Creating a **local AI agent** using **LangChain**, **Ollama**, and **LLaMA 3** involves setting up a local LLM (LLaMA 3 via Ollama), integrating it with LangChain, and building an agent that can reason and act based on tools and memory.

Here's a **step-by-step guide** to help you set up your own **local AI agent**:

□ Requirements

- Python 3.9+
- Ollama installed and running locally
- LangChain library
- LLaMA 3 model pulled via Ollama

□ Step 1: Install the Required Tools

Install LangChain and supporting libraries:

```
pip install langchain langchain-community langchain-core
pip install ollama
pip install python-dotenv
```

□ Step 2: Install and Run Ollama with LLaMA 3

1. **Install Ollama** from: https://ollama.com/download
2. Pull the **LLaMA 3 model**:

```
ollama pull llama3
```

3. Start the model (optional if it's started via API):

```
ollama run llama3
```

□ Step 3: Create a Basic LangChain Agent with Ollama LLaMA 3

Here's a simple Python script to create a LangChain agent using the LLaMA 3 model running locally with Ollama.

```
from langchain.agents import initialize_agent, AgentType
from langchain_community.llms import Ollama
from langchain.tools import Tool
import requests

# Step 1: Connect to the local LLaMA 3 model via Ollama
llm = Ollama(model="llama3")

# Step 2: Define any custom tools (e.g., a simple calculator)
def simple_calculator(query: str) -> str:
    try:
        result = eval(query)
        return str(result)
    except:
        return "Invalid math expression."

tools = [
    Tool(
        name="Calculator",
        func=simple_calculator,
        description="Useful for math expressions. Input should be a math
expression like '2+2' or '5*6'"
    )
]

# Step 3: Initialize the Agent
agent = initialize_agent(
    tools,
    llm,
    agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
    verbose=True
)

# Step 4: Run the Agent
response = agent.run("What's 15 * 12? Also, explain what this result means in
simple terms.")
print(response)
```

□ Step 4: Add Memory (Optional but Powerful)

You can give your agent memory using `ConversationBufferMemory`:

```
from langchain.memory import ConversationBufferMemory

memory = ConversationBufferMemory(memory_key="chat_history")

agent_with_memory = initialize_agent(
    tools,
    llm,
    agent=AgentType.CONVERSATIONAL_REACT_DESCRIPTION,
    memory=memory,
    verbose=True
)

# Example conversation
print(agent_with_memory.run("Who won the 2022 World Cup?"))
print(agent_with_memory.run("What team did they beat?"))
```

□ Optional Tools You Can Add

- Web search (using SerpAPI or Tavily)
- File tools (for reading local documents)
- APIs (weather, stock prices, etc.)
- Retrieval tools (vector databases like Chroma, FAISS)

□ Summary

Component	Role
Ollama	Hosts the local LLaMA 3 model
LangChain	Framework for building LLM apps and agents
LLaMA 3	The actual LLM you're running locally
Tools	Functions your agent can use to extend capabilities
Memory	Stores chat history or context

Let's add **memory** to your LangChain agent using **Ollama + LLaMA 3**, so it can remember things from earlier in the conversation.

We'll use `ConversationBufferMemory`, which stores the dialogue history in memory (RAM) during runtime.

□ Full Code: LangChain Agent + Ollama + LLaMA 3 + Memory

```
from langchain.agents import initialize_agent, AgentType
from langchain_community.llms import Ollama
from langchain.memory import ConversationBufferMemory
from langchain.tools import Tool

# Step 1: Connect to the local LLaMA 3 model via Ollama
llm = Ollama(model="llama3")

# Step 2: Define a simple calculator tool
def simple_calculator(query: str) -> str:
    try:
        result = eval(query)
        return str(result)
    except Exception as e:
        return f"Error: {e}"

tools = [
    Tool(
        name="Calculator",
        func=simple_calculator,
        description="Useful for doing basic math calculations. Input should
be a math expression like '2+2' or '5*6'"
    )
]

# Step 3: Add memory (conversation history)
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True)

# Step 4: Initialize the agent with memory
agent = initialize_agent(
    tools=tools,
    llm=llm,
    agent=AgentType.CONVERSATIONAL_REACT_DESCRIPTION,
    memory=memory,
    verbose=True
)

# Step 5: Run the agent in a multi-turn conversation
print(agent.run("My name is Alex."))
print(agent.run("What's my name?"))
print(agent.run("What is 7 * 8?"))
print(agent.run("What did I ask you earlier?"))
```

□ What's Happening?

