* **Isolation**: Keeps dependencies separate for each project to avoid conflicts.
* **Clean environment**: Prevents cluttering the global Python installation.
* **Reproducibility**: Makes it easy to recreate the same environment using requirements.txt.
* **Security**: Limits the risk by not installing unnecessary packages globally.
* **Python version control**: Allows using different Python versions for different projects (with tools like pyenv).
* Venv setup
  + Mac
    - python3 -m venv venv
    - source ./venv/bin/activate
  + Windows
    - python -m venv env
    - venv\Scripts\activate
* .env
  + OPENAI\_API\_KEY
* Requirements.txt
  + langchain[openai]
  + Python-dotenv
* pip install -r requirements.txt
* import os
* from dotenv import load\_dotenv
* from langchain.chat\_models import init\_chat\_model
* load\_dotenv()
* model = init\_chat\_model("gpt-4o-mini", model\_provider="openai")
* response = model.invoke("who is modi")
* print(response.content)
* GROQ\_API\_KEY
* Requirements.txt - langchain[groq]
* import os
* from dotenv import load\_dotenv
* from langchain.chat\_models import init\_chat\_model
* load\_dotenv()
* model = init\_chat\_model("llama-3.3-70b-versatile", model\_provider="groq")
* response = model.invoke("who is modi")
* print(response.content)

import os

from dotenv import load\_dotenv

from langchain.chat\_models import init\_chat\_model

from typing import Annotated

from typing\_extensions import TypedDict

from langgraph.graph import StateGraph, START, END

from langgraph.graph.message import add\_messages

load\_dotenv()

class State(TypedDict):

messages: Annotated[list, add\_messages]

graph\_builder = StateGraph(State)

llm = init\_chat\_model("llama-3.3-70b-versatile", model\_provider="groq")

def chatbot(state: State):

return {"messages": [llm.invoke(state["messages"])]}

# The first argument is the unique node name

# The second argument is the function or object that will be called whenever

# the node is used.

graph\_builder.add\_node("chatbot", chatbot)

graph\_builder.add\_edge(START, "chatbot")

graph\_builder.add\_edge("chatbot", END)

graph = graph\_builder.compile()

try:

img = graph.get\_graph().draw\_mermaid\_png()

with open("graph.png", "wb") as f:

f.write(img)

except Exception:

pass

# w → Write mode (creates the file if it does not exist, or overwrites if it does).

# b → Binary mode (means you are writing binary data, like images or audio—not text).

def stream\_graph\_updates(user\_input: str):

for event in graph.stream({"messages": [{"role": "user", "content": user\_input}]}):

for value in event.values():

print("Assistant:", value["messages"][-1].content)

while True:

try:

user\_input = input("User: ")

if user\_input.lower() in ["quit", "exit", "q"]:

print("Goodbye!")

break

stream\_graph\_updates(user\_input)

except:

# fallback if input() is not available

user\_input = "What do you know about LangGraph?"

print("User: " + user\_input)

stream\_graph\_updates(user\_input)

break

Save graph in image

try:

img = graph.get\_graph().draw\_mermaid\_png()

with open("graph.png", "wb") as f:

f.write(img)

except Exception:

pass

Prebuilt agent

from dotenv import load\_dotenv

load\_dotenv()

from langgraph.prebuilt import create\_react\_agent

agent = create\_react\_agent(

model="groq:llama-3.3-70b-versatile",

tools=[],

prompt="You are a helpful assistant"

)

# Run the agents

response = agent.invoke(

{"messages": [{"role": "user", "content": "what are large language models"}]}

)

print(response)

Weather tool with prompt

from dotenv import load\_dotenv

load\_dotenv()

from langgraph.prebuilt import create\_react\_agent

def get\_weather(city: str) -> str:

"""Get weather for a given city."""

return f"It's always sunny in {city}!"

agent = create\_react\_agent(

model="groq:llama-3.3-70b-versatile",

tools=[get\_weather],

prompt="always use weather tool to get weather conditions and call it just once"

)

# Run the agents

response = agent.invoke(

{"messages": [{"role": "user", "content": "how is the weather in bangalore"}]}

)

print(response)

No memory

from dotenv import load\_dotenv

load\_dotenv()

from langgraph.prebuilt import create\_react\_agent

agent = create\_react\_agent(

model="groq:llama-3.3-70b-versatile",

tools=[],

prompt="You are a helpful assistant"

)

# Run the agent

response = agent.invoke(

{"messages": [{"role": "user", "content": "when was he born"}]}

)

print(response)

With memory

from dotenv import load\_dotenv

load\_dotenv()

from langgraph.checkpoint.memory import InMemorySaver

from langgraph.prebuilt import create\_react\_agent

checkpointer = InMemorySaver()

agent = create\_react\_agent(

model="groq:llama-3.3-70b-versatile",

tools=[],

checkpointer=checkpointer,

prompt="You are a helpful assistant"

