Contents

[AGENTIC AI 2](#_Toc207395359)

[TRADITIONAL VERSUS AGENTIC AI 2](#_Toc207395360)

[EXAMPLES 2](#_Toc207395361)

[AGENTIC AI VERSUS AI AGENTS 3](#_Toc207395362)

[AI AGENTS 4](#_Toc207395363)

[AGENTIC AI 4](#_Toc207395364)

[DESIGN PATTERNS FOR AGENTIC AI 5](#_Toc207395365)

[REFLECTION PATTERN 5](#_Toc207395366)

[TOOL USE PATTERN 5](#_Toc207395367)

[PLANNING PATTERN 5](#_Toc207395368)

[MULTI-AGENT PATTERN 5](#_Toc207395369)

[CORE PILLARS OF AGENTIC AI 6](#_Toc207395370)

[EXAMPLE 7](#_Toc207395371)

[IMPLEMENTING TOOLS & TOOL CALLING IN LLMS FOR AGENTIC AI WITH LANGCHAIN 7](#_Toc207395372)

[TOOLS FOR AGENTIC AI SYSTEMS 8](#_Toc207395373)

[EXAMPLE 8](#_Toc207395374)

[DETAILS 9](#_Toc207395375)

[CODE 11](#_Toc207395376)

[BUILDING ReAct AGENT 15](#_Toc207395377)

[ReACT PATTERN IN AGENTIC AI 16](#_Toc207395378)

[STEPS FOLLOWED BY ReAct AGENT 16](#_Toc207395379)

[ReAct AGENT PROMPT FORMAT 17](#_Toc207395380)

[HOW AGENTS WORK 18](#_Toc207395381)

[IMPORTANT NOTES 19](#_Toc207395382)

[CREATING ReACT AGENT 20](#_Toc207395383)

[ReAct AGENT USING LANGGRAPH 20](#_Toc207395384)

[BUILD YOUR OWN LOOP 31](#_Toc207395385)

[TOOL CALLING 38](#_Toc207395386)

# AGENTIC AI

A diagram of a tool

AI-generated content may be incorrect.

* **Agentic AI** refers to AI systems that can **autonomously make decisions**, **take actions**, and **pursue goals** with minimal human intervention.
* These systems behave like **agents** — they don’t just respond to prompts, they **act independently** based on objectives, rules, and environmental feedback.

## TRADITIONAL VERSUS AGENTIC AI

Traditional AI

* **Task-specific**: Designed for narrow, well-defined tasks (e.g., image recognition, spam filtering).
* **Static logic**: Operates based on pre-programmed rules or trained models.
* **No autonomy**: Executes tasks without decision-making or goal-setting.
* **Pipeline-based**: Often part of a larger system, but not self-directed.

**Examples:**

* Machine learning models for fraud detection.
* NLP models for sentiment analysis.
* Recommendation engines.

Agentic AI

* **Goal-driven**: Operates with objectives and can plan steps to achieve them.
* **Autonomous**: Makes decisions, adapts to new information, and can self-correct.
* **Tool-using**: Can call APIs, search the web, write code, and interact with environments.
* **Memory-enabled**: Remembers past interactions and uses them to improve performance.
* **Multi-modal**: Can handle text, images, audio, and more in a unified workflow.

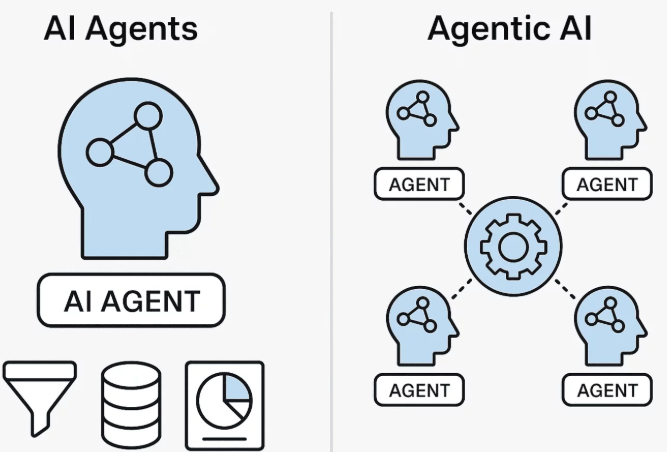
**Examples:**

* AI agents built with **LangGraph**, **AutoGPT**, or **OpenAI's Assistants API**.
* Agents that research, write reports, book meetings, or debug code.
* Agents that collaborate with other agents or humans in real-time.

