Agentic AI represents a paradigm shift where AI systems transition from passive responders to autonomous "digital coworkers" capable of pursuing high-level goals by planning and executing their own steps.

Core Operational Cycle: The Agent Loop

Unlike traditional linear AI (Input → Process → Output), agentic systems operate in a continuous, self-correcting cycle often called the **Perceive-Reason-Act-Learn (PRAL)** loop.

- **Perceive**: The agent gathers real-time data from its environment via APIs, sensors, databases, or user input.
- **Reason & Plan**: An LLM acts as the "brain," decomposing a complex goal (e.g., "organize a business trip") into smaller, logical sub-tasks.
- **Act**: The system uses "tools"—external functions or software integrations—to execute tasks like sending emails, querying a CRM, or running code.
- **Learn & Reflect**: The agent observes the results of its actions. If a step fails, it "reflects" on the error and adapts its plan on the fly to try a different approach.

Key Technical Components

For an agent to be truly "agentic," it must integrate several architectural pillars:

- **Planning Engine**: Uses frameworks like **ReAct** (Reason + Act) to interleave thinking with tool usage, ensuring the agent doesn't just guess but researches and verifies as it goes.
- **Memory Systems**:
  - **Short-term**: Maintains the state of the current task.
  - **Long-term**: Stores historical outcomes and user preferences to improve performance over time.
- **Tool-Use Layer**: Standardized interfaces (like the Model Context Protocol) that allow the LLM to securely call external APIs or databases.
- **Multi-Agent Orchestration**: Complex workflows may involve specialized agents (e.g., a "researcher" agent passing data to a "writer" agent) coordinated by a "manager" agent.

Real-World Examples

- **IT Support**: A self-healing helpdesk agent monitors device health, detects a VPN failure, autonomously diagnoses the cause, and applies a patch without human intervention.
- **Supply Chain**: An agent detects a shipping delay due to weather, automatically searches for alternative vendors, compares prices, and places a new order to prevent a stockout.
- **Software Engineering**: Agents can write code, run tests, analyze error logs, and iterate on the code until the tests pass.

Safety & Governance

Because these systems act autonomously, they require strict **guardrails**:

- **Human-in-the-loop (HITL)**: High-impact actions (e.g., executing a financial trade) can be "gated" to require human approval before completion.
- **Action Budgets**: Setting limits on how many steps or how much money an agent can spend on a single task.

To understand Agentic AI, think of it as a system with a **Brain** (LLM), **Hands** (Tools), and a **Notebook** (State/Memory). Below is a conceptual example using the popular LangGraph framework to build a "Research Assistant".

The Scenario

Goal: Create an agent that can search the web and write a summary.

- **Input:** "What is the current price of Gold?"

- **The Problem:** An LLM alone doesn't know today's price. It needs to *decide* to use a search tool.

1. Basic Technical Components

Building this agent involves three main parts in your code:

- **State (The Notebook):** A shared data structure that tracks the conversation and tool outputs.
- **Nodes (The Workstations):** Python functions that do the work. One node calls the LLM; another runs the search tool.
- **Edges (The Paths):** Logic that tells the agent where to go next (e.g., "If the LLM says it needs more info, go to the Search Node").

2. Conceptual Code Example (Python)

This example follows the **ReAct** (Reason + Act) pattern.

```
from langgraph.graph import StateGraph, END

from langchain_openai import ChatOpenAI
```

```
# 1. Define the 'Hands' (Tools)
def search_tool(query):
    # This represents an actual API call to Google or Bing
    return f"The current price of Gold is $2,700 per ounce."

# 2. Define the 'Brain' (LLM Node)
def call_model(state):
    messages = state['messages']
    # The LLM looks at the question and decides: "I need to search."
    response = llm.invoke(messages)
    return {"messages": [response]}

# 3. Define the 'Action' (Tool Node)
def call_tool(state):
    # Executes the search_tool based on the LLM's request
    last_message = state['messages'][-1]
    result = search_tool(last_message.tool_calls[0]['args']['query'])
    return {"messages": [result]}

# 4. Build the 'Blueprint' (The Graph)
workflow = StateGraph(AgentState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", call_tool)
```

```
# Set the flow: Start -> Agent -> (Decision: Tool or End?)
workflow.set_entry_point("agent")
workflow.add_conditional_edges("agent", should_continue) # Logic to stop or use tools
workflow.add_edge("tools", "agent") # After using a tool, go back to the brain

app = workflow.compile()
```

3. Detailed Execution Steps:-

When you run the agent, it follows this autonomous multi-step plan:

1. **Thought:** The agent receives your question. It realizes its training data is old and it lacks the current gold price.
2. **Action:** It triggers the `search_tool` node.
3. **Observation:** The tool returns "$2,700." The agent records this in its "Notebook" (State).
4. **Final Response:** The agent goes back to the "Brain" node, sees the new information, and tells you: "The current price of Gold is $2,700

4. Key Safety: Guardrails

In a real-world system, you must add **Guardrails** before allowing full autonomy:

- **Max Loops:** Limit the agent to 5 steps so it doesn't get stuck in an infinite loop.