)

config = {"configurable": {"thread\_id": "1"}}

first\_response = agent.invoke(

{"messages": [{"role": "user", "content": "who is modi"}]},

config

)

second\_response = agent.invoke(

{"messages": [{"role": "user", "content": "when was he born?"}]},

config

)

print(first\_response)

print('-------------')

print(second\_response)

Structured Response

Examples -

* Mail - sub, body
* Travel itinerary - Day1, day2, hotel, budget
* Code generation - language, code
* Health report - condition, medicine recommendation

from dotenv import load\_dotenv

from pydantic import BaseModel

load\_dotenv()

from langgraph.prebuilt import create\_react\_agent

class MailResponse(BaseModel):

subject: str

body: str

agent = create\_react\_agent(

model="groq:llama-3.3-70b-versatile",

tools=[],

response\_format = MailResponse

)

config = {"configurable": {"thread\_id": "1"}}

response = agent.invoke(

{"messages": [{"role": "user", "content": "write a mail applying leave for travel"}]},

config

)

print(response)

print("------------------------------")

print(response["structured\_response"])

print("------------------------------")

print(response["structured\_response"].subject)

print("------------------------------")

print(response["structured\_response"].body)

**LangChain MCP Adapters**

* requirements.txt - langchain\_mcp\_adapters
* async def run\_agent(): # func inside that
* if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(run\_agent())

* import asyncio
* GITHUB\_TOKEN = os.getenv("GITHUB\_TOKEN")

from dotenv import load\_dotenv

load\_dotenv()

import asyncio

from langchain\_mcp\_adapters.client import MultiServerMCPClient

from langgraph.prebuilt import create\_react\_agent

import os

GITHUB\_TOKEN = os.getenv("GITHUB\_TOKEN")

async def run\_agent():

client = MultiServerMCPClient(

{

"github": {

"command": "npx",

"args": [

"-y",

"@modelcontextprotocol/server-github"

],

"env": {

"GITHUB\_PERSONAL\_ACCESS\_TOKEN": GITHUB\_TOKEN

},

"transport": "stdio"

}

}

)

tools = await client.get\_tools()

agent = create\_react\_agent("groq:llama-3.3-70b-versatile", tools)

response = await agent.ainvoke({"messages": "what are the files present in repository keertipurswani/EducosysGenerativeAI"})

print(response["messages"][-1].content)

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(run\_agent())

from dotenv import load\_dotenv

load\_dotenv()

import asyncio

from langchain\_mcp\_adapters.client import MultiServerMCPClient

from langgraph.prebuilt import create\_react\_agent

import os

GITHUB\_TOKEN = os.getenv("GITHUB\_TOKEN")

async def run\_agent():

client = MultiServerMCPClient(

{

"github": {

"command": "npx",

"args": [

"-y",

"@modelcontextprotocol/server-github"

],

"env": {

"GITHUB\_PERSONAL\_ACCESS\_TOKEN": GITHUB\_TOKEN

},

"transport": "stdio"

},

"filesystem": {

"command": "npx",

"args": [

"-y",

"@modelcontextprotocol/server-filesystem",

"/Users/keertipurswani/GitHub/GenAI Bootcamp ref"

],

"transport":"stdio"

}

}

)

tools = await client.get\_tools()

agent = create\_react\_agent("openai:gpt-4o", tools)

response = await agent.ainvoke({"messages": "what are the files present in GenAI Bootcamp REF Directory"})

print(response["messages"][-1].content)

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(run\_agent())

response = await agent.ainvoke({"messages": "Create a new file 'main6.py' in GenAI Bootcamp REF Directory"})

Create MCP Server

from mcp.server.fastmcp import FastMCP

import os

mcp = FastMCP("EducosysFileSystem")

@mcp.tool()

def addFile(filename: str):

"""Create a new file in current directory"""

if not os.path.exists(filename):

with open(filename, "w") as f:

pass

print(f"File '{filename}' created.")

else:

print(f"File '{filename}' already exists.")

@mcp.tool()

def addFolder(directory\_name: str):

"""Create a new Directory in current directory"""

if not os.path.exists(directory\_name):

os.mkdir(directory\_name)

print(f"Directory '{directory\_name}' created.")

else:

print(f"Directory '{directory\_name}' already exists.")

if \_\_name\_\_ == "\_\_main\_\_":

mcp.run(transport="stdio")

"EducosysFileSystem": {

"command": "python",

"args": [

"./filesystem\_mcp.py"

],

"transport":"stdio"

}

from dotenv import load\_dotenv

load\_dotenv()

import asyncio

from langchain\_mcp\_adapters.client import MultiServerMCPClient

from langgraph.prebuilt import create\_react\_agent

import os

GITHUB\_TOKEN = os.getenv("GITHUB\_TOKEN")

async def run\_agent():

client = MultiServerMCPClient(

{

"github": {

"command": "npx",

"args": [

"-y",

"@modelcontextprotocol/server-github"