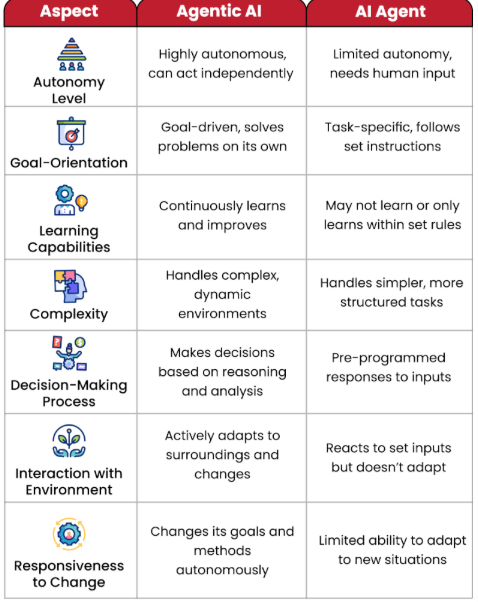
## EXAMPLES

|  |
| --- |
| AI Health Assistant (Personal Agent)  Scenario: A patient says: "Help me manage my diabetes."  *Agentic AI Actions:*   1. Retrieves your medical history and prescriptions. 2. Sets reminders for medication and meals. 3. Orders refills from the pharmacy. 4. Alerts your doctor if blood sugar readings are abnormal. 5. Adjusts your diet plan based on recent activity.   ✅ Autonomous, goal-driven, multi-step behavior. |
| AI Executive Assistant (e.g., AutoGPT)  Scenario:You say: "Plan a business trip to Mumbai next week."  *Agentic AI Actions:*   1. Searches for flights and books the best one. 2. Reserves a hotel near your meeting location. 3. Schedules meetings with clients. 4. Adds everything to your calendar. 5. Sends confirmation emails.   ✅ It acts like a human assistant, not just a chatbot. |
| 3. DevOps Agent  Scenario: "Monitor my app and scale it if traffic spikes."  *Agentic AI Actions:*   * Monitors server metrics. * Detects a traffic spike. * Automatically scales up resources. * Sends a Slack alert to the team. * Rolls back if errors increase.   ✅ Autonomous infrastructure management. |
| E-commerce Store Manager Bot  Scenario:"Manage my online store."  *Agentic AI Actions:*   1. Updates product listings. 2. Adjusts prices based on competitor data. 3. Responds to customer queries. 4. Flags suspicious orders. 5. Launches a weekend sale campaign.   ✅ Acts with initiative and adapts to changing conditions. |

## AGENTIC AI VERSUS AI AGENTS



|  |  |  |
| --- | --- | --- |
| Parameters | AI AGENTS | AGENTIC AI |
| Definition | * Ai Agents refers to an individual software program designed to perform specific task without any human intervention. * They are single entity with focused purpose | * Agentic AI describes a broader framework where multiple Ai Agents can collaborate and make decisions independently to achieve larger goals without human intervention * Network of agents working together to achieve a goal |
| Functionality | * It is meant for solve specific task with limited autonomy | * Solves more complex workflows and focused on overall system with the ability to learn , adapt and make complex decisions. |
| Decision Making | * Follow Pre-defined Rule | * Take decisions based on real time data & circumstances |



### EXAMPLES - AI AGENTS

* **I Agents** are systems designed to **perform tasks autonomously**. They usually follow a loop like:
  + **Receive a goal or instruction**
  + **Plan how to achieve it**
  + **Take actions (like calling APIs, searching, writing code)**
  + **Evaluate results**
  + **Repeat until the goal is met**

Example**:**

* Imagine you ask an AI agent: “Find the cheapest flight to Mumbai and book it.” . Then the agent might:
  + Search flight options
  + Compare prices
  + Choose the best one
  + Book the ticket
* It acts like a **smart assistant** that can think and act on its own.

### EXAMPLE - AGENTIC AI

### AI AGENTS V/S AGENTIC AI

## DESIGN PATTERNS FOR AGENTIC AI

A diagram of a design

AI-generated content may be incorrect.

### REFLECTION PATTERN

* **What it is**: The agent thinks about what it did, checks for mistakes, and tries to improve i.e. Agents analyze to refine strategies.
* **Real-world analogy**: Like a student reviewing their test answers and correcting errors before submitting.
* **AI example**: You ask an AI to summarize a document. It gives a summary, then reflects: “Did I miss anything important?” It re-reads the document and updates the summary to include missing points.
* **Why it’s useful**: It helps the agent become more accurate , reduces repetitive mistakes and reliable over time.
* **Core Components** 
  + **Memory Module**
  + **Evaluator**
  + **Feedback Loop**

### TOOL USE PATTERN

* **What it is**: The agent selects & uses external tools to get things done. i.e it can connect to the live data
* **Real-world analogy**: Like a person using a calculator, Google Maps, or a dictionary to help with tasks.
* **AI example**: You ask, “What’s the weather in Noida?” The agent doesn’t guess—it uses a weather API to fetch the latest data and gives you the answer.
* **Why it’s useful**: It allows the agent to go beyond its built-in knowledge and access real-time or specialized information.
* **Core Components**
  + **API Connectors**
  + **Data Validators**

### PLANNING PATTERN

* **What it is**: The agent breaks a big task into smaller steps and follows a plan.
* **Real-world analogy**: Like planning a trip—first book flights, then hotels, then create an itinerary.
* **AI example**: You ask an AI to write a research paper. It first creates an outline, then writes each section step-by-step, checking progress along the way.
* **Why it’s useful**: It helps the agent stay organized and handle complex tasks more effectively.

