
    n_ctx=2048,
```

```
    temperature=0.7,
    top_p=0.95,
    n_threads=4,  # Set based on your CPU cores
    verbose=True
)

# Run a prompt
prompt = "Explain LangChain in simple terms."
response = llm(prompt)

print("🧠 Response:")
print(response)
```

## 🧠 What This Does:

- Loads the LLaMA model locally with no internet or API key
- Sends a simple prompt and gets a response
- Uses only `langchain` and `llama-cpp-python`

## 🔧 Optional Parameters

You can tweak:

- `n_batch`: Number of tokens processed in parallel (start with 32 or 64)
- `stop`: List of stop words or strings
- `max_tokens`: Limit output length

Example:

`Python:-`

```
llm = LlamaCpp(
    model_path="model/llama-3-8b.Q4_K_M.gguf",
    temperature=0.5,
    max_tokens=100,
    n_threads=4,
    n_batch=64,
    verbose=True,
```

```
    stop=["\nUser:", "\nAI:"]
)
```

Great! Here's a **chat-style web UI** built with **Streamlit** that uses **LangChain + a local LLaMA model** (like LLaMA 3 in `.gguf` format). This gives you a lightweight, private ChatGPT-like experience — no OpenAI key needed.

---

## □ What You'll Get

- A **local LLM chatbot UI**
- Powered by **Streamlit** + **LangChain** + **llama-cpp-python**
- Simple and fast, runs fully offline with your `.gguf` model

---

## □ 1. Install the Required Packages

pip install streamlit langchain llama-cpp-python

## 📂 2. Folder Structure

```
chat_llama/
├── app.py
└── model/
    └── llama-3-8b.Q4_K_M.gguf
```

3. `app.py` – Full Code

```
import streamlit as st
from langchain.llms import LlamaCpp
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory

st.set_page_config(page_title="🧠 LLaMA 3 Chat", layout="centered")

st.title("🧠 Chat with Local LLaMA 3")
st.markdown("Running fully offline using `llama-cpp` and LangChain.")

# Initialize the LLaMA model (cached)
@st.cache_resource
def load_llm():
    return LlamaCpp(
        model_path="model/llama-3-8b.Q4_K_M.gguf",  # <- Update path if needed
        n_ctx=2048,
        n_threads=4,
        temperature=0.7,
        top_p=0.95,
        verbose=False,
        stop=["User:", "Assistant:"]
    )

llm = load_llm()

# Initialize conversation memory
if "memory" not in st.session_state:
    st.session_state.memory = ConversationBufferMemory()

# Create conversation chain
conversation = ConversationChain(
    llm=llm,
    memory=st.session_state.memory,
    verbose=False
)

# Chat UI
user_input = st.chat_input("Ask something...")
if user_input:
    with st.spinner("Thinking..."):
        response = conversation.run(user_input)
```

```
    st.chat_message("user").markdown(user_input)
    st.chat_message("assistant").markdown(response)
```

## ▶□ 4. Run the Chat App

```
streamlit run app.py
```

Let's now build the **“Chat with PDF”** example using a **local LLaMA model**

We'll use:

- **LangChain**
- **LLaMA 3 (or LLaMA 2)** running locally via **`llama-cpp-python`**
- **FAISS** for vector search
- **PyPDFLoader** for PDF parsing

# LangChain Use Case Examples:-

| Use Case | Description |
|---|---|
| □ Chat with PDFs | Ask questions about a document |
| □ Custom ChatGPT | Add memory and tools to your own chatbot |
| □ RAG (Retrieval-Augmented Generation) | Pull relevant data from a database or website before answering |
| □ Agents | Autonomous GPTs that take actions and make decisions |

# □ Prerequisites

## □ Install Required Packages

```
pip install llama-cpp-python langchain faiss-cpu PyPDF2
```

You’ll also need to download a **LLaMA model file** like `llama-2-7b.Q4_K_M.gguf` or `llama-3-8b.Q4_K_M.gguf`.

You can get those from TheBloke on Hugging Face — choose a quantized GGUF version.

## 📁 Directory Structure

```
chat_with_pdf/
├── model/
│   └── llama-2-7b.Q4_K_M.gguf
├── your_pdf.pdf
└── chat_with_pdf.py
```

# Full Example with Local LLaMA

**chat_with_pdf.py**

```
import os
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.llms import LlamaCpp
from langchain.chains import RetrievalQA

# 1. Load and split PDF
loader = PyPDFLoader("your_pdf.pdf")
pages = loader.load()

splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
docs = splitter.split_documents(pages)

# 2. Generate embeddings
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-
MiniLM-L6-v2")
db = FAISS.from_documents(docs, embedding_model)

