PAGE No.
DATE / /

Agentic AI Frameworks

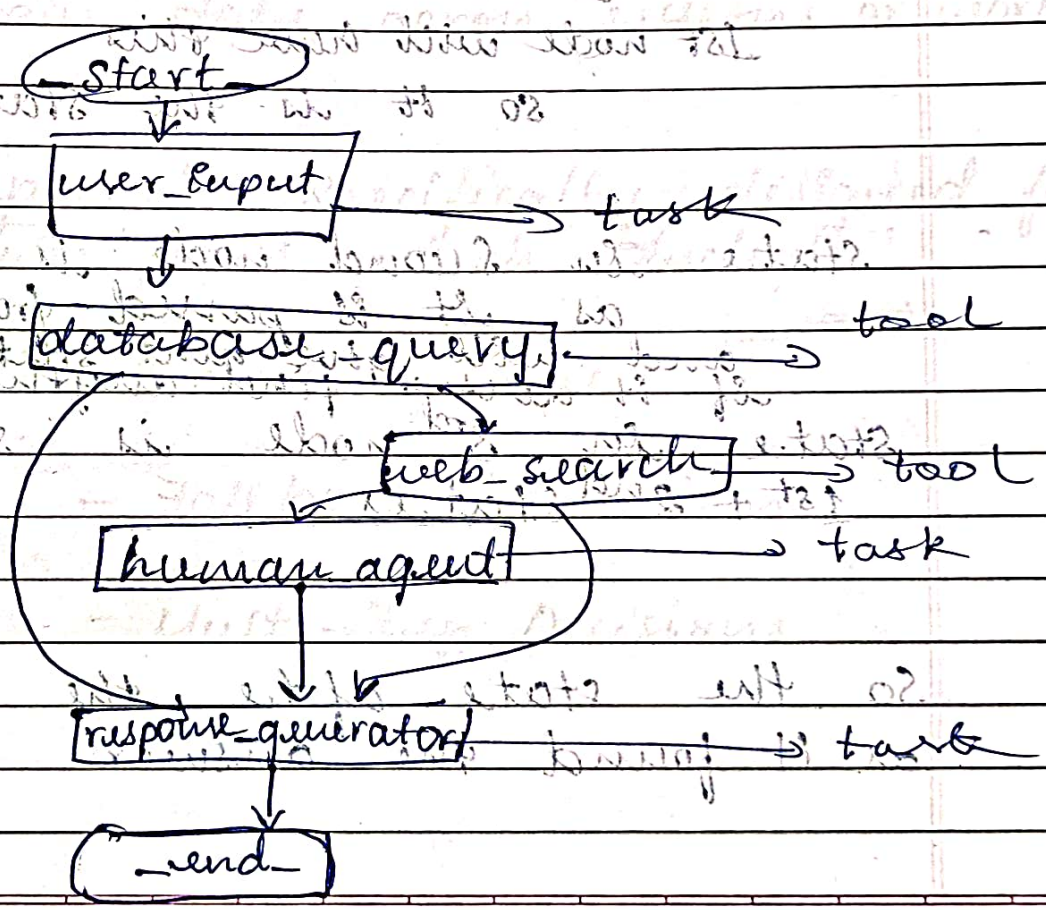
LangGraph

open-source framework for
building LLM-based application

Agentic AI, Langgraph helps us to build agents that can

- think
- make decisions

helps user design workflows in a graph like structure



In langgraph where a action or tool is being used its called node.

3 components of langg

- Node
- Edge → connection between 2 nodes
- State

What is state?

Ex:-

User: Find product information online

1st node will have this
so it is the state of 1st node

State in second node is the same as it is passed from node 1 and will give next state as dont know if it doesnt find anything

State in 3rd node is state of both 1st + 2nd states

So the state after the 3rd node as it found the answer

PAGE No.
DATE / /

State - contains all the info from all the nodes, the query has passed through

* Why so important?

So that the system can remember past actions, decisions and it can use that memory to guide features status

* Why is state management important?

It will help maintain the content of the conversation

can maintain state across sessions or workflow

* LangGraph is specifically designed for complex workflows which include:

- Branching logic
- Fallback Mechanisms
- Multi-Step Decisions

PAGE No
DATE / /

Branching logic means workflows can branchout into different parts based on different conditions.

Eg:- the agent cant find the answer in the database so it uses web search tool or human agent to find answer.

Fallback mechanism is like a backup plan, like it ensures the workflow doesnt fail completely, and continue by switching to a alternate process

Multi-Step Decisions means there are workflows which use multiple steps to reach a decision, each step depends on the outcome of the previous one.

PAGE No.
DATE / /

f. Important components of LangG also has
- State machine

To understand State better :-

Initial state :

state = {
"messages": [
{"role": "user", "content": "Hello what is Quantum Computing?"},
]
}

In Node 1 :

state = {
"messages": [
{"role": "user", "content": "Hello ~~what is quantum~~ ?"},
{"role": "db-query-tool", "content": "Hi, I am not sure about the answer"},
]
}

In node 2

State = {

PAGE No.
DATE / /

{ ——————— },
{"role": "web_search", "content": Quantum computing is a type of job ——"}

]
}

What is State Machine?

It decides what should be the next step

means user input is given
state machine sends it to the db_query_tool to find the answer
if found answer it sends it to the response_generator, if ans not found state machine sends to web_search_tool, then it goes on...

how does State machine know which node is the next node?

By using edges

every edge has a condition for the state machine to follow the path

state machine evaluates all the conditions of the edges after the node and then decides which edge to follow

Now Practical :-

storing in state

for

State machine

```
from typing import List, Dict
from langgraph.graph import StateGraph
from langchain_ollama.llms import OllamaLLM
```

llm

```
# Step 1 : Define State
class State(Dict):
    messages: List[Dict[str, str]]

# Step 2 : Define State machine
graph_builder = StateGraph(State)

# Init LLM
llm = OllamaLLM(model = "llama3.1")
# Define Chatbot function

def Chatbot(State: State):
    response = llm.invoke(state["messages"])
    state["messages"].append({"role":
    "assistant", "content": response})
    return {"messages": state["messages"]}
```

PAGE No
DATE / /

Add nodes and edges

State machine

```
graph_builder.add_node("chatbot", chatbot)
graph_builder.add_edge(START, "chatbot")
graph_builder.add_edge("chatbot", END)
```

Compile the Graph

```
graph = graph_builder.compile()
```

Now run but we need to take a flow for user input

user input

```
def stream_graph_updates(user_input: str):
# init the state with the user's input
state = {"messages": [{"role": "user",
                       "content": user_input}]}

for event in graph.stream(state):
    for value in event.values():
        # Print the assistant's response
        print("Assistant:", value["messages"][-1]["content"])
```

prints the assistant response

```
# Rami Chatbot in a loop
if __name__ = "__main__":
    while True:
        try:
            user_input = input("usr: ")
            if user_input.lower() in ["quit", "exit", "q"]:
                print("Goodbye")
                break
            stream_graph_updates(user_input)
        except Exception as e:
            print(f"error: {e}")
            break
```

runs this function

PAGE No.
DATE / /

How to Create Agents in Langchain?