### MULTI-AGENT PATTERN

* **What it is**: Multiple agents work together, each doing a specific job.
* **Real-world analogy**: Like a team project—one person does research, another writes, another edits.
* **AI example**: One agent specializes in medical knowledge, another in summarizing text, and another in generating visuals. You ask for a medical infographic, and they collaborate to produce it.
* **Why it’s useful**: It allows for specialization and teamwork, making the system more powerful and flexible.

How These Patterns Work Together – Home Health System



1. Planning Pattern:
   1. Schedule a video call with the doctor
2. Tool Use Pattern:
   1. Book an appointment with the doctor via clinics API
3. Reflection Pattern:
   1. Meanwhile – the reflection agent Analyze the call outcome to improve future scheduling
4. MultiAgent Pattern:
   1. Transfer the post appointment notes to a diet plan agent (another AI agent)

## CORE PILLARS OF AGENTIC AI

Planning

* The agent can **break down goals into steps**, decide what to do next, and adapt its strategy.
* Often uses **chain-of-thought prompting**, **decision trees**, or **LangGraph-style workflows**.
* Enables **autonomous task execution** and multi-step reasoning.

Tools

* Agents can **call external APIs**, **search the web**, **write code**, or **interact with databases**.
* Tools extend the agent’s capabilities beyond language—into action.
* Examples: Tavily search, Python execution, SQL queries, file handling.

Memory

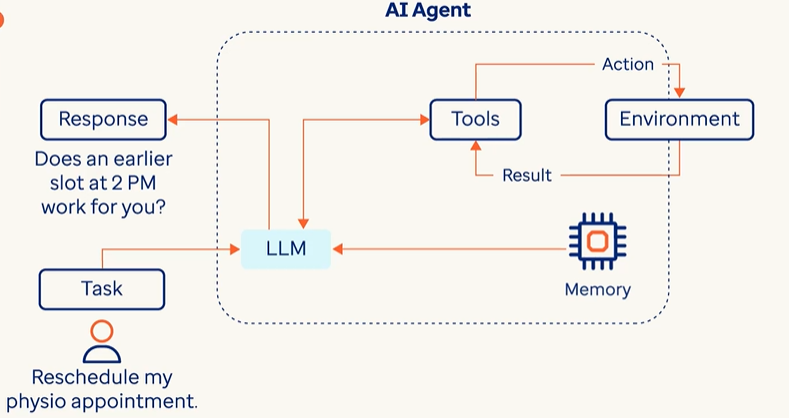
* Agents remember **past interactions**, **user preferences**, and **task history**.
* Supports **context continuity**, personalization, and long-term learning.
* Can be short-term (within a session) or long-term (across sessions).

How They Work Together

Imagine an agent tasked with booking a medical appointment:

* **Planning**: Breaks the task into steps—check history, find slots, confirm preference.
* **Tools**: Uses APIs to access EHR, calendar, and notification systems.
* **Memory**: Remembers patient preferences and past interactions.

### EXAMPLE



Task Initiation (Planning Begins)

* **Input**: *"Reschedule my physio appointment.".* This is the **goal** the agent receives.
* The **LLM** begins **planning** how to achieve this—breaking it into steps like checking availability, confirming preferences, and updating records.

Tool Usage

* The LLM interacts with **Tools** to:
  1. Query the calendar system.
  2. Check provider availability.
  3. Suggest alternate slots (e.g., *"Does an earlier slot at 2 PM work for you?"*).
  4. These tools act on the **Environment**—real-world systems like EHR(Electronic Health Record), scheduling APIs, or notification services.

|  |
| --- |
| * Vector DB and Python list stores this data enabling instant retrieval of data during decision loops |

Memory Integration

After tools act on the environment, the results (e.g., available slots, patient preferences) are stored in **Memory**.

This allows the agent to:

* Remember past appointments.
* Learn user preferences (e.g., prefers afternoon slots).
* Avoid redundant or conflicting actions.

Response Generation

* The LLM uses updated context from **Tools** and **Memory** to generate a meaningful response:
* *"Does an earlier slot at 2 PM work for you?"*
* This closes the loop, completing the agentic cycle.