Human-in-the-Loop: For sensitive tasks like "Send Email," the agent must pause and ask for your permission before the final step.

Using ollama with llama3 ai model example for creating ai agent :-

```
from typing import Annotated, TypedDict

from langgraph.graph import StateGraph, END

from langgraph.prebuilt import ToolNode

from langchain_ollama import ChatOllama

from langchain_core.messages import BaseMessage, HumanMessage

from langchain_core.tools import tool


# 1. Define the 'Hands' (Tools) using the @tool decorator

@tool

def search_tool(query: str):

    """Consult this tool to get the current price of commodities like gold."""

    # In a real app, use a library like 'tavily-python' or 'requests'

    return "The current price of Gold is $2,700 per ounce."


tools = [search_tool]


# 2. Setup the 'Brain' (Ollama Llama 3)

# Ensure you have run: ollama pull llama3.1

llm = ChatOllama(model="llama3.1", temperature=0).bind_tools(tools)
```

```
# 3. Define the State
class AgentState(TypedDict):
    messages: Annotated[list[BaseMessage], "The conversation history"]

# 4. Define Logic Nodes
def call_model(state: AgentState):
    response = llm.invoke(state["messages"])
    return {"messages": [response]}

# Define the Tool Node using LangGraph's prebuilt helper
tool_node = ToolNode(tools)

# 5. Conditional Logic: Should we stop or call a tool?
def should_continue(state: AgentState):
    last_message = state["messages"][-1]
    if last_message.tool_calls:
        return "tools"
    return END

# 6. Build the Graph
workflow = StateGraph(AgentState)

workflow.add_node("agent", call_model)
workflow.add_node("tools", tool_node)
```

```
workflow.set_entry_point("agent")

workflow.add_conditional_edges("agent", should_continue)

workflow.add_edge("tools", "agent")


app = workflow.compile()


# --- Execution ---

inputs = {"messages": [HumanMessage(content="What is the current price of gold?")]}

for output in app.stream(inputs):

    print(output)
```

Key Changes Explained

1. **`ChatOllama`**: Replaced OpenAI with the local Ollama provider. Note that you must have the **Ollama app** running on your machine.
2. **`bind_tools`**: This is critical. It tells Llama 3 what tools are available and how to format the "request" to use them.
3. **`ToolNode`**: Instead of writing a manual `call_tool` function, LangGraph provides a prebuilt `ToolNode` that automatically handles the execution and formatting of tool outputs.
4. **Llama 3.1+**: Ensure you are using at least **Llama 3.1**, as the original Llama 3 (8B) does not support structured tool calling natively as well as the updated versions.

To run the LangGraph agent locally with Ollama and Llama 3.1, follow this step-by-step guide to set up your environment and execute the code.

Step 1: Install and Configure Ollama

1. **Download Ollama**: Visit the official Ollama website and download the version for your OS (Windows, macOS, or Linux).
2. **Pull Llama 3.1**: Open your terminal (CMD or PowerShell) and run the following command to download the 8B model, which supports native tool calling:

bash

```
ollama pull llama3.1
```

Use code with caution.

3. **Verify**: Ensure the Ollama server is running (usually indicated by an icon in your system tray or by visiting `http://localhost:11434` in your browser).

Step 2: Set Up Your Python Environment

It is best practice to use a virtual environment to manage dependencies.

1. **Create and Activate Environment**:

bash

```
python -m venv ai_env
# Windows:
ai_env\Scripts\activate
# macOS/Linux:
source ai_env/bin/activate
```

2. **Install Required Libraries**: Install the latest versions of LangChain's Ollama integration and LangGraph:

```
pip install -U langchain-ollama langgraph
```

Use code with caution.

Step 3: Execute the Agent Code

Create a file named `agent.py` and paste the following code. Note that **Llama 3.1** is required for `bind_tools` to work correctly.

