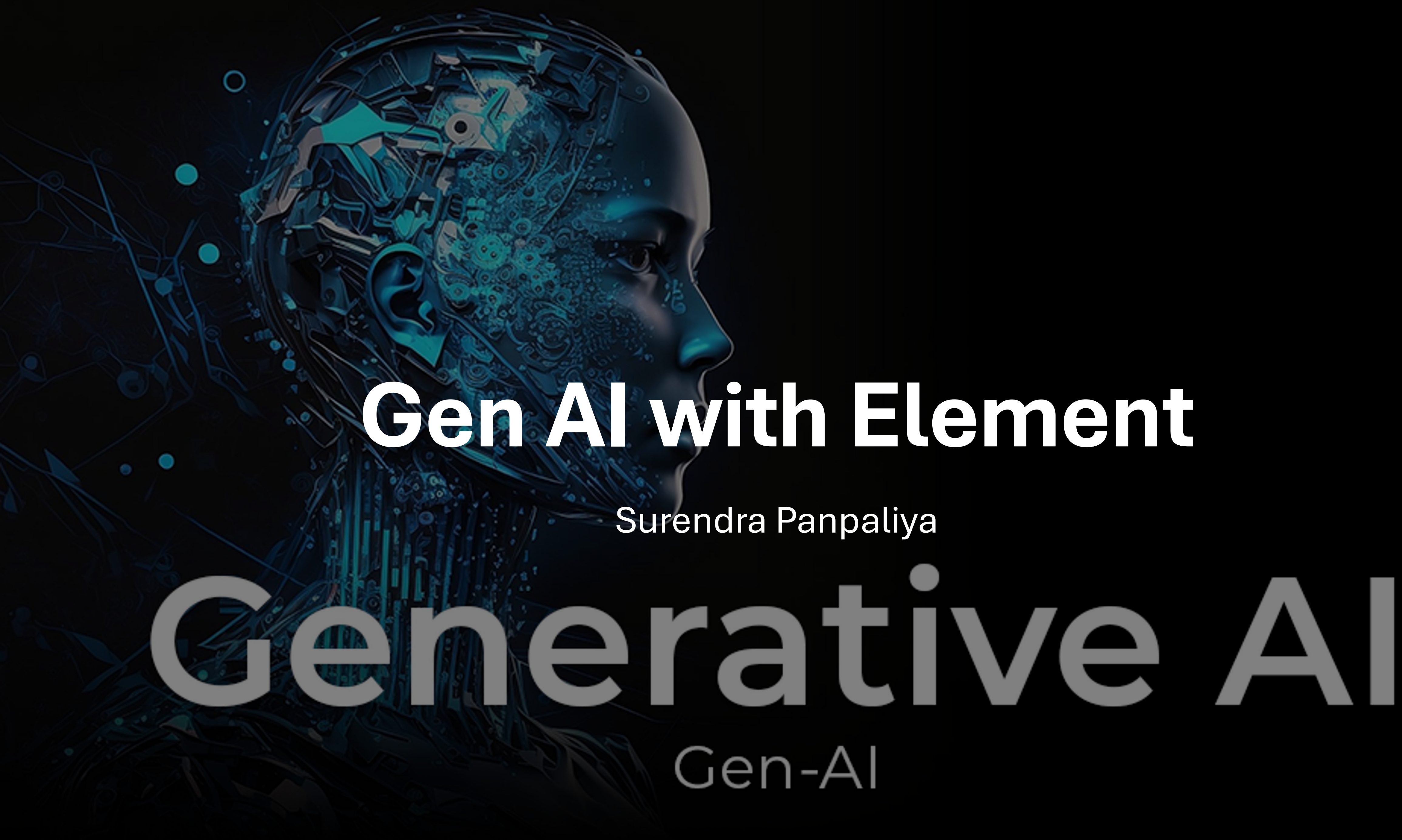
Gen AI with Element

Surendra Panpaliya



PREREQUISITES

Participants should have:

Basic knowledge of **Python programming**

Familiarity with **APIs**, **JSON**, and **HTTP requests**

General understanding of **Machine Learning** and **NLP concepts**

PREREQUISITES

Awareness of **cloud platforms** (Azure or GCP preferred)

Prior exposure to **Jupyter Notebooks** or **VS Code**

Knowledge of **RESTful APIs**, **Docker**, and **Git**

Some experience with **LLMs** or **prompt engineering**

LAB SETUP REQUIREMENTS

Python 3.10+ installed (preferably in a virtual environment)

JupyterLab or **VS Code** with Python plugin

Access to:

Azure OpenAI API key (or)

Google GenAI credentials (Vertex AI Studio, PaLM/Gemini API)