# IMPLEMENTING TOOLS & TOOL CALLING

## TOOLS FOR AGENTIC AI SYSTEMS

* In Agentic AI systems, tools are functions that allow the large language model (LLM) to do things it cannot do on its own. These include actions like searching for information, looking up data from a database, performing calculations, or calling APIs.
* Tool calling allows the LLM to request a tool when it needs help answering a question. In LangChain, this is done using the **bind\_tools** method. This tells the model which tools are available, how they work, and when to use them when input queries are passed to the LLM.
* LanGraph can also be leveraged to create reactive agents by passing tools, prompts, and LLMs to built-in functions.
* **Note that the LLM will not call the tools automatically; they must be executed manually**

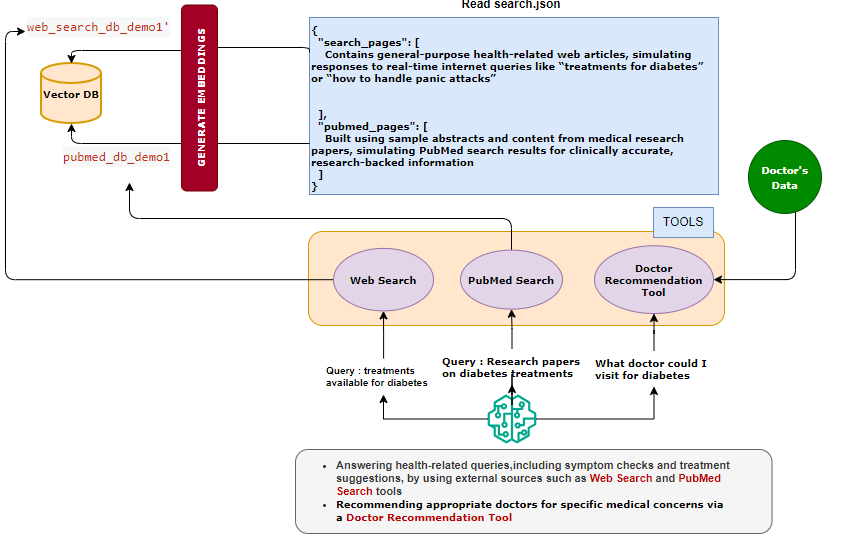
## EXAMPLE

NOTE: IN THE BELOW EXAMPLE WE WILL JUST SEE HOW TOOLS CAN BE CALLED MANUALLY

|  |
| --- |
| We aim to build **HealthBuddy**, an Agentic AI system designed to assist users with health-related queries through intelligent reasoning and tool use. Leveraging the **ReAct framework** (Reasoning + Acting), HealthBuddy integrates a **Large Language Model (LLM)** with custom tools and structured instructions to provide accurate, actionable, and personalized responses.  HealthBuddy is capable of:   * Answering health-related queries, including symptom checks and treatment suggestions, by using external sources such as **Web Search** and **PubMed Search** tools * Recommending appropriate doctors for specific medical concerns via a **Doctor Recommendation Tool** * Displaying available **appointment slots** and **booking appointments** after collecting essential user details (name, email, phone) * Maintaining **multi-turn conversations** in a natural, chat-like flow * Supporting **multi-user interactions** with isolated memory contexts for each user to enable personalized and continuous engagement |

A diagram of a computer

AI-generated content may be incorrect.



### DETAILS

STEP 1: Prepare Databases for Simulated Web Search and PubMed Search Tools

* We will load sample data simulating real web page documents and research papers from PubMed. This simulates real-time web or research paper searches using preloaded documents as the source.
* We will create two separate vector databases:
  + web\_search\_db: Contains general-purpose health-related web articles, simulating responses to real-time internet queries like “treatments for diabetes” or “how to handle panic attacks”
  + pubmed\_db: Built using sample abstracts and content from medical research papers, simulating PubMed search results for clinically accurate, research-backed information

Step 2: Preparing Database for Doctor Recommendations Tool

* We will create a small, in-memory database containing information about doctors. This data will be used by our Doctor Recommendation Tool to help users find the right doctor based on their health query or symptoms.
* The database includes a list of doctors along with their:
  + Name
  + Specialization (e.g., Dermatology, Pediatrics, Cardiology)
  + Location
  + Availability
  + Contact information
* We will use a simple Python list of dictionaries to store the doctor data. In a real-world application, this would typically be replaced by a backend database like PostgreSQL, MongoDB, or an external API.

Step 3: Create Tools for AI Agent

* LangChain makes it easy to create and register tools using the Tool class. The tool includes:
* A name and description
* The python function to be called
* An input schema that tells the model what arguments it can use
* When tools are properly defined, they enable the model to solve more complex problems by allowing it to perform actions and access external data. This makes the system more useful and reliable.
* These tools will allow the agent to retrieve information from our preloaded vector databases (web search and PubMed), as well as recommend doctors from our in-memory doctor database.
* The goal is to modularize the logic for different types of tasks into reusable components that can be invoked by the LLM when needed. These include:
  + A **Web Search Tool** that queries the web\_search\_db to simulate general online web search
  + A **PubMed Search Tool** that retrieves information from pubmed\_db for research-grade medical content
  + A **Doctor Recommendation Tool** that finds suitable doctors based on user symptoms or needs
* This tool-based setup is essential for enabling agentic behavior, where the LLM reasons through a problem, decides which action to take, and requests to call the right tools to gather more information or perform a task.