- LLMs → responsible for understanding user query and generate responses

- tools → are internal functionalities
  If LLM doesnt know the answer, it can use a tool to answer the question

Practical

→ from langchain_community.tools import WikipediaQueryRun

from langchain_community.utilities import WikipediaAPIWrapper

& to fetch from wikipedia

api_wrapper = WikipediaAPIWrapper(top_k_results=1, doc_content_chars_max=300)

wiki_tool = WikipediaQueryRun(api_wrapper=api_wrapper)

→ from langchain_openai import ChatOpenAI

llm = ChatOpenAI(temperature=0, api_key="", model="gpt-4o-mini")

→ Tools = [wikitool]

tool binding

llm_with_tools = llm.bind_tool(tools)

PAGE NO.
DATE / /

Tool Calling

result = llm_with_tools.invoke("Hello World!")
result
result.content

→ Hello! how can i assist you today?

lets do in LangG

→ from langgraph.prebuilt import create_react_agent

agent_executor = create_react_agent(llm, tools)

→ from langchain_core.messages import HumanMessage

# first up, let's see how it responds when there's no need to call a tool:
response = agent_ex.invoke({"messages": [HumanMessage(content="hi")]})
print(response["messages"][-1].content)

→

→ if change content to "what is agentic AI", the llm doesnt have ans, but the tool is used for it

How to create Custom Tools in langG

→ from IPython.display import Image, display
" langgraph.graph " StateGraph, START

To display graphs

" langchain_openai import ChatOpenAI

import requests

from langchain_core.messages import SystemMessage, HumanMessage

from langgraph.graph import MessageState

from langg.prebuilt import ToolNode, Tools condition

from typing import Annotated
from typing_extensions import TypedDict
from langg.graph.messages import add_messages

here all our messages are getting stored
→ class State(TypedDict):
messages: Annotated[list, add_messag

lets create a DuckDuckGo search tool
→ from langchain_community.tools import DuckDuckGoSearchRun
search = Duck——()
search.invoke("Obama's first name?")

→

lets put this tool in 1 function

→ def search_duck~~~ (query: str):

search = Du~~~ ()

return search.invoke (query)

lets create a basic tool

→ def multiply (a: int, b: int) → int:

return a * b

This tool performs multiplication

our llm

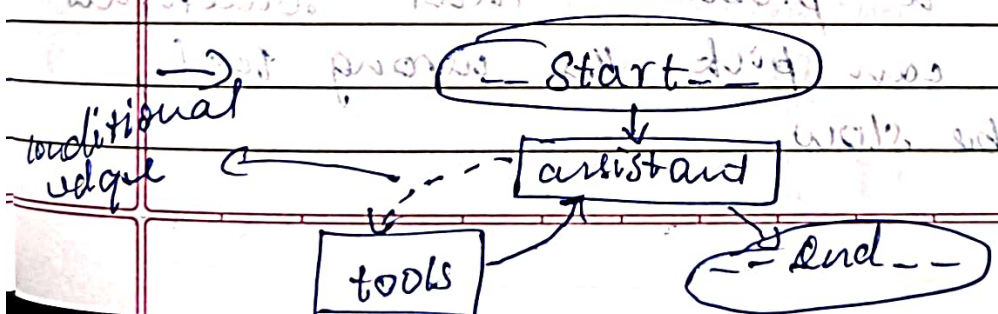
→ llm = ChatO~~~

→ tools = [search_duckduckgo, add, multiply]

llm_with_tools = llm.bind_tools (tools)

we are binding our llm to these tools

```
→ def Chatbot (State: State):
        return {"messages": [llm_with_tools.invoke
                                (state["messages"]
                                )]}


→ from langgraph.prebuilt import ToolNode,
                                 tools_conditions

graph_builder = StateGraph(State)

# Define Nodes
graph_builder.add_node("assistant", chatbot)
graph_builder.add_node("tools", ToolsNode(tools))

# define edges
graph_builder.add_edge(START, "assistant")
              .add_conditional_edges("assistant",
                                     tools_condition)
graph_b...  .add_edge("tools", "assistant")
react_graph = graph_builder.compile()


# lets Visualize in a graph

→ display(Image(react_graph.get_graph().
                draw_mermaid_png
                ()))
```



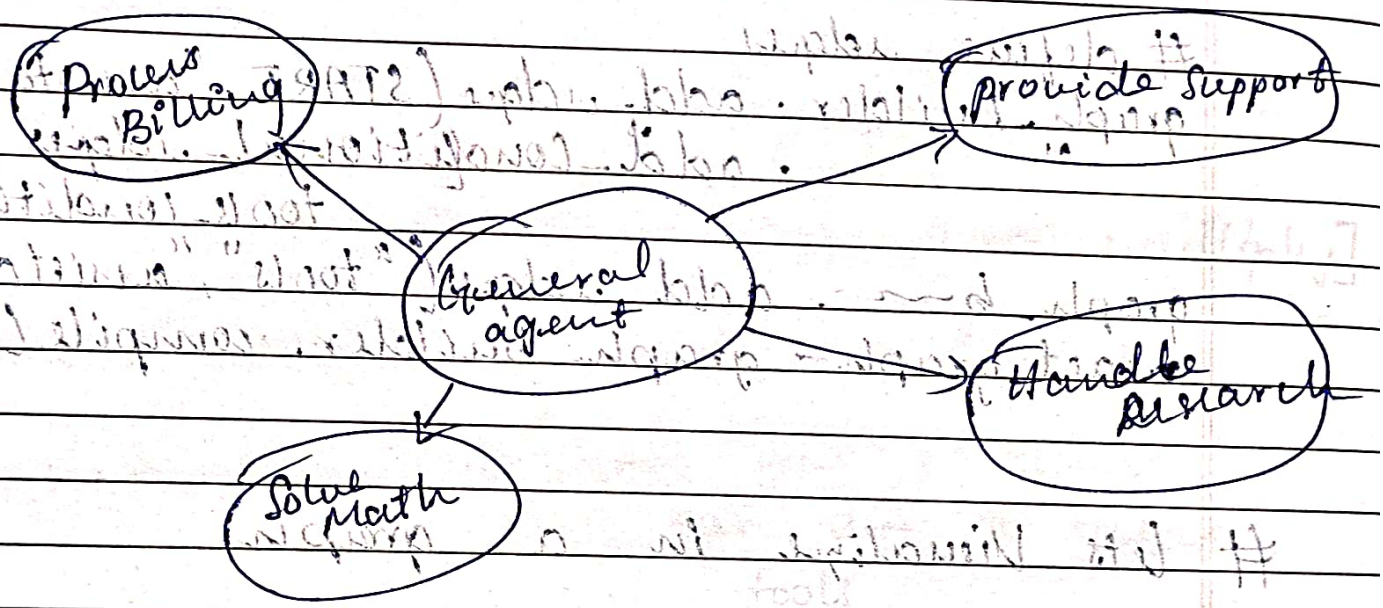
PAGE No.
DATE / /

* This is how we build our own tools for langgraph

* How to create multi-Agent System in langgraph?