📌 **LEARNING OUTCOMES**

Explain key concepts in **Generative AI** and **Transformer-based LLMs**

Build and interact with **OpenAI/Gemini models** using Python

📌 **LEARNING OUTCOMES**

Design **RAG pipelines** integrated with

LangChain + Milvus

Apply structured prompt

engineering strategies in LLM apps

📌 LEARNING OUTCOMES

Utilize Walmart's internal

LLM Gateway and **evaluation platforms**

Create simple **Agentic applications**

with planning, execution, and tools

📌 LEARNING OUTCOMES

- Troubleshoot common LLM issues
- such as hallucination or bias
- Build and present a functional
- **Excel-based report builder** GenAI app

Agenda

- DAY 1: GENAI FOUNDATION & ARCHITECTURE
- DAY 2: WALMART GENAI ECOSYSTEM
- DAY 3: APPLICATION DEVELOPMENT WITH GENAI
- DAY 4: HACKATHON & DEPLOYMENT

DAY 3: APPLICATION DEVELOPMENT WITH GENAI

Objective:

Build enterprise-level GenAI apps

using agent-based designs,

chaining, and governance features.

Agenda

- Agentic AI & Task Chaining
- Governance and Evaluation
- Hands-On Session

Objective

BUILD ENTERPRISE-LEVEL GENAI APPS

USING AGENT-BASED DESIGNS,

CHAINING, AND

GOVERNANCE FEATURES.

Agentic AI & Task Chaining

What is Agentic AI?	Core components:	Agents
Tools	Planner	Executor

What is Agentic AI?

- Agentic AI = Smart AI agents that
- Understand goals
- Plan steps
- Execute tasks
- Work together (like a team)

What is Agentic AI?

New way of using AI

to **plan, execute, and manage tasks**

independently

like an intelligent assistant

that thinks and acts.

What is Agentic AI?

ACT LIKE A RESPONSIBLE ASSISTANT,

DOING TASKS **INDEPENDENTLY**

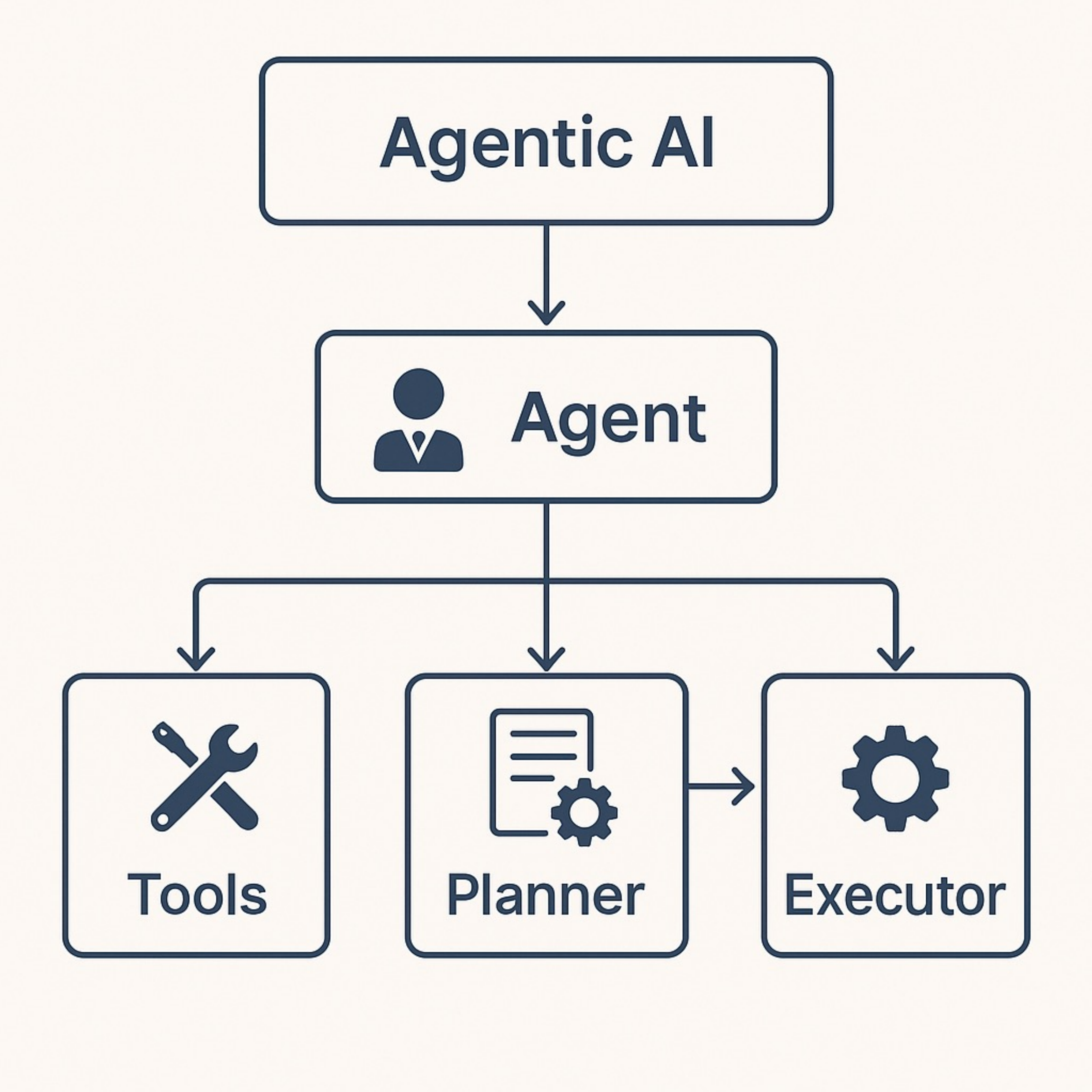
BASED ON GOALS YOU GIVE IT.