Component	Function
`ConversationBufferMemory`	Stores all previous messages during this session
`memory_key="chat_history"`	Key that LangChain uses to inject history into prompts
`return_messages=True`	Ensures memory is passed as structured chat messages (not plain text)

□ Sample Output

```
> Entering new AgentExecutor chain...
AI: Hello Alex!
> Finished chain.

> Entering new AgentExecutor chain...
AI: Your name is Alex.
> Finished chain.

> Entering new AgentExecutor chain...
AI: 7 * 8 = 56
> Finished chain.

> Entering new AgentExecutor chain...
AI: You told me your name is Alex and asked what your name is, then asked me
to calculate 7 * 8.
> Finished chain.
```

How to Implement FAISS-Based Persistent Memory

Here's how to do it with LangChain + Ollama + FAISS + LLaMA 3.

Step-by-Step Code

```
from langchain_community.llms import Ollama
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
import os

# Step 1: Load your local LLM (LLaMA 3)
llm = Ollama(model="llama3")

# Step 2: Use a sentence embedding model
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Step 3: Create or load FAISS vector store (persistent)
faiss_path = "faiss_index"

if os.path.exists(faiss_path):
    vectorstore = FAISS.load_local(faiss_path, embeddings=embedding_model)
else:
    # If creating new, load some initial text or memory
    loader = TextLoader("memory_seed.txt")  # Optional: a file with
background knowledge
    documents = loader.load()
    text_splitter = CharacterTextSplitter(chunk_size=500, chunk_overlap=50)
    docs = text_splitter.split_documents(documents)

    vectorstore = FAISS.from_documents(docs, embedding_model)
    vectorstore.save_local(faiss_path)

# Step 4: Add retriever-based memory
retriever = vectorstore.as_retriever(search_type="similarity", k=5)

memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True,
    output_key="answer"
)

# Step 5: Create the conversational retrieval chain
qa_chain = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=retriever,
    memory=memory,
    verbose=True
)
```

```
# Step 6: Run conversation with persistent memory
query1 = "My favorite color is green."
qa_chain({"question": query1})

query2 = "What color did I say I liked?"
response = qa_chain({"question": query2})
print(response["answer"])
```

What Happens Here?

- The user message (`query`) is embedded.
- LangChain uses **FAISS** to retrieve similar past messages or knowledge.
- The LLM answers using that retrieved context.
- The FAISS index is **saved to disk**, so it's persistent across runs.

Optional Enhancements

- ✍ Store every new user message as a document in FAISS after embedding.
- Add files (PDFs, notes, etc.) to vector store using `PyMuPDFLoader`, `PDFLoader`, etc.
- Use `CombinedMemory` if you want **short-term (buffer)** + **long-term (FAISS)** memory together.

Summary

Feature	FAISS Memory
Type	Vector-based (semantic search)
Persistence	Yes (saves to disk)
Strength	Recalls related info even if phrased differently
Best for	Long-term memory, knowledge base, context-rich agents

let’s walk through how to build a **LangChain AI agent** with:

 Local LLaMA 3 via Ollama
 FAISS for persistent memory
 PDFs as knowledge source (load + embed)

□ Goal

Create a chatbot that can:

- Load content from PDF files
- Embed and store that content in FAISS
- Use **semantic memory** to answer questions about the PDFs
- Use **LLaMA 3** locally via **Ollama**

□ Full Working Example with PDFs

Here’s a complete Python script that:

1. Loads PDF(s)
2. Splits + embeds text
3. Saves to persistent FAISS vector store
4. Builds a conversational agent with memory using LLaMA 3