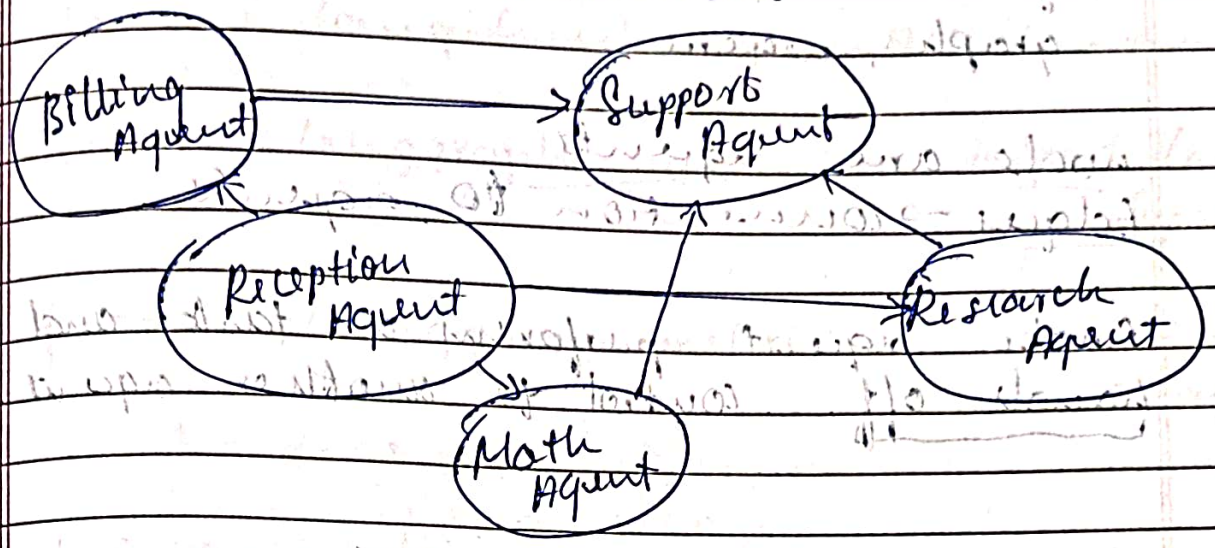
Previously we used single agent and gave multiple tools to the agent

Single Agent System handling all tasks



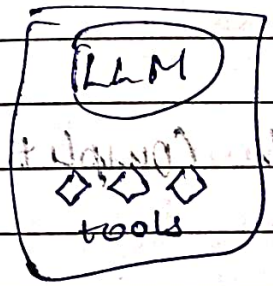
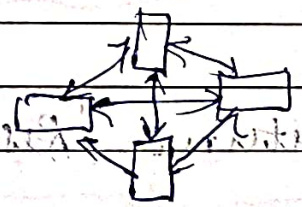
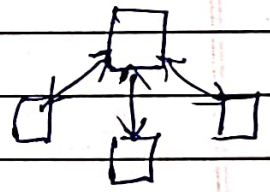
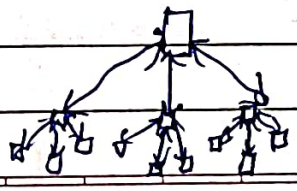
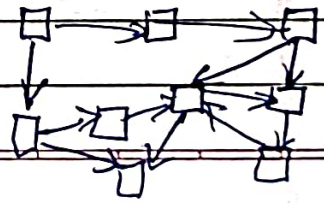
* There is a problem that sometimes the agent can pick the wrong tool, can also be slow.

To Solve these Problems Multi-Agents System is Introduced



These agents communicate with each other to solve the user query

There are different ways agents can interact:

Single Agent	Network	Supervisor
		
Supervisor (as tools)	Hierarchal	custom
		

PAGE No.
DATE / /

Multi-agent System as graphs

Agents can be visualized as a graph, where

node are agents
Edges → connection to agents

Each agent performs a task and then hands off control to another agent.

Example

⇒ def research_agent (state):
    if "math" in state["task"]:
        return Command(
            goto = "math-agent",
            update = {"info": "Passing to Math Expert"}
)
    else:
        return "Research Completed"

hands-off to math agent ⟵ goto = "math-agent"

lets make a system in Supervisor way.

→ from langchain_openAI import

from langgraph supervisor import create_supervisor

from langgraph.prebuilt import create_react_agent

init model

model = ChatOpenAI ( )

tools

→ def ~~search~~ duckduckgo (query: str):

we did this prev

def add ( )

def multiply ( )

PAGE No.
DATE / /

create different agents

→ math_agent = create_react_agent(
model = model,
tools = [add, multiply],
name = "Math expert",
prompt = "You are a math expert. Always use one tool at a time."
)

→ research_agent = create_react_agent(
model = model,
tools = [duckduckgo],
name = "research expert",
prompt = "You are a world class researcher with access to web search. Do not do any math."
)

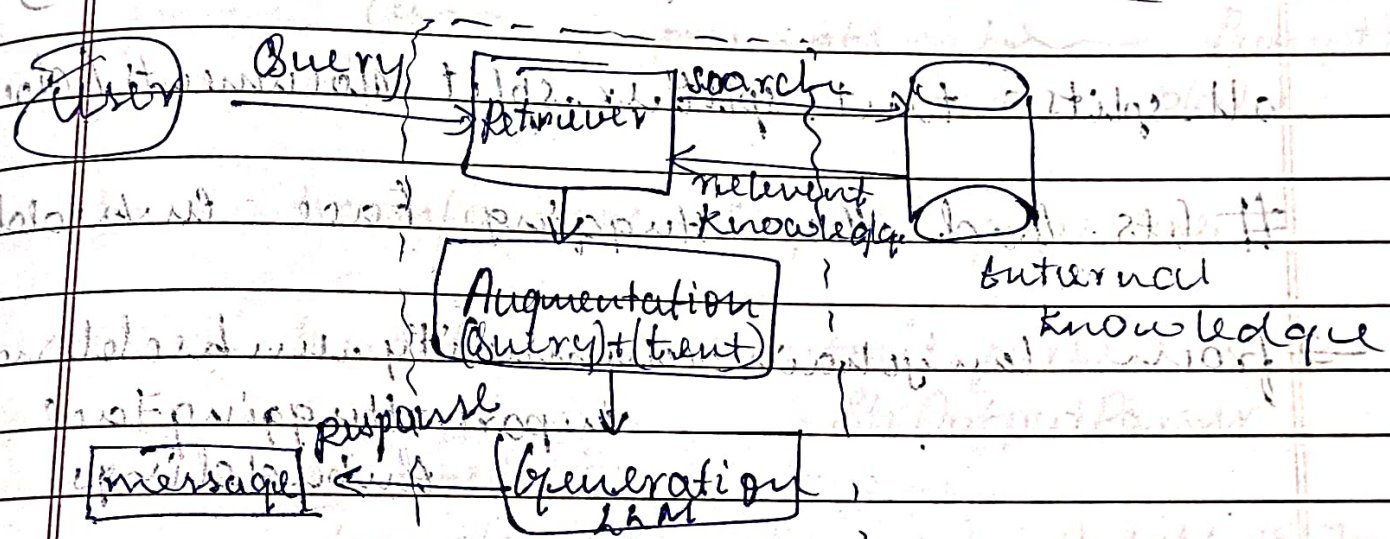
now lets create a Supervisor Workflow

workflow = create_supervisor(
[research_agent, math_agent],
model = model,
prompt = (
"you are a team supervisor managing a research expert and a math expert."
"For current events, use research_agent."
"For math problems, use math_agent."
))