Agentic AI uses Agents that can

- Understand goals
- Break them into tasks
- Use tools or APIs to complete tasks
- Adjust actions based on results

Core Components of Agentic AI

1. Agents
2. Tools
3. Planner
4. Executor


Agentic AI
Agent
Tools
Planner
Executor

1. Agents

Individual units with specific roles or goals.

Can reason, learn, and make decisions

based on their goals.

2. Tools

Resources or capabilities available to agents.

For example: APIs, databases,

web search engines, or AI models.

3. Planner

Determines how to achieve the goal.

Decides which agents

should collaborate, and in what order.

4. Executor

Executes the planned tasks.

Coordinates interaction

between agents and tools

to complete tasks.

Core Components of Agentic AI

Component	Purpose
🧑‍💼 Agent	An intelligent entity that receives instructions and decides what to do.
🧰 Tool	A function or API the agent can use (like a search engine, database query, code interpreter, etc.)
📋 Planner	Breaks down a big problem into smaller tasks.
⚙️ Executor	Executes tasks one by one, using tools, and adjusts based on feedback.

Walmart-Specific Analogy

Imagine an **AI Assistant**

for Inventory Management at Walmart

Goal:

Restock top-selling items in Pune region.

Walmart-Specific Analogy

Role	Example Behavior
Agent	Decides what to do, e.g., "I need to find the top-selling items, then check current inventory, then create purchase orders."
Tool	Calls internal APIs like get_sales_data(), check_inventory(), create_PO()

Walmart-Specific Analogy

Role	Example Behavior
Planner	Breaks it into: 1. Identify top items 2. Check stock 3. Restock
Executor	Runs the steps, checks outputs, and moves to the next step

Real-world Example for Walmart

Imagine Walmart wants an AI solution

to automatically handle

customer inquiries about products

Real-world Example for Walmart

Agent	Role/Goal	Tools
Chat Agent	Interacts with customers	ChatGPT, Dialogflow
Product Agent	Finds product information	Walmart product catalog API
Stock Agent	Checks stock availability	Walmart inventory database
Pricing Agent	Provides current pricing details	Pricing API

Agent Design Patterns

ReAct

Chain-of-Thought

Plan-and-Execute

What Are Agent Design Patterns?

AGENT DESIGN PATTERNS DEFINE

HOW AN AI AGENT

THINKS, PLANS, AND

EXECUTES TASKS.

What Are Agent Design Patterns?

These patterns guide:

- How the agent reasons
- How tasks are broken down
- How tools are used
- How the agent adapts to feedback

Key Agent Design Patterns

- ReAct (Reasoning and Acting)
- Chain-of-Thought (CoT)
- Plan-and-Execute (PnE)

ReAct (Reasoning and Acting)

The agent **reasons**

step-by-step and then

decides an action,

repeating until

the final answer is found.

Example

“Which supplier provides the best rate for top-selling laptops?”

Thinks:

I need to fetch top-selling laptops

→ compare supplier prices → choose best.

Acts: Calls APIs or tools one step at a time.

Pattern

Thought → Action → Observation →

Thought → Action → … → Final Answer

Chain-of-Thought (CoT)

The agent is encouraged

to **explain its reasoning process**

before giving an answer.

It doesn’t act with tools

but reasons clearly.

What is Chain-of-Thought (CoT)?

Instead of jumping to the final answer,

the model is instructed to **think aloud**

by breaking the problem into

logical reasoning steps.

What is Chain-of-Thought (CoT)?

Improves accuracy, interpretability

often factual correctness,

especially for complex or

multi-step questions.

Example

If Walmart has 20% more customers this month,

what could be the reason?

Example

Agent explains:

"Footfall has increased."