],

"env": {

"GITHUB\_PERSONAL\_ACCESS\_TOKEN": GITHUB\_TOKEN

},

"transport": "stdio"

},

"filesystem": {

"command": "npx",

"args": [

"-y",

"@modelcontextprotocol/server-filesystem",

"/Users/keertipurswani/GitHub/GenAI Bootcamp"

],

"transport": "stdio"

}

}

)

tools = await client.get\_tools()

print("Tools -----------------")

print(tools)

print("-----------------------------")

agent = create\_react\_agent("openai:gpt-4o", tools)

response = await agent.ainvoke({"messages": "create an empty file educosys.txt in the directory /Users/keertipurswani/GitHub/GenAI Bootcamp"})

print(response["messages"][-1].content)

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(run\_agent())

Day 2

Streamlit

* Requirements.txt - streamlit

from dotenv import load\_dotenv

load\_dotenv()

from langgraph.checkpoint.memory import InMemorySaver

from langgraph.prebuilt import create\_react\_agent

import streamlit as st

st.title("Educosys Chatbot App")

if "messages" not in st.session\_state:

st.session\_state.messages = []

checkpointer = InMemorySaver()

agent = create\_react\_agent(

model="openai:gpt-4o",

tools=[],

checkpointer=checkpointer,

prompt="You are a helpful assistant"

)

def stream\_graph\_updates(user\_input : str):

assistant\_response = ""

with st.chat\_message("assistant"):

message\_placeholder = st.empty()

for event in agent.stream({"messages": [{"role": "user", "content": user\_input}]}, {"configurable": {"thread\_id": "def"}}):

for value in event.values():

new\_text = value["messages"][-1].content

assistant\_response += new\_text

message\_placeholder.markdown(assistant\_response)

st.session\_state.messages.append(("assistant", assistant\_response))

# Display previous chat history

for role, message in st.session\_state.messages:

with st.chat\_message(role):

st.markdown(message)

prompt = st.chat\_input("What is your question?")

if prompt:

# Display user input as a chat message

with st.chat\_message("user"):

st.markdown(prompt)

# Append user input to session state

st.session\_state.messages.append(("user", prompt))

# Get response from the chatbot based on user input

response = stream\_graph\_updates(prompt)

* Just for understanding of streaming - Outer loop for event =>
  + Loops over each streamed event (state updates from LangGraph execution)
  + An event in LangGraph is a snapshot of which node just ran and what it produced
  + Event looks like this -
    - {

"<node\_name>": {

"messages": [...],

"action": ...,

"observation": ...,

...

}

}

* Each value in the event corresponds to a node’s output
* streamlit run streamlit\_app.py

App using Multi-Modal Models

* .env - GEMINI\_API\_KEY
* Requirements.txt - google.genai

import streamlit as st

import os

from dotenv import load\_dotenv

from google import genai

from google.genai import types

from PIL import Image

from io import BytesIO

load\_dotenv()

client = genai.Client()

st.title("Educosys Image Generator")

user\_prompt = st.text\_input("What do you want to generate image for?")

if st.button("Generate Image"):

if not user\_prompt:

st.warning("Please enter the prompt!")

else:

try:

with st.spinner("Generating image..."):

response = client.models.generate\_content(

model="gemini-2.0-flash-exp-image-generation",

contents=user\_prompt,

config=types.GenerateContentConfig(

response\_modalities=['Text', 'Image']

)

)

st.subheader("Generated Image")

for part in response.candidates[0].content.parts:

if part.text is not None:

st.write(part.text)

elif part.inline\_data is not None:

image = Image.open(BytesIO((part.inline\_data.data)))

st.image(image)

except Exception as e:

st.error("Error generating image")

st.title("Educosys Image Caption Generator")

uploaded\_image = st.file\_uploader("Upload an image for caption generation", type=["png", "jpg", "jpeg"])

if uploaded\_image:

image = Image.open(uploaded\_image)

st.image(image, caption="Uploaded Image")

if st.button("Generate Caption"):

try:

with st.spinner("Generating caption..."):

response = client.models.generate\_content(

model="gemini-2.0-flash",

contents=["What is this image?", image])

st.subheader("Generated Caption:")

st.write(response.text)

except Exception as e:

st.error("Error generating caption")

st.title("Educosys YouTube Video Summarizer")

youtube\_url = st.text\_input("Enter YouTube Video URL")

if st.button("Summarize Video"):

if not youtube\_url:

st.warning("No YouTube URL Present!")