Compile and Run

```
app = workflow.compile()

result = app.invoke({
    "messages": [
        {
            "role": "user",
            "content": "What is quantum?"
        }
    ]
})
```

RAG with LangGraph



PAGE No.
DATE / /

→ from langchain-community.document_loaders import UnstructuredURLLoader

urls = ['htt ———————————']
loader = UnStructuredURLLoader(urls=urls
docs = loader.load()

Text Splitting

→ from langchain_text_splitters import RecursiveCharacterTextSplitter

text splitter = R ———————— (chunk_size=1000, chunk_overlap=200)

all_splits = text_splitter.split_documents(docs)

lets load the Hugging Face embedding

→ from langchain_community.embeddings import HuggingFaceEmbeddings

embeddings = HuggingFaceEm ———— ()

vector = embeddings.embed_query("Hello, world!")

Lets Create a Vector D.B

from langchain_chroma import Chroma
from langchain_core.documents import Document

vectorstore = Chroma.from_documents(documents=all_splits, embedding=HuggingFaceEmbeddings())

→ from langchain_huggingface import HuggingFacePipeline

from langchain_core.prompts import PromptTemplate

from transformers import pipeline

from langchain_core.output_parsers import StrOutputParser

import torch
from transformers import AutoModelForCausalLM, AutoTokenizer

model_id = "tiiuae/falcon-7b"

PAGE No.
DATE / /

```
text_gen_pipeline = pipeline(
        "text-generation",
        model = model_id,
        model_kwargs = {"torch_dtype": torch.
                                bfloat16},
        max_new_tokens = 200,
        device = 0,
        temperature = 0.7,
        top_k = 50
    )

llm = HuggingFacePipeline(pipeline = text_gen_
                                pipeline)
```

→ from langchain_core.prompts import
PromptTemplate

```
template = """ Use the following
{context}

Question: {question}

Helpful Answer: """
prompt = PromptTemplate.from_template(
                template)
```

PAGE NO
DATE / /

→ from typing_extensions import List, TypedDict

Define State for application

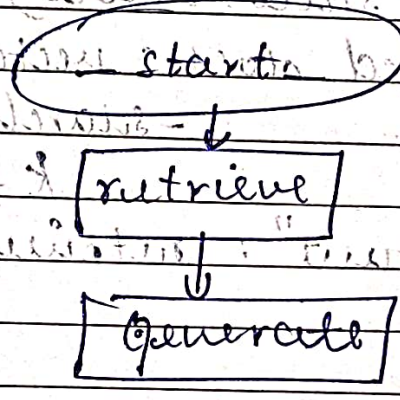
class State (TypedDict):
    question: str
    context: List[Document]
    answer: str

Now we create our Retriever function

→ def retrieve (state: State):
    retrieved_docs = vectorstore.similarity_search (state["question"], k=1)
    return {"context": retrieved_docs}

→ def Generate (state: State):
    docs_content = "\n\n".join(doc.page_content for doc in state["context"])
    messages = prompt.invoke({"question": state["question"], "context": docs_content})
    response = llm.invoke(messages)
    return {"answer": response}

```
from langgraph.graph import START, StateGraph

# Compile application and test

graph_builder = StateGraph(State).add_sequence([retrieve, generate])
graph_builder.add_edge(START, "retrieve")
graph = graph_builder.compile()

# Visualize the graph
display(Image(graph.get_graph().draw_mermaid_png()))
```

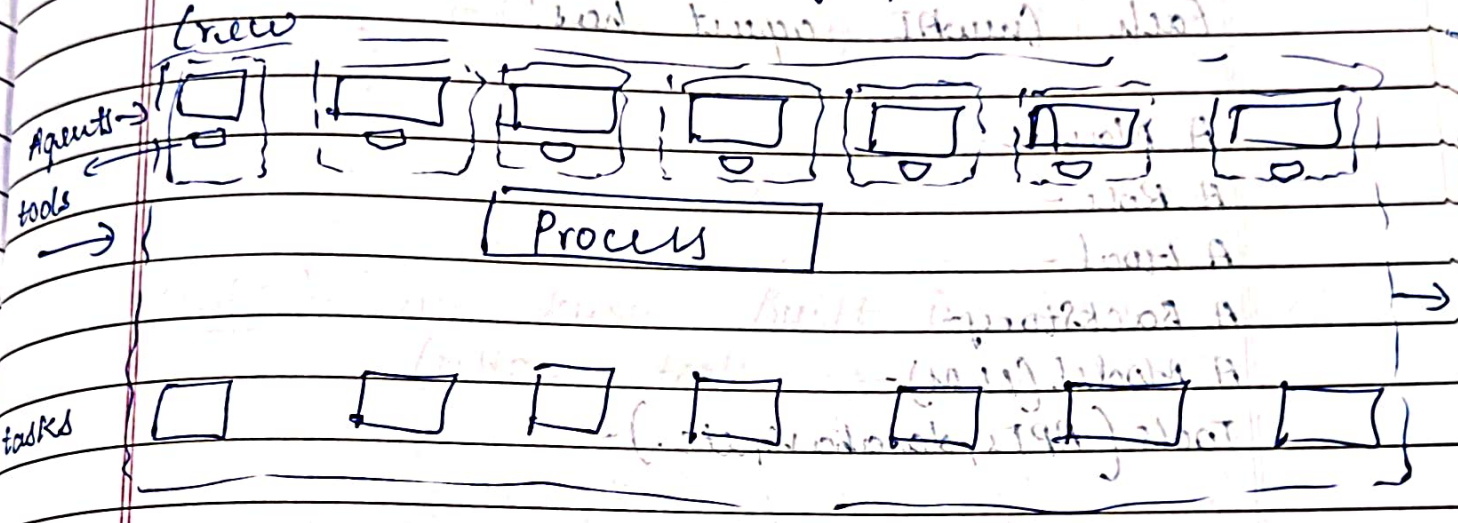



```
# test
response = graph.invoke({"question": "what is langgraph?"})
print(response["answer"])
```

Crew AI

In CrewAI we have multiple agents, we have list of task they perform, and we have process,

the task of the process is to assign task to the agent, order of the task, and more is handled by process



Example:

AI powered trip advisor:

- Flight booking assistant
- Hotel Recommendation Assistant
- Sightseeing Planner Agent
- Food Advisor Agent

Trip Planner Agent

* Best flights and hotels
* Top tourist attractions
* Local food recommendation
* Daily schedule

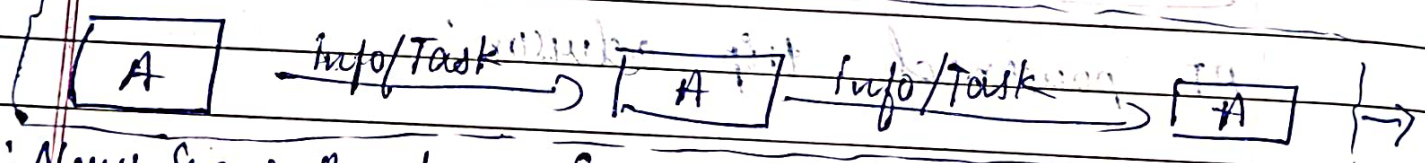
How CrewAI Works ?