"Marketing campaign launched last month."

"Competitor store closed nearby."

Prompt Example

prompt = "Walmart's sales increased by 20%.

Think step-by-step about possible causes."

response = llm(prompt)

Plan-and-Execute (PnE)

The agent **first plans**

the full sequence of tasks,

then executes each step.

Example (Walmart Use Case)

“Launch a weekend sale campaign

across 3 top-selling categories.”

Example (Walmart Use Case)

Plan	Plan: Identify top categories
Pricing	Fetch pricing
Create	Create campaign draft
Submit	Submit for approval
Execute	Execute: Each step is called via API or tool.

What is Plan-and-Execute (PnE)?

Powerful agent pattern that:

First plans a sequence of subtasks

based on the user request.

Then **executes each step**,

often using tools, APIs, or functions.

What is Plan-and-Execute (PnE)?

This improves

reliability,

transparency, and

modularity in AI task solving.

Use Case

Prompt:

“Launch a weekend sale campaign

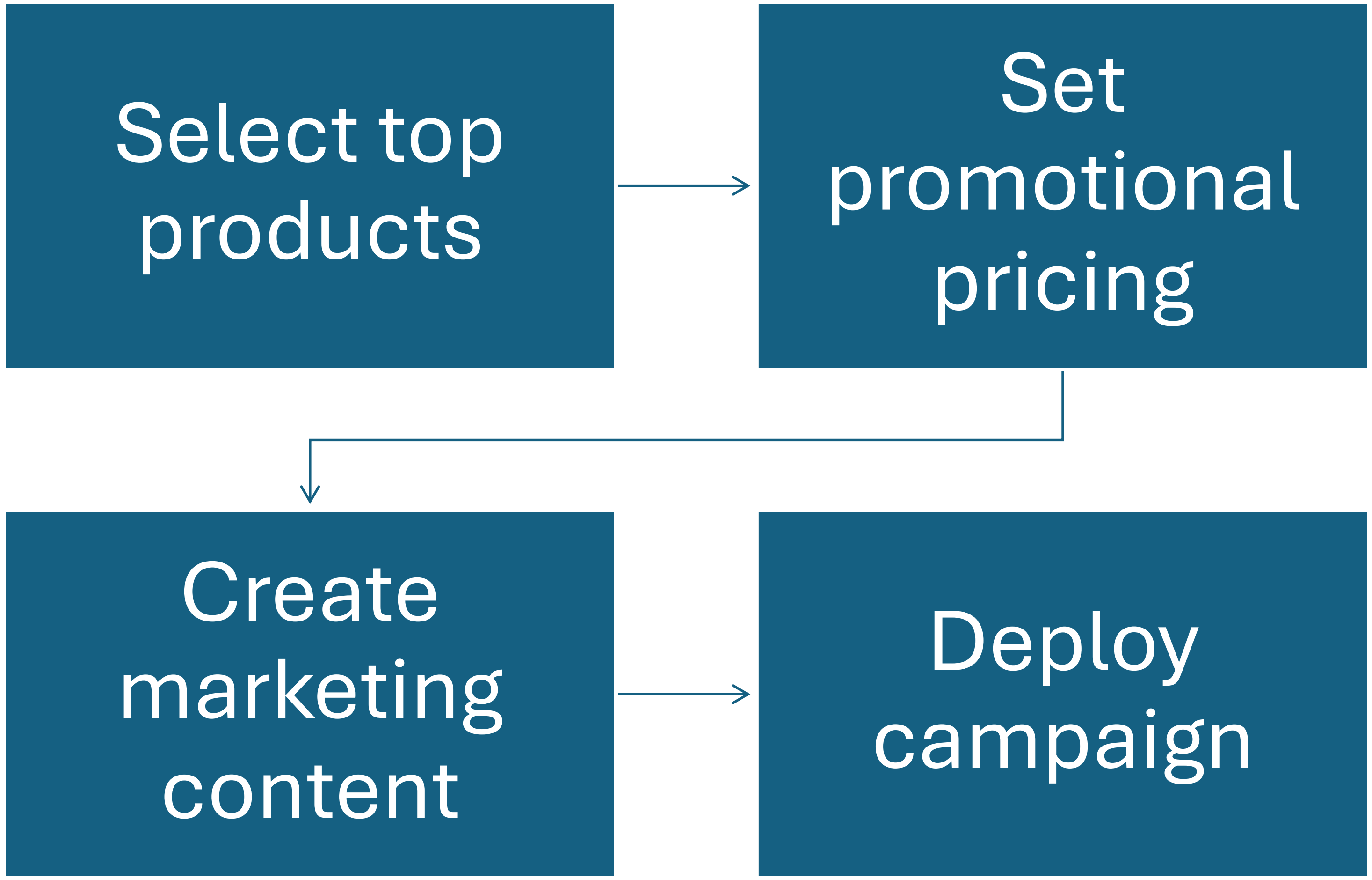
across 3 top-selling categories.”