# 3. Load LLaMA model locally
llm = LlamaCpp(
    model_path="model/llama-2-7b.Q4_K_M.gguf",  # Path to your model
```

```
    n_ctx=2048,
    n_threads=4,     # Adjust based on CPU cores
    temperature=0.7,
    top_p=0.95,
    verbose=True
)

# 4. Create Retrieval QA chain
retriever = db.as_retriever()
qa_chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)

# 5. Ask questions
while True:
    query = input("Ask a question about the PDF (or 'exit'): ")
    if query.lower() in ['exit', 'quit']:
        break
    response = qa_chain.run(query)
    print("\nAnswer:", response, "\n")
```

# Notes

- **Model format** must be `.gguf` (for `llama-cpp-python`).
- The `sentence-transformers/all-MiniLM-L6-v2` embedding model is lightweight and works well for basic retrieval tasks.
- You can switch to **streamlit** or **Gradio** for a web app version later.

## Chat with PDF using Local LLaMA 3 Model

---

# 1. Prerequisites

- Make sure you have **LLaMA 3** model weights in `.gguf` format (quantized, e.g., `llama-3-8b.Q4_K_M.gguf`).
- Install the needed libraries:

```
pip install llama-cpp-python langchain faiss-cpu PyPDF2
```

---

# 2. Example Code

```
import os
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.llms import LlamaCpp
from langchain.chains import RetrievalQA

# Load PDF and split into chunks
loader = PyPDFLoader("your_pdf.pdf")
pages = loader.load()

splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
docs = splitter.split_documents(pages)

# Use HuggingFace embeddings for vectorization
embedding_model = HuggingFaceEmbeddings(model_name="sentence-transformers/all-
MiniLM-L6-v2")
vector_db = FAISS.from_documents(docs, embedding_model)

# Load LLaMA 3 model locally
llm = LlamaCpp(
    model_path="model/llama-3-8b.Q4_K_M.gguf",  # path to your LLaMA 3 GGUF model
    n_ctx=2048,
    n_threads=4,          # adjust based on your CPU cores
    temperature=0.7,
    top_p=0.95,
```

```
    verbose=True
)

# Create retrieval-based QA chain
retriever = vector_db.as_retriever()
qa_chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)

# Interactive question answering loop
print("Chat with your PDF! Type 'exit' to quit.")
while True:
    query = input("Ask a question: ")
    if query.lower() in ["exit", "quit"]:
        break
    answer = qa_chain.run(query)
    print("\nAnswer:", answer, "\n")
```

**Here's your** LangChain + local LLaMA 3 + PDF chat **example converted into a** Streamlit app**.**
**This lets you upload a PDF and ask questions through a web interface.**

```
import streamlit as st
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.llms import LlamaCpp
from langchain.chains import RetrievalQA
import tempfile
import os

st.title("📄 Chat with your PDF using Local LLaMA 3")
```

```python
# Upload PDF
uploaded_file = st.file_uploader("Upload a PDF file", type=["pdf"])

if uploaded_file:
    # Save uploaded file temporarily
    with tempfile.NamedTemporaryFile(delete=False, suffix=".pdf") as tmp_file:
        tmp_file.write(uploaded_file.read())
        pdf_path = tmp_file.name

    # Load and split PDF
    loader = PyPDFLoader(pdf_path)
    pages = loader.load()
    splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
    docs = splitter.split_documents(pages)

    # Embed documents
    embedding_model = HuggingFaceEmbeddings(model_name="sentence-
transformers/all-MiniLM-L6-v2")
    vector_db = FAISS.from_documents(docs, embedding_model)

    # Load LLaMA 3 model
    llm = LlamaCpp(
        model_path="model/llama-3-8b.Q4_K_M.gguf",  # Change to your model path
        n_ctx=2048,
        n_threads=4,
        temperature=0.7,
        top_p=0.95,
        verbose=False
    )

    # Create retrieval QA chain
    retriever = vector_db.as_retriever()
    qa_chain = RetrievalQA.from_chain_type(llm=llm, retriever=retriever)

    # Input question
    query = st.text_input("Ask a question about the PDF:")

    if query:
        with st.spinner("Thinking..."):
            answer = qa_chain.run(query)
        st.markdown(f"**Answer:** {answer}")

    # Clean up temp file on app rerun
    st.text("")  # Just to trigger rerun behavior
```

```
    os.unlink(pdf_path)
else:
    st.info("Please upload a PDF to start chatting.")
```

### How to run:

1. Save this as `app.py`
2. Make sure your llama 3 `.gguf` model is at `"model/llama-3-8b.Q4_K_M.gguf"`
3. Run:

```
streamlit run app.py
```

# Final Thoughts

LangChain is like the **backend framework** for AI apps. It gives large language models the **structure**, **memory**, and **tools** they need to perform useful tasks in the real world.

If you already understand Python and want to build cool projects with AI — **LangChain is one of the most powerful tools you can learn today**.