|  |  |
| --- | --- |
| how the LLM is deciding which tool to call in code?   * Tool Binding and Function Calling: The **bind\_tools** method converts the tools into OpenAI function specifications. When we bind tools to the LLM, each tool's metadata (name, description, parameters) becomes available to the model as function specifications. * Tool Descriptions: Each tool has **a descriptive docstring that helps the LLM understand when to use i**t: * **Decision Making Process**: When we send a prompt, the LLM analyzes:   + The prompt content   + The available tools and their descriptions   + The purpose of each tool * Response Types: The code checks for two types of responses:  |  | | --- | | **if result.content:**  **print("No tool call was needed")**  **# LLM decided it can answer directly**    **if result.tool\_calls:**  **print("LLM decided to call tools")**  **# LLM decided it needs to use a tool** |  * Decision Factors: The LLM chooses tools based on:   + If the query is about scientific research → search\_pubmed   + If the query is about finding a doctor → recommend\_doctor   + If the query is about general information → search\_web   + If the query can be answered without tools → direct response * For example:   + "Research papers on diabetes treatments" → search\_pubmed because it explicitly asks for research papers   + "What doctor could I visit" → recommend\_doctor because it's about finding a suitable doctor   + "Explain what is diabetes" → might get a direct response because it's a general knowledge question   **The LLM makes these decisions automatically based on:**   * + **The semantic meaning of the user's query**   + **The tool descriptions provided in the docstrings**   + **The tool's parameter specifications**   + **The expected return types of the tools** |

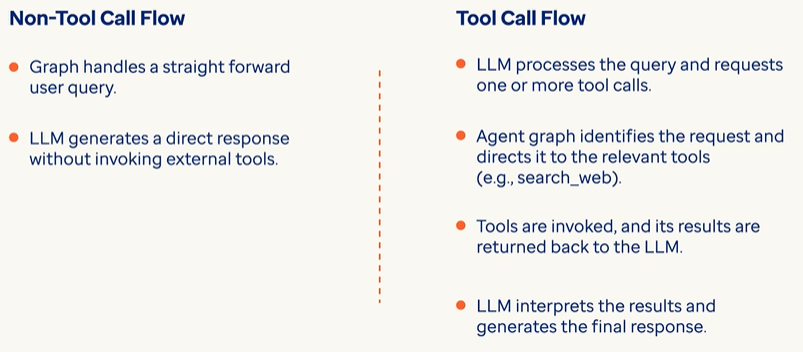
Step 4: Explore LLM Tool Calling with Custom Tools

* An agent is basically an LLM that has the capability to automatically call relevant functions to perform complex or tool-based tasks based on inputs or prompts provided by users.
* Tool calling, also popularly known as function calling, is the ability to reliably enable such LLMs to call external tools and APIs.

|  |
| --- |
| Tool calling in LangChain works by:   1. Registering defined tool functions using @tool decorator 2. Binding the tools to the model using llm.bind\_tools([tool1, tool2, ...]) 3. Passing a user query to the bound model 4. Letting the model decide whether to use a tool or respond directly  * This setup lets the model behave more like an agent that can take actions, observe results, and continue the conversation intelligently. * By using bind\_tools, we give the LLM the ability to understand what tools are available and make requests to use them only when needed. |

**Tool Call Requests**

* LLMs will not call and execute the tools, but will request tool calls based on reasoning on the input (user query) if they feel that they do not have enough information to answer the question directly.
* If the LLM can handle and generate the response without invoking the external tool is called “**Non-tool Call Flow**”