Use Case

Launching promotional campaigns.

Restocking workflows.

Example Plan



PnE vs CoT vs ReAct

Pattern	Reasoning Style	Complexity	Ideal Use Cases (Walmart)
ReAct	Iterative reasoning-action loop	Medium	Inventory, pricing decisions
Chain-of-Thought	Step-by-step reasoning	Low	Sales trend analysis, customer insights
Plan-and-Execute	Detailed planning & structured execution	Medium-High	Promotional campaigns, restocking workflows

PnE vs CoT vs ReAct

Pattern	Suitable for	Reasoning type	Complexity
ReAct	Dynamic decision-making with tools	Reason and act iteratively	Medium
Chain-of-Thought	Complex analytical tasks	Step-by-step logical reasoning	Low
Plan-and-Execute	Structured multi-step workflows	Planning ahead, then executing	Medium-High

PnE vs CoT vs ReAct

Style	Planning	Tool Use	Ideal For
CoT	Yes	❌	Pure reasoning/explanation
ReAct	Step-by-step	✅	Interactive tool calls while thinking aloud
Plan-and-Execute	✅ Full upfront plan	✅ Sequential tool calls	Multi-step automation, real tasks

Governance and Evaluation

- Walmart GenAI Evaluator: Why evaluation is critical
- Standardization of LLM outputs
- Monitoring hallucination, bias, and failures
- Overview of Walmart’s Agentic AI platform offerings

What is GenAI Governance?

Set of practices ensuring that

AI solutions comply with organizational policies,

legal frameworks, ethical standards,

and business objectives.

What is GenAI Evaluation?

Systematic assessment

to ensure AI models are accurate,

fair, reliable, robust, and

aligned with business requirements.

What is GenAI Evaluation?

Evaluation is a structured approach

to measure and improve the effectiveness,

accuracy, fairness, and safety

of Generative AI (GenAI) models.

Why Is GenAI Evaluation Essential at Walmart?

- Ensuring Accuracy & Reliability
- Mitigating Risks & Biases
- Enhancing Customer Trust
- Regulatory Compliance
- Continuous Improvement

1️⃣ Ensuring Accuracy & Reliability

Confirms the AI-generated responses and

actions align with Walmart's business requirements.

Maintains trust by providing consistently accurate results.

2️⃣ Mitigating Risks & Biases

Detects and reduces unwanted biases

that could harm Walmart’s reputation.

Prevents incorrect decisions

that could lead to financial or operational risks

3️⃣ Enhancing Customer Trust

- Customers interact confidently
- with reliable, transparent AI solutions.
- Strengthens Walmart’s brand value
- by ensuring fairness and trustworthiness
- in automated interactions.

4️⃣ Regulatory Compliance

Ensures that Walmart's AI solutions

comply with global regulations and

standards, avoiding legal issues.

5️⃣ Continuous Improvement

Provides insights into

model performance,

highlighting areas

for further development.

5️⃣ Continuous Improvement

Enables Walmart

to maintain competitive

advantage through adaptive and

improved GenAI solutions.

Key Metrics in GenAI Evaluation

Accuracy & Precision:

Correctness of AI responses.

Fairness & Bias:

AI decisions equitable across diverse user groups.

Key Metrics in GenAI Evaluation

Robustness:

Stability of AI under various conditions.

Safety & Ethics:

AI adherence to ethical guidelines and policies.

Potential Risks Without Effective Evaluation

- Misleading customer interactions.
- Financial losses from incorrect AI decisions.
- Reputational damage from
- biased or inappropriate outputs.
- Legal and compliance risks.

Practical Steps for Walmart GenAI Evaluators

- Set clear, measurable criteria aligned with business objectives.
- Implement standardized evaluation frameworks and tools.
- Conduct regular audits and reviews.
- Use feedback loops for continuous refinement

Conclusion

At Walmart,

rigorous GenAI evaluation

is not optional

it’s foundational.

Conclusion

It directly impacts

customer trust,

brand reliability,

business profitability, and compliance.

Conclusion

- Prioritizing GenAI evaluation ensures
- Walmart remains a leader in
- ethical, efficient, and
- effective AI deployment.

Hands-On Session

- Build a **Modular Control Point (MCP) Server**
- Create a LangChain-based **SQL Generator Agent**
- Prompt engineering for SQL
- Schema-aware querying
- Tool execution flow

Building an Agentic SQL Generator with MCP Server using LangChain

Surendra Panpaliya
GKTCS Innovations
https://www.gktcs.com

What is an MCP Server?