Agents
tools
Tasks
Process

Each CrewAI agent has:

- A Name -
- A Role -
- A Goal -
- A BackStory -
- A Model (LLM) -
- Tools (APIs, database, etc.) -

Ex:-



Name: News Scraper Agent | Summarizer Agent | Writer Agent
Role: Web Crawler | Content Analyst | Report Generator
Task: Fetch the latest News | Extract key Insights from the news | Generate a Structured summary

Ex:- In Code

→ news_scrapper = Agent(
name = "News Scrapper",
role = "Web Crawler",
goal = "Fet",
backstory = "An expert in web crawling and retrieving news articles",
llm = ChatOpenAI(model_name = "gpt-4"),
tools = [news_scrapper_tool]
)

Tools :- we know Built in tools and Custom tools

Tasks :- Tasks are the specific actions that AI agents must complete.

→ fetch_news_task = Task(
description = "Fetch AI news from online sources.",
agent = news_Scrapper
)

and more

PAGE No.
DATE / /

Process :-

* Workflow management system

* means it will see which task should be done 1st and which task to be assigned to which agent

Ex :-

Task : Fetch news
Assigned agent : News Scraper Agent
Process Type : Step 1 (Start)

Task : Summarize key points
As
Process Type : Step 2 (After fetching News)

Task : Write the final
As
Process Type : Step 3 (Final Output)

DATE / / /

Types of Execution in a Process:

* Sequential Execution (Step-by-Step)

Task 1 → Task 2 → Task 3

* Parallel Execution (Multiple Tasks at Once)

Task 1 & task 2 → Task 3

Fetch news and summarize at same time → Write Report

Practical:-

News Summarization App

```
#tool
→ from crewai tools import WebsiteSearchTool
import os
web_search_tool = WebsiteSearchTool()

#llm
→ from langchain_openai models import ChatOpenAI
os.environ["OpenAI_API_KEY"] = "~~~~~~~~"
llm = ChatOpenAI(~~~~~~~~~~)
```

PAGE No.
DATE / /

→ from crewai import Agent

```
researcher = Agent(
    role = 'Market Research Analyst',
    goal = 'Pro
    backstory = "",
    tools = [web_search_tool],
    verbose = True
)

writer = Agent(
    role = 'Content Writer',
    goal = '
    backstory = '
    tools = [ ], # No tools needed; writes based on research summary
    verbose = True
)
```

→ from crewai import Task, Crew

```
research = Task(
    description = 'Search the web for the latest AI trend and provide a summarized report.'
    expected_output = 'A summary of top 3 tr
    agent = researcher
```

```
write = Task(
        description = '_____',
        expected_outcome = 'A well structure_____',
        agent = writer,
        output_file = 'blog-posts/new_post.md'

# Assemble the Crew

=> Crew = Crew(
        agents = [researcher, writer],
        tasks = [researcher, write],
        verbose = True
        Planning = True  # Enable AI planning feature
)
crew.Kickoff() # Execute the task
```

2nd Crew

AI travel planner using CrewAI, langchain, Streamlit

PAGE No.
DATE / /

→ from CrewAI import LLM
from langchain_ollama.llms import OllamaLLM

```
llm = LLM(
    model = "ollama/llama3.2",
    base_url = "http://localhost:11434"
)
```

tool

→ from crewai.tools import tool
from langchain_community.tools import DuckDuck
import json

```
@tool
def search_web_tool(query: str):

    search_tool = Du———(num_results=10,
                        verbose=True)
    return search_tool.run(query)
```

Agent

→ from crewai import Agent

```
guide_expert = Agent(
    role = "City local Guide Expert",
    goal = "————————",
    backstory = "————————",
    tools = [search_web_tool],
)
```

verbose = True,
max_iter = 3,
llm = LLM(model="ol ",),
allow_delegation = False,
),

Agent

→ location_expert = Agent(

)

Agent

→ Planner_Agent = Agent(

task

→ from datetime import datetime
from crewai import Task

location_Task = Task(
description = "

expected_output = "

agent = location_expert,
output_file = 'city_report.md',
)

more tasks
= Task(

= task(

But since we are doing in Streamlit and there must be in function

def location_task(agent, from_city, destination_city, date_from, date_t
return task(
description = f"""

{from }
{ }
{ }
)

```
def guide_task(
    return task(
    )

def planner_task(
    ret
    )

# create variable using these functions
→ location_task = location_task(
        location_expert,
        from_city,
        destination_city,
        date_from,
        date_to,
    )

→ guide_task = guide_task(
        guide_expert,
    )

→ planner_task = planner_task(
    )
```

```
from crewai import Crew, Process

crew = Crew(
agents = [loc——, qu——, pl——],
tasks = [loc——, q——, p——],
process = Process.sequential,
full_output = True,
share_crew = False,
verbose = True
)
result = crew.kickoff()
```

Now with these create a Streamlit app

→ `from ——`

} all the imports from modules from crew files and funcs

```
st.title("AI ——")

st.markdown("""

""")
```

```
# User Inputs

from_city = st.text_input ("From City", "India")
destination_city = st.text_input ("Des City", "Rome")
date_from = st.date_input ("_____")
date_to = st.date_input ("_____")
interests = st.text_area ("_____")

# Button to run CrewAI

if st.button ("Generate Travel Plan"):
    if not from_city or not destination_city
        or not d___ or not d___ or not
        in___ :
        st.error ("Please fill all")

    else :
        st.write ("... Please wait.")

        # init tasks
        loc_task = location_task (location_expert,
                                  f_____)
        guide_task = guide_task (guide_expert,
                                 _____)
        plan_task = planner_task ([loc_task,
                                   guid_task],
                                  planner_expert, ____
                                  _____)

        # Define Crew
        Crew = Crew(
```

```
agents = [lo___, g___, pl___],
tasks = [loc_task, guide_task, plan_
                                task],
process= Process.sequential,
full_output = True,
verbose=True,
)

# Run Crew AI
result= crew.kickoff()

# Display Results
st.subheader("Your Plan")
st.markdown(result)

# Ensure result is a string
travel_plan_text = str(result)

st.download_button(
        label = "Download"
        data= Travel_plan_text,
        file_name= f"Travel_Plan_
                    {destination_city}.
                    txt",
        mime = "text/plain"
)
```

Agentic Framework

Agno

* for building multi-agent systems with memory, knowledge and Reasoning

* Use Agno to build 5 levels of Agentic Systems:

1: Agents with tools and instructions
2: " " Knowledge and Storage
3: " " memory and Reasoning
4: " Teams that can reason and Collaborate
5: Agentic workflows with state and determinism

* Agno will help you build best in class, high-performatic agentic systems, saving you hours of Research and boilerplate.