Step-by-Step Code

```
from langchain_community.llms import Ollama
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter
import os

# === CONFIG ===
PDF_PATH = "your_file.pdf"        # <- Replace with your PDF file
FAISS_INDEX_PATH = "pdf_faiss_index"

# === STEP 1: Load LLaMA 3 from Ollama ===
llm = Ollama(model="llama3")

# === STEP 2: Use an embedding model ===
```

```
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# === STEP 3: Load PDF and split ===
loader = PyPDFLoader(PDF_PATH)
documents = loader.load()

# Split text into chunks (adjust size based on your needs)
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
docs = text_splitter.split_documents(documents)

# === STEP 4: Create or load FAISS vector store ===
if os.path.exists(FAISS_INDEX_PATH):
    vectorstore = FAISS.load_local(FAISS_INDEX_PATH,
embeddings=embedding_model)
else:
    vectorstore = FAISS.from_documents(docs, embedding_model)
    vectorstore.save_local(FAISS_INDEX_PATH)

# === STEP 5: Create retriever and memory ===
retriever = vectorstore.as_retriever(search_type="similarity", k=5)

memory = ConversationBufferMemory(
    memory_key="chat_history",
    return_messages=True,
    output_key="answer"
)

# === STEP 6: Build the Conversational Retrieval Chain ===
qa_chain = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=retriever,
    memory=memory,
    verbose=True
)

# === STEP 7: Chat with PDF knowledge ===
print("Ask questions about the PDF (Ctrl+C to quit)\n")

while True:
    try:
        query = input("You: ")
        result = qa_chain({"question": query})
        print("AI:", result["answer"])
    except KeyboardInterrupt:
        print("\nExiting.")
        break
```

□ Example Use Case

Assume you have a PDF: `company_policy.pdf`

You can ask:

- “What’s the company’s vacation policy?”
- “Who do I contact for IT support?”
- “Summarize the remote work section.”

The agent searches the PDF semantically (not just keyword match), finds the relevant chunks, and answers using LLaMA 3.

□ Requirements

Install these if you haven’t already:

```
pip install langchain langchain-community
pip install faiss-cpu
pip install sentence-transformers
pip install pypdf
pip install ollama
```

Let's now **wrap your LLaMA 3 + FAISS + PDF-based retrieval chatbot** into a **Gradio web app**, so you get a simple UI like ChatGPT — but all running **locally** and **offline** with Ollama + LangChain.

□ Features of This Chatbot

- □ Uses **LLaMA 3** locally via **Ollama**
- □ Loads and indexes **PDFs**
- □ Stores and retrieves content from **FAISS**
- □ Remembers context in the session (via `ConversationBufferMemory`)
- □ **Gradio** UI for chat

□ Folder Structure (Example)

```
project/
```

```
├── app.py                # Main Gradio chatbot app
├── documents/            # Folder containing PDFs
│   ├── doc1.pdf
│   └── doc2.pdf
└── faiss_index/          # Will be created automatically
```

□ Step-by-Step `app.py` Code

```
import os
import gradio as gr
from langchain_community.llms import Ollama
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter

# === CONFIG ===
PDF_DIR = "documents"
FAISS_INDEX_PATH = "faiss_index"

# === LLM: LLaMA 3 via Ollama ===
llm = Ollama(model="llama3")

# === Embedding Model ===
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# === Load or Create FAISS Vector Store ===
if os.path.exists(FAISS_INDEX_PATH):
    vectorstore = FAISS.load_local(FAISS_INDEX_PATH,
embeddings=embedding_model)
else:
    all_docs = []
    for filename in os.listdir(PDF_DIR):
        if filename.endswith(".pdf"):
            loader = PyPDFLoader(os.path.join(PDF_DIR, filename))
            documents = loader.load()
            splitter = CharacterTextSplitter(chunk_size=1000,
chunk_overlap=200)
            docs = splitter.split_documents(documents)
            all_docs.extend(docs)
    vectorstore = FAISS.from_documents(all_docs, embedding_model)
    vectorstore.save_local(FAISS_INDEX_PATH)

# === Create Retriever and Memory ===
retriever = vectorstore.as_retriever(search_type="similarity", k=5)
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True, output_key="answer")

# === Create Retrieval Chain ===
qa_chain = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=retriever,
```

```
    memory=memory,
    verbose=False
)

# === Chat Function for Gradio ===
def chat(message, chat_history):
    result = qa_chain({"question": message})
    response = result["answer"]
    chat_history.append((message, response))
    return chat_history, chat_history

# === Gradio UI ===
with gr.Blocks() as demo:
    gr.Markdown("#  PDF Chatbot (Local & Private)")
    chatbot = gr.Chatbot()
    msg = gr.Textbox(label="Ask something about the PDFs...")
    clear = gr.Button("Clear Chat")

    state = gr.State([])

    msg.submit(chat, [msg, state], [chatbot, state])
    clear.click(lambda: ([], []), outputs=[chatbot, state])

# === Launch the app ===
demo.launch()
```

□ Install Requirements

```
pip install langchain langchain-community faiss-cpu sentence-transformers
pip install pypdf gradio ollama
```

□ How to Run

1. Place your PDFs in the `documents/` folder.
2. Run the app:

```
python app.py
```

3. Open the Gradio link in your browser.
4. Ask questions like:
    - “Summarize the second PDF.”
    - “What are the steps mentioned on page 3?”
    - “Who is the author of the document?”

let’s now **add PDF upload support to the Gradio UI**, so users can dynamically upload new PDFs **during runtime**, and your app will **immediately ingest, embed, and store them into FAISS**.