A **Modular Control Point (MCP)** Server:

Acts as the **central brain** to manage agent flow

Coordinates tools, models, inputs, and outputs

Modular, pluggable, and reusable across use cases

What is a SQL Generator Agent?

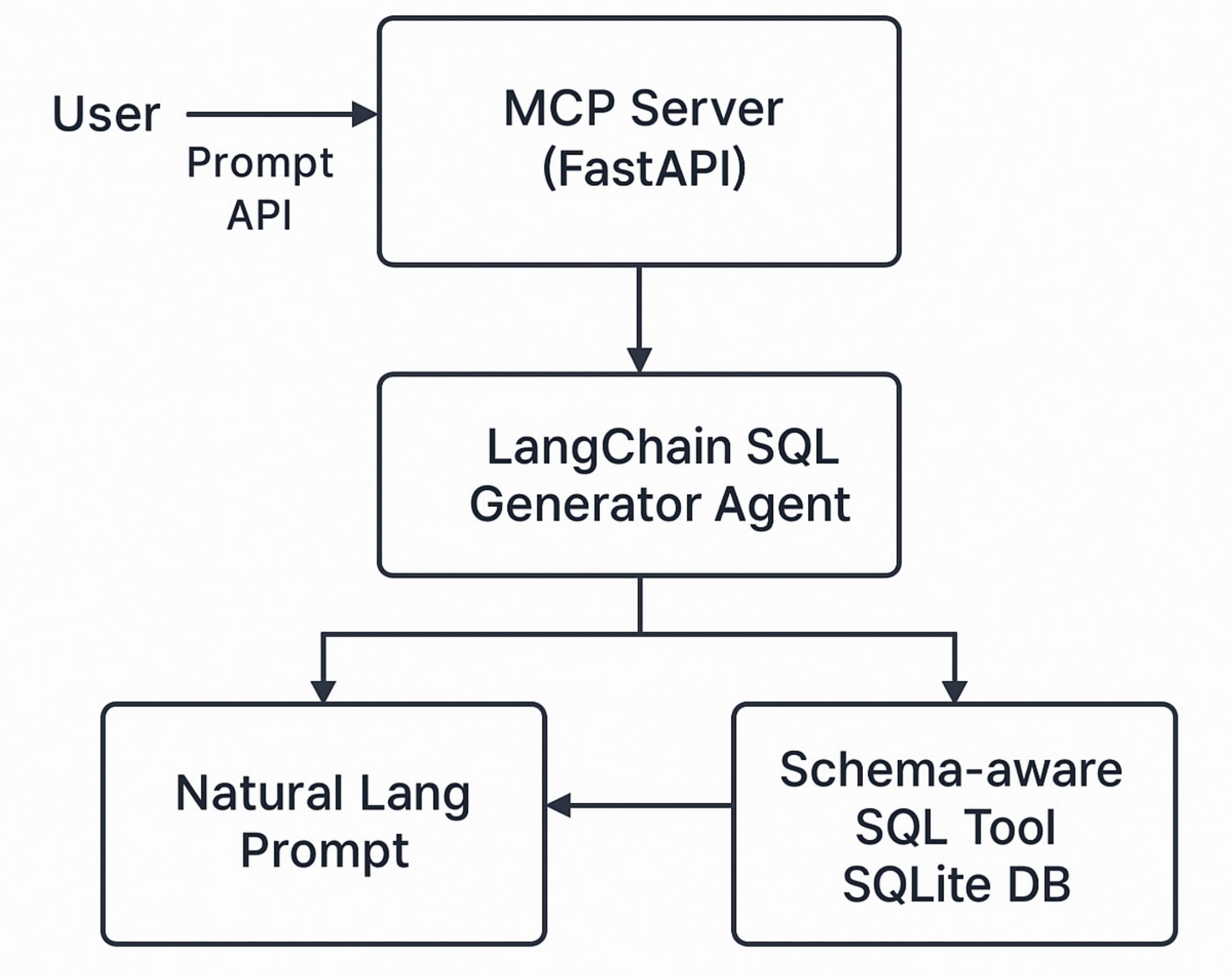
An **LLM-powered agent** that:

Accepts natural language questions

Understands the **SQL schema**

Uses LangChain tools to **generate executable SQL**

Architecture Diagram



Prerequisites

- Python 3.10+
- pip install langchain openai fastapi uvicorn sqlite3
- OpenAI API Key

STEP 1: Set Up a Basic MCP Server

#Create mcp_server.py

```
from fastapi import FastAPI, Request
from langchain.chains import ConversationChain
from langchain.memory import ConversationBufferMemory
from langchain.llms import OpenAI
```

STEP 1: Set Up a Basic MCP Server

```
app = FastAPI()

# MCP-level LLM engine
llm = OpenAI(temperature=0)
```

STEP 1: Set Up a Basic MCP Server

```
 # Store context
memory = ConversationBufferMemory()

# Core conversation chain
chain = ConversationChain(llm=llm, memory=memory)
```

STEP 1: Set Up a Basic MCP Server

```
@app.post("/mcp")
async def handle_query(req: Request):
    data = await req.json()
    query = data.get("query")
    response = chain.run(query)
    return {"response": response}
```

STEP 1: Set Up a Basic MCP Server

Run Server:

uvicorn mcp_server:app --reload

What is an MCP Server?