Natively multi-modal: Agno agents are natively multi-modal, they accept text, image, audio, and video as input and generate text, image, audio and video as output

LangGraph is not Multi-modal by default

Ex:- building an agent that ans ques about image
1. use multi-modal model like GPT-4 vision
2. Write code to accept the image
3. need code to convert
4. Create a LangG node to run that logic

PAGE No.
DATE / /

* Issue : We have to write the logic ourself

* In other hand Agno does it Default

→ from agno.agent import Agent

# Define Agno agent
agno_agent = Agent(
model = OpenAIChat(id="gpt ~~~~"),
description = "Agno Q&A agent",
markdown = False
)

to use key and api

→ agno_agent.print_response("What is India", stream = True)

→ ~~~~~~~~~~~~~~~~

PAGE No.
DATE / /

→ from dotenv ———
import os

load_dotenv(dotenv_path='/———')
api_key = os.getenv("OpenAI_API_key")
os.environment["OPENAI_API_KEY"] = api_key

→ from agno ——— import agent

agno_agent = Agent(
model = OpenAIChat(id='gpt-4o')
description = "you are a cheerful AI pal who loves a good chat!",
markdown = True
)

agent.print_response("———", stream=True)

→ ———

* Now lets use tools

Pre-defined tools
you can visit agno website
go to tools option
and check available
toolkits you will
see alot.

Example :

→ from agno.agent import Agent
from agno.tools.duckduckgo import Duc~~

agent = Agent (tools=[Du~~()], show_tool_calls
= True)

agent.prin~~ ("~~", markdown=True)

→ ~~

→ from ag~~.tools.googlesearch imp~~ Gou~~
agent1 = agent (
model = OpenA~~ ("~~"),
tools = [GoogleSearchTools(
stop_after_tool_call_tools
= ["google_searc
show_result_tools = ["google-search"],
)
],
show_tool_calls = True,
)

agent will stop once tool output you get

will show the response

Agent will show with tool is used

* If you dont mention any model the default model OpenAI will be used, so you have set your OpenAI API key

* Custom Tools

→ from date.time import datetime
from Agno.agent import Agent

```
def get_today_date() -> str:
    today = datetime.now().strftime("Today is %B %d, %Y")
    return today
```

This tool will return the current date, and time

→ agent = Agent(
    tools = [get_today_date],
    show_tool_calls = True
    mem ——— = True)

PAGE No.
DATE / /

Memory

3 types for Agno

- Session Storage : It stores the chat history both user and agent

- User Memories : Whatever the agent has learnt about the user all those will be saved in user memories

  means only about the user

- Session Summaries : The agent can store a condensed representation of the session, useful when chat histories gets too long, this is called "Sumary" in Agno

→ from agno.agent import Agent
from agno.memory.v2.memory import Memory
" " " .db.sqlite " SqliteMemoryDb
from agno.storage.sqlite import SqliteStorage
from rich.pretty import pprint

PAGE No.
DATE / /

* are doing User memory

everything will be stored in SqliteMemoryDb

→ we create Db

→ db_file = "———"

user_id = "Aarohi"

```
memory = Memory(
    db = SqliteMemoryDb(table_name = "user_memories",
                        db_file = db_file),

Agent = Agent(
    model = Op——("gpt 40"),
    description = "———",
    memory = memory
    storage = SqliteStorage(table_name = "agent_sessions",
                            db_file = db_file),
    enable_user_memories = True
    add_history_to_messages = True
    num_history_runs = 3,
    session_id = "my_chat_session",
    markdown = True,
)
```

PAGE No.
DATE / /

→ agent.pr_respon("_____", user_id=user_id)

→ _____

What is Model Content Protocol MCP?

How things Worked Before MCP?

Example: lets say we want ~~to~~ our AI agent to check today's weather

User: what is the weather in delhi today?

How we usually handle it without MCP is:

```
api → weather = requests.get("http~~~~~~~~~~")
      data = weather.json()
      print(data["temperature"])
```

output: its sunny in Delhi today with a temperature of 22°C

What is the Problem here?

suppose Weather Company - API modified Temperature → temp

we will get issue with earlier code as temperature variable is not present and to work we need to re-write the code

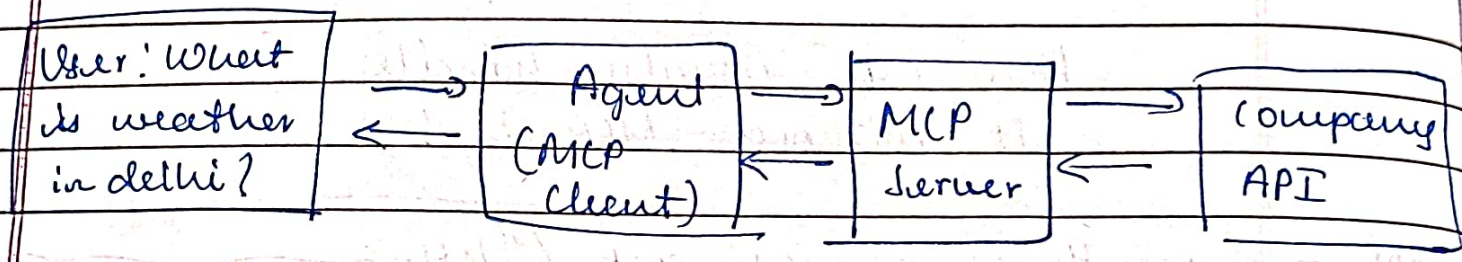
DATE / /

* How this problem is solved with MCP?