### CODE

|  |
| --- |
| from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI  import os  from dotenv import load\_dotenv  import json  from doctors import doctors\_db  from IPython.display import display, Markdown  # Import Chroma vector database for storing and searching embeddings  from langchain\_chroma import Chroma  from langchain\_core.tools import tool  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0,  )  # Initialize the Azure OpenAI Embeddings  embeddings\_client = AzureOpenAIEmbeddings(      azure\_deployment=embedding\_deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,  )  # Read the JSON File  def load\_json\_data\_source():      try:          current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))          json\_file\_path = os.path.join(current\_dir, "search.json")          if not os.path.exists(json\_file\_path):              print(f"Error: search.json not found at {json\_file\_path}")              return None          with open(json\_file\_path, "r") as f:              search\_docs = json.load(f)          return search\_docs      except json.JSONDecodeError as e:          print(f"Error decoding JSON: {e}")          return None      except Exception as e:          print(f"Unexpected error: {e}")          return None  # Load the JSON data  search\_docs = load\_json\_data\_source()  if search\_docs is None:      print("Failed to load search data")  else:      print(f"Major Document Types: {list(search\_docs.keys())}")  # Store the publications and web Articles in in vector Db in different namespaces  web\_search\_db = Chroma.from\_texts(      search\_docs["search\_pages"],      collection\_name="web\_search\_db\_demo1",      embedding=embeddings\_client,  )  pubmed\_db = Chroma.from\_texts(      search\_docs["pubmed\_pages"],      collection\_name="pubmed\_db\_demo1",      embedding=embeddings\_client,  )  ### Tools Definations  @tool  def search\_web(query: str) -> list:      """Search the web for a query. Useful for retrieving general or up-to-date healthcare information."""      # Perform semantic similarity search over the web search vector database      results = web\_search\_db.similarity\_search(query)      docs = [doc.page\_content for doc in results]      return docs  @tool  def search\_pubmed(query: str) -> list:      """Search PubMed for scientific articles related to the query."""      # Perform semantic similarity search over the PubMed vector database      results = pubmed\_db.similarity\_search(query)      docs = [doc.page\_content for doc in results]      return docs  # Tool for recommending a doctor based on user symptoms or health-related queries  @tool  def recommend\_doctor(query: str) -> dict:      """Recommend the most suitable doctor based on the user's symptoms."""      doctors\_list = str(doctors\_db)      # Use the LLM to reason over the list and identify the best match for the user's concern      prompt = f"""You are an assistant helping recommend a doctor based on a patient's health issues.  Here is the list of available doctors:  {doctors\_list}  Given the user's query: "{query}"  Choose the most suitable doctor from the list. Only pick one doctor.  Return only the selected doctor's information in JSON format.  If unsure, recommend the General Physician.  """      response = llm.invoke(prompt)      return {"recommended\_doctor": response.content}  tools = [search\_web, search\_pubmed, recommend\_doctor]  llm\_with\_tools = llm.bind\_tools(tools)  prompts = [      "treatments available for diabetes",      "Research papers on diabetes treatments",      "What doctor could I visit for diabetes",      "Explain what is diabetes in simple terms",  ]  results = []  for prompt in prompts:      result = llm\_with\_tools.invoke(prompt)      results.append(result)  for prompt, result in zip(prompts, results):      # If the model provided a direct response without using any tools      if result.content:          print("No tool call was needed")          print(f"Prompt: {prompt}")          print(f"Direct LLM Response: {result.content}")      # If the model determined that a tool should be called      if result.tool\_calls:          print("LLM decided to call tools")          print(f"Prompt: {prompt}")          print(f"Tool Call Request: {result.tool\_calls}")      print("-" \* 50)      print()  # results = search\_web.invoke("Recent treatments in Diabetes")  # print(f"Total documents: {len(results)}")  # print()  # display(Markdown((results[0][:3000])))  # results = search\_pubmed.invoke("Recent treatments in Diabetes")  # print(f"Total documents: {len(results)}")  # print()  # display(Markdown((results[1][:3000])))  # result = recommend\_doctor.invoke("Treatments for Diabetes")  # print(f"Raw Tool Output:\n{json.dumps(result, indent=2)}")  # print("-" \* 50)  # print(f"\nFormatted Tool Output:\n{result['recommended\_doctor']}") |
| OUTPUT  **Major Document Types: ['search\_pages', 'pubmed\_pages']**  **LLM decided to call tools**  **Prompt: treatments available for diabetes**  **Tool Call Request: [{'name': 'search\_web', 'args': {'query': 'treatments available for diabetes'}, 'id': 'call\_Ntg63iNie58G2bPVuZNotiXl', 'type': 'tool\_call'}]**  **--------------------------------------------------**  **LLM decided to call tools**  **Prompt: Research papers on diabetes treatments**  **Tool Call Request: [{'name': 'search\_pubmed', 'args': {'query': 'diabetes treatments'}, 'id': 'call\_qsGSDhexFNXXbPUGfopzuTDX', 'type': 'tool\_call'}]**  **--------------------------------------------------**  **LLM decided to call tools**  **Prompt: What doctor could I visit for diabetes**  **Tool Call Request: [{'name': 'recommend\_doctor', 'args': {'query': 'diabetes'}, 'id': 'call\_0XQ6raiZjwS8X30LkKvu3U4N', 'type': 'tool\_call'}]**  **--------------------------------------------------**  **No tool call was needed**  **Prompt: Explain what is diabetes in simple terms**  **Direct LLM Response: Diabetes is a health condition that occurs when the body has trouble using sugar (glucose) properly. Glucose is an important source of energy for our cells, but to use it, the body needs a hormone called insulin.**  **In diabetes, either the body doesn't make enough insulin, or it can't use the insulin it makes effectively. This leads to high levels of sugar in the blood, which can cause various health problems over time.**  **There are two main types of diabetes:**  **1. \*\*Type 1 Diabetes\*\*: The body doesn't produce insulin at all. This type usually starts in childhood or young adulthood.**    **2. \*\*Type 2 Diabetes\*\*: The body doesn't use insulin well or doesn't make enough insulin. This type is more common and often develops in adults, but it can also occur in children and teenagers.**  **Managing diabetes typically involves monitoring blood sugar levels, making healthy food choices, exercising, and sometimes taking medication or insulin.**  **--------------------------------------------------** |