An **MCP server** is like the **brain** of an AI system:

It **receives inputs** from users (e.g., natural language queries)

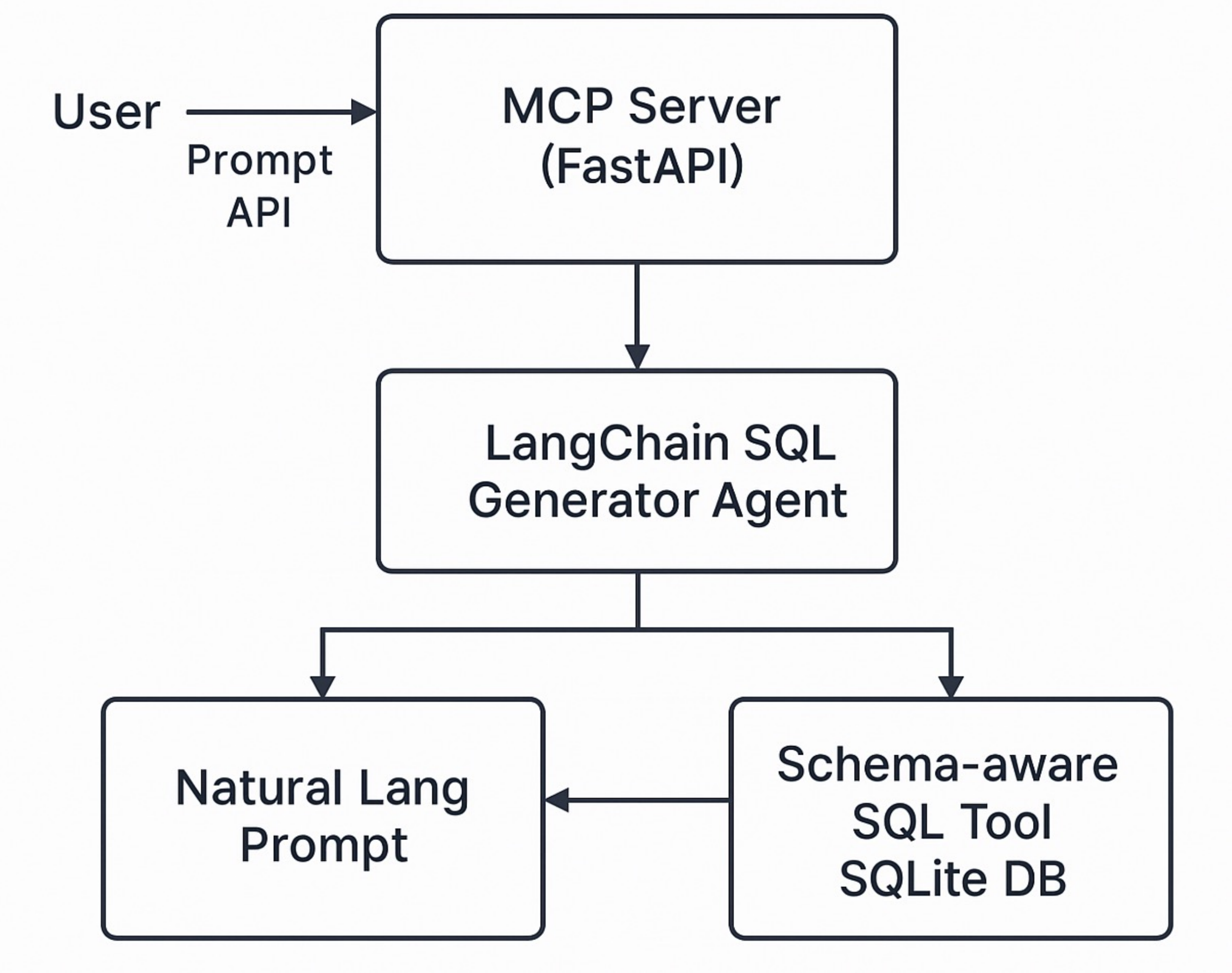
Coordinates tools (e.g., LLMs like GPT-4o)

What is an MCP Server?