MCP Client
↓
Agent
on our machine
or cloud

MCP Server
↓
will run
on the api company
- weather MCP server

* How MCP Works?



all the MCP servers understand a single format ~~with~~ which is JSON-RPC

So the MCP Client convert the msg into JSON-RPC before sending it to MCP Server

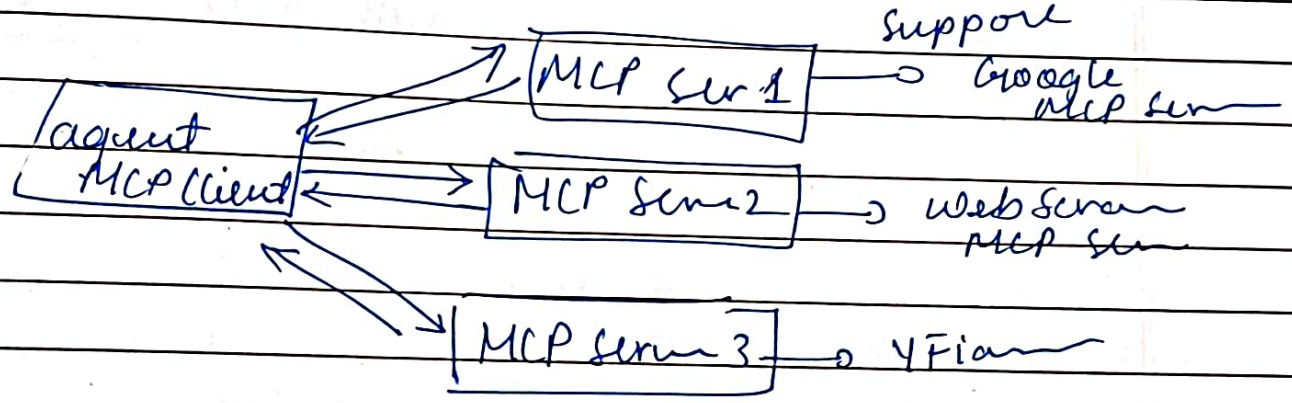
And the MCP Server converts that JSON-RPC to a format that the company api accepts

And same way reversed, is sent through and to the user, as MCP Client has agent the data reply will be user friendly.

So in this way if the Company api changes something in their code, we wont face any problem, as we only write the MCP Client code to connect to the MCP Server

* Benefits of MCP:

1. Universal Communication Standard:



we can connect to multiple company MCP servers through One agent and agent must have MCP Client

2. No Code Changes When Backend Changes

3. Works with any MCP-Compatible Tool -

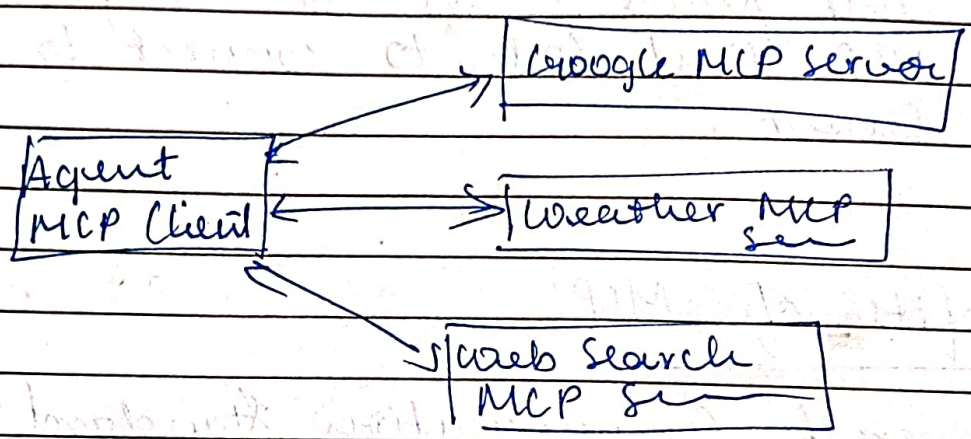
Example

User: Send me an email about today's weather and top 3 news headlines

gmail api

weather api

news api



Agentic RAG

→ RAG + Agentic AI

* RAG Vs Agentic RAG

Ex:-
User: whats the Best laptops for AI under 80K in India?

Basic RAG App:

RAG will search a pre-stored knowledge base (vector DB) and then writes an answer based on what it finds

Agentic RAG App:

Step 1: Plan the Task
a- Search ecommerce sites
b- Filter out low-performance models
c- Compare specs: CPU, GPU, RAM
d- Choose and recommend the best o

So the agent Agent is thinking and Reasoning

PAGE No.
DATE / /

* What is Knowledge?

is domain-specific info that the agent can search at runtime to make better decisions (Dynamic few-shot learning) and provide accurate responses (Agentic RAG)

* there are different kinds of knowledge base like text, PDFs, JSON, CSV, Arxiv, etc.

lets do practical on text knowledge base

→ from agno.knowledge.text import TextKnowledgeBase

from agno.vectordb.pgvector import PGVector

this

By this will store embeds in postgres SQL

```
knowledge_base = TextK___(
    path=' ___',
    vector_db = PGVector(
        Table_name = "Text_Documents",
        db_url = "p___ ___",
    ),
)
```

have to put your own

→ agent = Agent(
    knowledge_base = knowledge_base,
    search_knowledge = True,
)

PAGE No.
DATE / /

* To download and use PostGresSQL dB

→ psql --version

* install postgres

* * * Remember password and Portnumber

set the path in environment variables

* Create a DB

→ psql -U postgres
→ Give password : ——

* And to create a user AI and pass ai

→ CREATE USER ai WITH PASSWORD 'ai';

→ CREATE DATABASE ai;

→ now grant this user permission to this db

→ GRANT ALL PRIVALAGES ON DATABASE ai TO a

PAGE No
DATE / /

* Have to install vector extension for postgres.

So march the internet to do that on Chatgpt

Download and load papers into VectorDB

→ agent.knowledge.load(recreate=False)

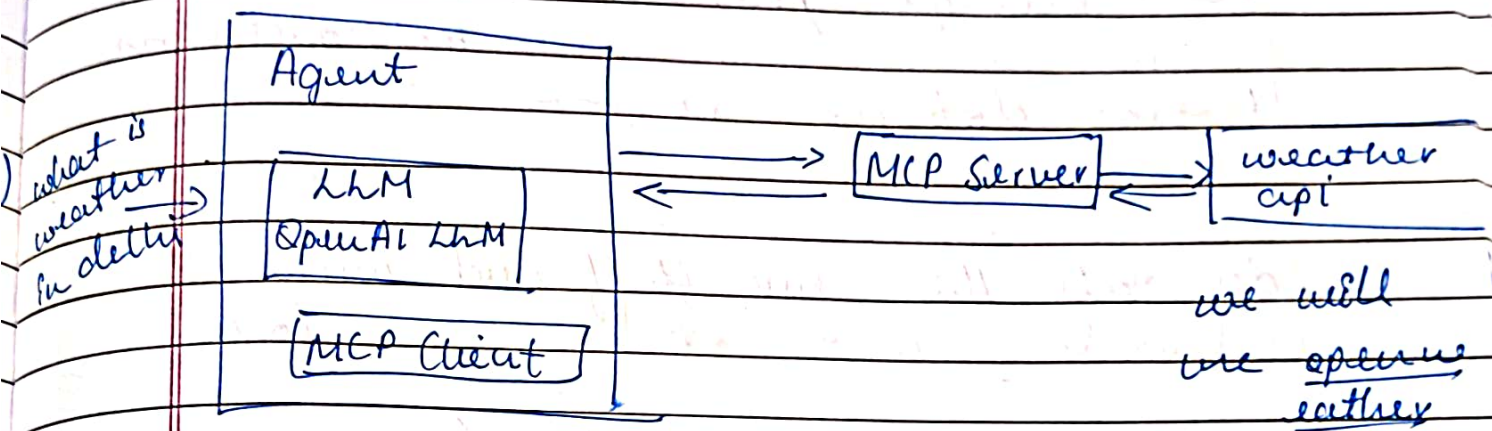
→ user_question = input("\n ask ques from bard!"