else:

try:

with st.spinner("Generating summary..."):

response = client.models.generate\_content(

model='models/gemini-2.0-flash',

contents=types.Content(

parts=[

types.Part(text='Can you summarize this video?'),

types.Part(

file\_data=types.FileData(file\_uri=youtube\_url)

)

]

)

)

st.subheader("Video Summary")

st.write(response.text)

except Exception as e:

st.error("Error generating summary")

RAG

Requirements.txt - langchain-chroma

langchain\_community

from dotenv import load\_dotenv

load\_dotenv()

from langchain.chat\_models import init\_chat\_model

from langchain\_community.document\_loaders import WebBaseLoader

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langchain.tools import tool

from langgraph.prebuilt import create\_react\_agent

from langchain\_chroma import Chroma

loader = WebBaseLoader(

web\_paths=["https://www.educosys.com/course/genai"]

)

docs = loader.load()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=200)

all\_splits = text\_splitter.split\_documents(docs)

print(all\_splits)

from langchain\_openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings(model="text-embedding-3-large")

vectorstore = Chroma(collection\_name="educosys\_genai\_info", embedding\_function=embeddings, persist\_directory="./chroma\_genai")

vectorstore.add\_documents(documents=all\_splits)

print(vectorstore.\_collection.count()) # Check total stored chunks

@tool

def retrieve\_context(query: str):

"""Search for info related to educosys genai course"""

try:

embeddings = OpenAIEmbeddings(model="text-embedding-3-large")

vector\_store = Chroma(

collection\_name="educosys\_genai\_info",

embedding\_function=embeddings,

persist\_directory="./chroma\_genai",

)

retriever = vector\_store.as\_retriever(search\_type="similarity", search\_kwargs={"k": 3})

print(f"Querying retrieve\_context with: {query}")

print("--------------------------------------------------------------")

results = retriever.invoke(query)

print(f"Retrieved documents: {len(results)} matches found")

for i, doc in enumerate(results):

print(f"Document {i + 1}: {doc.page\_content[:100]}...")

print("--------------------------------------------------------------")

content = "\n".join([doc.page\_content for doc in results])

if not content:

print(f"No content retrieved for query: {query}")

return f"No reviews found for '{query}'."

print("--------------------------------------------------------------")

print(f"Returning content: {content[:200]}...")

return content

except Exception as e:

print(f"Error in retrieve\_context: {e}")

return f"Error retrieving reviews for '{query}'. Please try again."

llm = init\_chat\_model("gpt-4o", model\_provider="openai")

agent\_executor = create\_react\_agent(llm, [retrieve\_context])

input\_message = (

"give me curriculcum of week 1 of educosys genai course?"

)

for event in agent\_executor.stream(

{"messages": [{"role": "user", "content": input\_message}]},

stream\_mode="values"

):

event["messages"][-1].pretty\_print()

* .env - GOOGLE\_API\_KEY (create on <https://aistudio.google.com/apikey>)

​​

from langchain\_google\_genai import GoogleGenerativeAIEmbeddings

embeddings = GoogleGenerativeAIEmbeddings(model="models/gemini-embedding-001")

Print embeddings

* for i, (doc, emb) in enumerate(zip(results["documents"], results["embeddings"]), 1): print(f"--- Chunk {i} ---") print("Text:", doc[:200], "...") # print first 200 chars of text print("Embedding length:", len(emb)) # number of dimensions print("Embedding sample:", emb[:20], "...") # print first 20 values print()

Multi-Agent Architectures

### **Supervisor Architecture**

* Customer Support Bot - Classifier, FAQ Bot, Sentiment Analyzer, Escalation Agent
* Code Generation Workflow - Designer, Coder, Tester, Explainer
* Loan Approval Automation - Document Parser, Validator, Risk Scorer, Decision Maker
* Interview Preparation Assistant - Resume Reviewer, Mock Interviewer, Feedback Generator
* E-commerce Product Uploader - Image Recognizer, Category Classifier, Description Writer, Price Suggester

### **Swarm Architecture**

* Research Assistant - Researcher, Critic, Summarizer, Fact-checker
* Story Writing Team - Plot Developer, Character Designer, Dialogue Writer, Editor
* Scientific Paper Generator - Literature Reviewer, Data Analyst, Writer, Reviewer
* Startup Pitch Creator - Business Modeler, Market Analyst, Financial Planner, Designer
* Debate Simulator - Government, Citizen, Expert, Journalist

Ollama

* **Local Execution**: Models run on your device, keeping sensitive data private and secure.
* **Cross-Platform**: Supports macOS, Linux, and Windows.
* **Multi-Model Compatibility**: Easily switch between LLaMA, Gemma, Mistral, and others.
* **Offline Use**: No internet needed after downloading a model.
* **Customizable**: Modify models and settings based on your hardware or specific needs.
* **Easy to Use**: Simple command-line interface, beginner-friendly.
* **Open Source**: Transparent and modifiable codebase.