## DIFFERENT WAYS OF TOOL CALLING

There are multiple ways to define agent tools

**1. Using the @tool Decorator**

|  |
| --- |
| from langchain.tools import tool  @tool  def get\_text\_length(text: str) -&gt; int:  return len(text)   * This is the **simplest and most declarative** way. * Automatically registers the function as a tool. * You can optionally pass metadata like name, description, etc. |

**2. Using the Tool Class Directly**

|  |
| --- |
| from langchain.tools import Tool  def get\_text\_length(text: str) -&gt; int:  return len(text)  tool = Tool.from\_function(  func=get\_text\_length,  name="GetTextLength",  description="Returns the length of the input text"  )   * Gives **more control** over tool metadata. * Useful when you want to dynamically create tools or wrap existing functions. |

**3. Using StructuredTool for Typed Inputs**

|  |
| --- |
| from langchain.tools import StructuredTool  def get\_text\_length(text: str) -&gt; int:  return len(text)  tool = StructuredTool.from\_function(  func=get\_text\_length,  name="GetTextLength",  description="Returns the length of the input text"  )   * Ensures **structured input/output** using type hints. * Ideal for tools that require **multiple or complex parameters**. |

**4. Using BaseTool for Custom Behavior**

|  |
| --- |
| from langchain.tools import BaseTool  class CustomTool(BaseTool):  name = "CustomTextLength"  description = "Custom tool to get text length"  def \_run(self, text: str) -&gt; int:  return len(text)  def \_arun(self, text: str) -&gt; int:  raise NotImplementedError("Async not supported")  tool = CustomTool()   * Allows **full customization** of tool behavior. * You can override \_run and \_arun for sync/async execution.   Useful for **advanced use cases** like logging, error handling, or chaining. |

**5. Wrapping External APIs or Services**

|  |
| --- |
| We can wrap any external API call as a tool:  import requests  from langchain.tools import Tool  def get\_weather(city: str) -&gt; str:  response = requests.get(f"https://api.weatherapi.com/v1/current.json?q={city}")  return response.json()["current"]["condition"]["text"]  weather\_tool = Tool.from\_function(  func=get\_weather,  name="GetWeather",  description="Returns current weather for a city"  )  • Enables integration with external services.  • Can be used to build tools for databases, REST APIs, etc. |

**6. Using Tool Collections**

|  |
| --- |
| You can group tools together:  from langchain.agents import initialize\_agent  from langchain.agents.agent\_types import AgentType  tools = [tool1, tool2, tool3]  agent = initialize\_agent(tools, llm, agent=AgentType.ZERO\_SHOT\_REACT\_DESCRIPTION)   * Useful for **multi-tool agents**. * Each tool can be defined using any of the above methods. |

# BUILDING ReAct AGENT

## ReACT PATTERN IN AGENTIC AI

* The ReAct pattern is inspired by the problem-solving approach of Humans. For example, A medical doctor diagnosing a patient. The doctor thinks through the symptoms, observer test results and act prescribe treatment.
* This iterative process of analyzing data, adopting new information and executing decisions led the foundation of ReAct(Reasoning & Acting Framework) for Agentic AI

|  |  |
| --- | --- |
| A screenshot of a cell phone  AI-generated content may be incorrect. | Similar to humans AI agent using ReAct will   1. Decompose the complex tasks into multiple structured steps 2. Integrate real time feedback 3. Interact with tools to achieve goals |

### STEPS FOLLOWED BY ReAct AGENT

Let us understand the steps of React with an example

#### REASONING

A diagram of a disease

AI-generated content may be incorrect.

1. Let the user as questions on his medical problem. The LLM will first Think followed by planning.

#### ACTION

A diagram of a web search

AI-generated content may be incorrect.

* In this step for the user’s query LLM will make use of tolls (tool calling) for web search and tool to fetch Electronic health Record

#### REASONING

A diagram of a process

AI-generated content may be incorrect.

Note :

* React uses Chain of thought prompting which enables LLMs to reason better
* Tools & Function calling which enables to take better action

### ReAct AGENT PROMPT FORMAT

* When we use a **ReAct agent** (in LangChain, LangGraph, or a custom setup), the **prompt usually follows a specific format** because the agent needs a structured way to **reason** and **decide which tool to call**.