→ agent.print_response(user_questions, user_id = "user_1", stream=True)

→ ——————

Building & Linking Multiple MCP Servers

In this tutorial:



we will use openweather api

for using this server visit MCP git repo search openweather api and clone the repo in local machine and follow the steps

* Pre Req:
GO 1.30 for your system
Openweather api key

following all those steps and a MCP server is installed in your repo

PAGE No.
DATE / /

→ from asyncio
from os
from langchain_openai
from langchain_mcp_adapters.client import MultiServerMCPClient
from langgraph.graph import StateGraph
from langgraph.prebuilt import ToolNode, tools_condition
from dotenv import

* do all the env stuff and write the code

→ Client = MultiServerMCPClient(
{
"weather": {
"transport": "stdio",
"command": "t://c",
"args": [],
"env": {"OWM_API_KEY": owm_key
}
}

our mcp server location

tools = await client.get_tools()

model = ChatOpenAI(model='g', api_key = api_key)

PAGE No.
DATE / /

Bind tool to model

→ def call_model (state : MessageState):
    response = model.bind_tools(tools).invoke
        (state["messages"])
    return {"messages": response}

Building the langG flow

→ builder = StateGraph(MessagesState)
builder.add_node('call_model', call_model)
" " " ('tools', ToolNode(tools))
" .add_edge(START, "call_model")
" .add_conditional_edges("call_model,
    tool_condition)
" .add_edge("tools", "call_model")
" .compile

Running the Graph

result = await graph.ainvoke({"messages":
    "whats the weather in
    Delhi today?"})
print(result["messages"][-1].content)

Now in a terminal go to the MCP server exe and run it

→ .\mcp-weather.exe

* Now the server is running

→ if __name__ == "__main__":
    mcp.run(transport="stdio")

Now to run the code open another terminal
and run the py file

and we are getting the reply from MCP Server

* This was Single Server now lets connect with multiple

our code will remain same

just change Client

```
Client = MultipleClient(
    {
        "weather": {
            },
        "calculator": {
            },
        "WebSearch": {
            }
    })
```

for using these tool go to git and run those servers locally

Building Custom MCP Server With LangG

all this mcp must be done in a seperate file

→ from mcp.server.fastmcp import FastMCP

```
mcp = FastMCP("Math")
@mcp.tool()
def add(a: int, b: int) -> int:
    return a+b
```

decorator when you want to make this function as tool for mcp

this function is registered as tool for this mcp server

* add more functions and add the decorator above

```
@mcp.tool()
def multiply(
```

* Run this server in a term

PAGE No.
DATE / /

* Now the mcp client file

→ #Same all import as prev

→ # do all the .env , key and stuff

→ model = Ch— (model="—", ap—
——)

client = Mut— (
{
"math": {
"command": "python",
"args": ["E:/g— .py"],
"transport": "stdio",
}

full path to file of mcp

this method will work when both mcp, client are on same machine

—: {
—
}
—— {
}
——

})

→ Same Rest all code as prev code

→

If What if the mcp server is on Remote location

→ Client = M—— (

{ "math" : {
"transport" : "streamable-http",
"url" : "http://127 ——/mcp"
}
}
)

Url of mcp server

→ if __name__ = "__main__":
mcp.run(transport = "str——")

Build a Custom MCP Server on the Cloud

* To build
  - Set up a new VPS
  - Install required packages
  - Write the custom MCP Server code on the VPS
  - Run the MCP Server in the Cloud
  - Connect a local MCP Client using langly to MCP Server on Cloud

* Set ur VPS on Hostinger

* take the SSH acces key

* Now use PuTTY app to login to our remote computer using SSH and password

  we chose ubuntu virtual machine

* Create a virtual environment

PAGE No.
DATE / /

* install python and all packages to run the mcp server

* The previous mcp code we wrote we want to send to the server

so open powershell and from there cd ~ to your file directory

Now command to send this file to cloud

→ pscp custom_server.py root@31.97.233.200:/opt/mcp-server/

(custom_server.py: our file in local machine)

(root@31.97.233.200: ip address of cloud)

(/mcp-server/: inside this folder)

* Now in the ~~ts~~ putty

→ python ~~custom_mp.py~~ custom_server.py

Now our server is running

PAGE No.
DATE / /

* Now our mcp client setup

Similar code to last ~~c~~ code

but change

→ Client = M———(

{

"math": {

transport: "Streamable_http",

"url": "http://———/mcp"

}

}

)

ip address of our cloud

→ if __name__ = "__main__":

asyncio.run(main())

and run the file

it works

How to Use MCP Servers with Agno

we know how install servers from Git

Script for server

→ imp asyncio
os
from dotenv
from agno.tools.mcp import MCPTools
from agno.agent import Agent
from agno.models.openai import OpenAIC

load_dotenv()
async def main():
Script for uvo

agent = Agent(
model = ,
tools = [
MCPTools("python -m mcp-server_calc
"),
],
description = " ",)

to run the file

PAGE No.
DATE / /

```
agent.print_response("What is 12* 8?"
                     ,stream=True)

if __name__ = "__main__":
        asyncio.run(main())
```

* run this file
  and it works

* For multiple Servers
  install the servers

and in code

```
agent = Agent(
    model = ————————,
    tools = [
        MCPTools("py ————————"),
        MCPTools("————————"),
        MCPTools("————————"),
    ],
    description = "————————",
    markdown = True,
)
```

→ Forum the servers

And for custom ones

write code for custom server
like math mcp tool
like we did previously

~~and run the server~~
no need to run the server
the client code will
run it
automatically

Code or client side

→ agent = ———(
———
———

tools = [
MCPTools("———"),
"
"
("———"),
("python math server.py"),
]
———
———
)

our
custom
server

* how about the cloud mcp server

* same way we did the server running
on the virtual machine
follow those same steps

PAGE No.
DATE / /

and run the server on Cloud

Now in Client Code

→ math_server_url = "http://_____"

```
async def run_agent(message: str):

    multi_mcp_tools = MultiMCPTools(
        commands=[],
        urls=[math_server_url],
        urls_transport=["streamable-http"]

    # Connect to all MCP servers
    await multi_mcp_tools.connect()

    # Create agent
    agent = Agent(
        model = _____
        tools=[multi_mcp_tools],
        descr___ = _____,
        markdown = True,
    )

    # ask something
    print("\n Agent response:\n")
    await agent.aprint_response(message,
                                stream=True)
```

```
# Close connections
await multi_mcp.close()

if __name__ == "__main__":
    asyncio.run(
        run_agent("Calcu[illegible]
        [illegible]")
```

* Run the file it must work