python

```python
from typing import Annotated, TypedDict
from langgraph.graph import StateGraph, END
from langgraph.prebuilt import ToolNode
from langchain_ollama import ChatOllama
from langchain_core.messages import BaseMessage, HumanMessage
from langchain_core.tools import tool

# 1. Define your tool with a clear docstring (essential for LLM to
understand)
@tool
def search_tool(query: str):
    """Search for real-time commodity prices like gold or silver."""
    return "The current price of Gold is $2,700 per ounce."

tools = [search_tool]

# 2. Initialize Llama 3.1 via Ollama with tools bound to it
llm = ChatOllama(model="llama3.1", temperature=0).bind_tools(tools)

# 3. Define the State and Nodes
class AgentState(TypedDict):
    messages: Annotated[list[BaseMessage], "The conversation history"]

def call_model(state: AgentState):
    response = llm.invoke(state["messages"])
    return {"messages": [response]}

def should_continue(state: AgentState):
    last_message = state["messages"][-1]
    return "tools" if last_message.tool_calls else END

# 4. Build and Compile the Graph
workflow = StateGraph(AgentState)
workflow.add_node("agent", call_model)
workflow.add_node("tools", ToolNode(tools))

workflow.set_entry_point("agent")
workflow.add_conditional_edges("agent", should_continue)
workflow.add_edge("tools", "agent")
```

```
app = workflow.compile()

# 5. Run the agent
inputs = {"messages": [HumanMessage(content="What is the price of gold
today?")]}
for chunk in app.stream(inputs):
    print(chunk)
```

Step 4: Troubleshooting Tips

- **Connection Error**: If you see a connection error, make sure the Ollama app is open and running in the background.
- **Tool Calling Fails**: Older models like the original `llama3` (not 3.1) often struggle with structured tool output. Always use `llama3.1` or higher for agentic workflows.
- **Performance**: If the model is slow, ensure you have sufficient RAM (minimum 8GB for the 8B model) or a dedicated GPU

To add persistent memory using **FAISS**, you move from a simple message history to a **Long-Term Memory** architecture. The agent will now "save" important facts into a local vector database and "recall" them in future sessions.

1. Install Additional Requirements

You need the FAISS library and a local embedding model (to turn text into numbers).

bash

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

Use code with caution.

2. The Memory-Enhanced Agent Code

This version adds a **Memory Node** that stores the user's name or preferences permanently to your disk.

python

```python
import faiss
import os
from typing import Annotated, TypedDict
from langgraph.graph import StateGraph, END
from langchain_ollama import ChatOllama, OllamaEmbeddings
from langchain_community.vectorstores import FAISS
from langchain_community.docstore.in_memory import InMemoryDocstore
from langchain_core.messages import BaseMessage, HumanMessage, SystemMessage

# --- 1. SETUP PERSISTENT FAISS ---
# Use Ollama for local embeddings to keep everything 100% private
embeddings = OllamaEmbeddings(model="llama3.1")
index_file = "faiss_index"

if os.path.exists(index_file):
    vector_store = FAISS.load_local(index_file, embeddings,
allow_dangerous_deserialization=True)
else:
    # Initialize an empty FAISS index
    index = faiss.IndexFlatL2(4096) # 4096 is the dimension for Llama 3
    vector_store = FAISS(embeddings, index, InMemoryDocstore({}), {})

# --- 2. DEFINE THE BRAIN ---
llm = ChatOllama(model="llama3.1", temperature=0)
```

```
class AgentState(TypedDict):
    messages: Annotated[list[BaseMessage], "The conversation history"]

def call_model(state: AgentState):
    # RECALL: Search FAISS for relevant past facts
    user_query = state["messages"][-1].content
    past_memories = vector_store.similarity_search(user_query, k=2)

    # GROUNDING: Inject memories into the System Prompt
    memory_context = "\n".join([m.page_content for m in past_memories])
    system_msg = SystemMessage(content=f"You are a helpful assistant. Past
info you remember: {memory_context}")

    response = llm.invoke([system_msg] + state["messages"])

    # SAVE: If the user shared a fact, save it
    if "my name is" in user_query.lower() or "i like" in user_query.lower():
        vector_store.add_texts([user_query])
        vector_store.save_local(index_file)

    return {"messages": [response]}

# --- 3. BUILD GRAPH ---
workflow = StateGraph(AgentState)
workflow.add_node("agent", call_model)
workflow.set_entry_point("agent")
workflow.add_edge("agent", END)
app = workflow.compile()

# --- 4. TEST PERSISTENCE ---
# Session 1: Tell the agent something
app.invoke({"messages": [HumanMessage(content="Hi, my name is Alex and I like
Blue.")]})

# Session 2: Ask it later (even after a restart!)
response = app.invoke({"messages": [HumanMessage(content="What is my name and
favorite color?")]})
print(response["messages"][-1].content)
```

How it Works

- **Semantic Search**: Before the LLM answers, the agent uses FAISS to find the most "mathematically similar" pieces of your past conversations.
- **Prompt Injection**: It inserts those found facts (e.g., "The user's name is Alex") into the **System Message** so the LLM has that context.
- **Local Storage**: By using `vector_store.save_local()`, your memory is written to a folder on your computer. It survives even if you turn off your PC.

When to Use This

- **FAISS (Long-Term)**: Use this for facts that should last **forever** (e.g., user profile, business rules).
- **Checkpointers (Short-Term)**: Use LangGraph's `SqliteSaver` if you just want the agent to remember the **last 5 messages** of the current chat