□ What This Update Includes:

□ Upload new PDF(s) via Gradio
□ Embed + add to FAISS index on-the-fly
□ Continue chatting with updated knowledge
□ Still uses local LLaMA 3 via Ollama

□ Final Version of `app.py` with PDF Upload Support

```
import os
import gradio as gr
from langchain_community.llms import Ollama
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import CharacterTextSplitter

# === CONFIG ===
PDF_DIR = "documents"
FAISS_INDEX_PATH = "faiss_index"

# Ensure directory exists
os.makedirs(PDF_DIR, exist_ok=True)

# === Initialize LLM (LLaMA 3 via Ollama) ===
llm = Ollama(model="llama3")

# === Embedding Model ===
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# === Load or Create FAISS Vector Store ===
if os.path.exists(FAISS_INDEX_PATH):
    vectorstore = FAISS.load_local(FAISS_INDEX_PATH,
embeddings=embedding_model)
else:
    vectorstore = FAISS.from_documents([], embedding_model)  # Empty store
    vectorstore.save_local(FAISS_INDEX_PATH)

# === Retriever and Memory ===
retriever = vectorstore.as_retriever(search_type="similarity", k=5)
memory = ConversationBufferMemory(memory_key="chat_history",
return_messages=True, output_key="answer")
```

```
# === Conversational Chain ===
qa_chain = ConversationalRetrievalChain.from_llm(
    llm=llm,
    retriever=retriever,
    memory=memory,
    verbose=False
)

# === Function to process uploaded PDFs ===
def process_uploaded_pdfs(files):
    global vectorstore
    new_docs = []
    for file in files:
        filepath = os.path.join(PDF_DIR, file.name)
        file.save(filepath)

        loader = PyPDFLoader(filepath)
        docs = loader.load()
        splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=200)
        chunks = splitter.split_documents(docs)
        new_docs.extend(chunks)

    if new_docs:
        vectorstore.add_documents(new_docs)
        vectorstore.save_local(FAISS_INDEX_PATH)
        return f"□ Successfully added {len(new_docs)} new document chunks to
memory."
    else:
        return "□ No new content added."

# === Chat function ===
def chat(message, chat_history):
    result = qa_chain({"question": message})
    response = result["answer"]
    chat_history.append((message, response))
    return chat_history, chat_history

# === Gradio UI ===
with gr.Blocks() as demo:
    gr.Markdown("## □ Chat with your PDFs (Powered by LLaMA 3 + FAISS, all
local!)")

    with gr.Row():
        chatbot = gr.Chatbot()
        with gr.Column():
            msg = gr.Textbox(label="Ask a question...")
            upload = gr.File(label="Upload PDF(s)", file_types=[".pdf"],
file_count="multiple")
            upload_output = gr.Textbox(label="Upload Status",
interactive=False)
            clear = gr.Button("Clear Chat")

    state = gr.State([])

    msg.submit(chat, [msg, state], [chatbot, state])
    clear.click(lambda: ([], []), outputs=[chatbot, state])
```

```
    upload.upload(fn=process_uploaded_pdfs, inputs=[upload],
outputs=[upload_output])

# === Launch the app ===
demo.launch()
```

□ What Happens Now?

- PDFs uploaded through Gradio are:
  - Saved to disk (`documents/`)
  - Loaded and split into chunks
  - Embedded and added to FAISS vector store
- Your **LLM agent immediately** gains access to the new information.

□ Install These If You Haven't

```
pip install gradio langchain langchain-community faiss-cpu sentence-
transformers pypdf ollama
```

□ Example Use

1. Upload a PDF with company policy.
2. Ask: "What's the leave policy?"
3. Upload another PDF with employee contacts.
4. Ask: "Who is the HR contact?"