Maintains **memory** (context between sessions)

Returns **relevant outputs** (like SQL for internal business logic)

MCP Server Architecture



Code Explained

- # 🚀 Import FastAPI and LangChain dependencies
- from fastapi import FastAPI
- from pydantic import BaseModel
- from fastapi.responses import JSONResponse

- FastAPI: Web framework to expose /mcp endpoint
- BaseModel: Used to define JSON request schema
- JSONResponse: Return proper error messages

Code Explained

- from langchain_openai import ChatOpenAI
- from langchain_core.messages import HumanMessage
- from langchain_core.runnables import RunnableLambda
- from langchain_core.runnables.history import RunnableWithMessageHistory
- from langchain_community.chat_message_histories import ChatMessageHistory

Code Explained

- ChatOpenAI: GPT-4o from OpenAI
- HumanMessage: Standard message format for conversation
- RunnableLambda: A callable function wrapped into LangChain interface
- RunnableWithMessageHistory: Adds memory (past messages) to conversation chain
- ChatMessageHistory: Stores conversation history per user session

Code Explained

- import os, traceback
- from pprint import pprint
- os: To set environment variables like API keys
- traceback: Helps with detailed error printing
- pprint: For better debugging logs

Code Explained

- # Set API key
- os.environ["OPENAI_API_KEY"] = "..." # Replace this with secure env setup
- Authenticates your requests to OpenAI’s GPT-4o API

Code Explained

- **Security Note**: Never hardcode API keys in production.
- Use dotenv, secret manager, or environment variables.

- app = FastAPI()
- Initializes the FastAPI server

Code Explained

- # LLM Model
- llm = ChatOpenAI(temperature=0, model="gpt-4o")
- You define the language model with deterministic behavior (temperature=0)
- gpt-4o is the model version being used

Code Explained

```
# Request body
class QueryRequest(BaseModel):
    session_id: str
    query: str
```

Defines the expected request format: a session ID and a query string

Code Explained

```
# Session memory
session_store = {}

def get_memory(session_id: str):
    if session_id not in session_store:
        session_store[session_id] = ChatMessageHistory()
    return session_store[session_id]
```

Code Explained

session_store:

Dictionary to keep memory objects

for each user session

If session is new →

it initializes ChatMessageHistory

Code Explained

```
# Natural language → SQL template
def query_to_sql(messages):
    messages = messages["input"]
    user_input = messages[-1].content
```

messages["input"] accesses the list of HumanMessages

messages[-1].content: Grabs the last user message content

Code Explained

```
llm_messages = [
    HumanMessage(content=f"Convert the following natural language query to SQL: {user_input}")
 ]
```

Code Explained

Constructs a **prompt** with instruction for GPT-4o

```
pprint(llm_messages)
return llm.invoke(llm_messages)
```

Sends prompt to GPT-4o and gets response (i.e., SQL)

Code Explained

```
# Wrap with LangChain memory handler
chain = RunnableWithMessageHistory(
    RunnableLambda(query_to_sql),
    lambda session_id: get_memory(session_id),
    input_messages_key="input",
    history_messages_key="history",
)
```

Code Explained

- This wraps query_to_sql so
- that **LangChain adds session memory**
- It maps "input" and "history" keys
- for proper state management

Code Explained

```
# 🚀 Endpoint to handle Walmart agent queries
@app.post("/mcp")
async def handle_query(request: QueryRequest):
HTTP POST route /mcp that accepts session_id and query
    try:
        session_id = request.session_id
        query = request.query
```

Code Explained

Extract data from JSON body

```
print(f"[Walmart-MCP] Session: {session_id} | Query: {query}")
```

For debugging/logging

```
response = await chain.ainvoke(
    {"input": [HumanMessage(content=query)]},
    config={"configurable": {"session_id": session_id}},
)
```

ainvoke() is used for asynchronous chaining

Code Explained

This passes the user query into memory-managed RunnableWithMessageHistory

```
        return {"response": response.content}
```

Extract final response and return to the user

```
    except Exception as e:
        traceback.print_exc()
        return JSONResponse(status_code=500, content={"error": str(e)})
```

Catch and return detailed error info in JSON

Summary: How MCP Server Works in This Example

Component	Description
🧾 **Input**	Natural language query from the user (e.g., “Show me today’s top 5 items”)
🧠 **LLM Agent**	GPT-4o generates SQL output using LangChain interface
💾 **Memory**	ChatMessageHistory maintains ongoing session per user
🔗 **Modular Chain**	RunnableWithMessageHistory allows plug-and-play for other agents/tools
🌐 **Endpoint**	FastAPI /mcp POST handles input/output

Happy Learning !!
Thanks for Your Patience ☺

Surendra Panpaliya

GKTCS Innovations