The general **ReAct prompt pattern** looks like this:

|  |  |
| --- | --- |
| **Typical ReAct Prompt Format** | **Example** |
| Question: <the user’s question>  Thought: <agent reasons about what to do>  Action: <the tool to call>  Action Input: <input to the tool>  Observation: <result returned by the tool>  ... (repeat Thought → Action → Observation) ...  Final Answer: <the final answer to the user> | Suppose the user asks:  “What’s the weather in Delhi today, and what’s 5 + 7?”  The agent’s reasoning trace might look like:  Question: What’s the weather in Delhi today, and what’s 5 + 7?  Thought: I need to get the weather for Delhi.  Action: WeatherAPI  Action Input: "Delhi"  Observation: It’s 34°C and sunny in Delhi.  Thought: Now I need to calculate 5 + 7.  Action: Calculator  Action Input: "5+7"  Observation: 12  Thought: I now have both answers.  Final Answer: The weather in Delhi is 34°C and sunny, and 5 + 7 = 12. |

#### WHY THIS FORMAT?

* **Thought** → lets the model reason step by step (chain-of-thought).
* **Action & Action Input** → tell the framework which tool to call.
* **Observation** → captures the tool’s response.
* **Final Answer** → ensures a clean response for the user.

|  |
| --- |
| * LangChain and LangGraph **inject this format automatically** into the system prompt. * So, we usually **don’t need to handcraft it** every time — but if we are building a custom agent, we must follow this structure for ReAct to work properly. |

## HOW AGENTS WORK

A diagram of a tool execution

AI-generated content may be incorrect.

1. User Query Initiation

* The process begins when a **user submits a query**.
* This query is passed to the **React Agent**, which is a system designed to reason and act using tools and an LLM (Large Language Model).

2. LLM Reasoning (First Pass)

* The agent makes a **special LLM call** to interpret the query.
* The LLM analyzes the intent and decides:
  + Whether a tool is needed.
  + Which tool is appropriate.
  + What parameters might be required.

3. LLM Output Parsing

* The LLM returns a **textual response** (not structured code).
* This response needs to be **parsed** to extract:
  + The tool name.
  + The input parameters.
  + Any intermediate reasoning.

**This parsing step is crucial because LLMs typically return natural language, not executable code.**

4. Tool Execution

* Based on the parsed output, the agent:
  + Selects the correct tool.
  + Executes the tool with the provided input.
* The tool returns a **structured response**.

5. Second Iteration with LLM

* The agent now has:
  + The original query.
  + The tool’s output.
* It runs **another LLM call** to:
  + Decide if another tool is needed.
  + Refine the reasoning.
  + Determine if it can now answer the query.

6. Final Decision

* The agent evaluates:
  + Do I have enough information to answer?
  + Should I call another tool?
  + Should I return the final answer?

This loop continues until the agent is confident it can return a complete and correct response.

Summary of React Agent Workflow

* **React** stands for **Reason + Act**.
* It combines **LLM reasoning** with **tool execution** in a loop:
  1. **Reason** with LLM.
  2. **Act** by calling tools.
  3. **Repeat** until the answer is ready.

### IMPORTANT NOTES

|  |
| --- |
| * The ReAct Agent always has structured output format. * Below is **Structured Output Format (used by the agent).** This is what the **LLM is expected to generate** during execution. It includes:   ***Question: <user question>***  ***Thought: <reasoning>***  ***Action: <tool name>***  ***Action Input: <input to the tool>***  ***Observation: <result from the tool>***  ***Thought: <updated reasoning>***  ***Final Answer: <final answer>***   * This format is parsed by tools like **ReActSingleInputOutputParser** to extract actionable steps.   Prompt Format (used to instruct the LLM):  This is what **we write and feed into the LLM** to guide it to produce the structured output. For example:  ***template = """***  ***Answer the following questions as best you can. You have access to the following tools:***  ***get\_text\_length(text: str) -> int - Get the length of the text in tokens.***  ***Use the following format:***  ***Question: the input question you must answer***  ***Thought: you should always think about what to do***  ***Action: the action to take, should be one of [get\_text\_length] Action***  ***Action Input: the input to the action***  ***Observation: the result of the action***  ***... (this Thought/Action/Action Input/Observation can repeat N times)***  ***Thought: I now know the final answer***  ***Final Answer: the final answer to the original input question***  ***Begin!***  ***Question: {input}***  ***Thought::***  ***"""***   * **NOTE : This prompt teaches the LLM to follow the ReAct-style output format.** |

## CREATING ReACT AGENT

We can create ReAct agents using

Built-in utility function : **create\_react\_agent()**

* **Uses LangGraph's pre-built create\_react\_agent() function**
* **State management is handled automatically by the pre-built agent**
* **No explicit state definition needed**
* **Automatic tool calling and message routing**
* **ReAct pattern implemented internally**
* **Less customizable - follows pre-defined ReAct pattern**
* **Simpler but more constrained**

**BUILD FROM SCRATCH**

We need to

* Manually implements the entire agent architecture using LangGraph primitives
* Custom state management, nodes, and graph construction:
* Explicit state schema definition
* Manual graph construction with explicit nodes and edges:
* Highly customizable - full control over agent behavior
* Can modify reasoning logic, add custom nodes, change routing logic
* More complex but flexible

### ReAct AGENT BUILT-IN UTILITY FUNCTIONS: **create\_react\_agent()**

#### SIMPLE CALCULATR USING create\_react\_agent()

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from langchain\_core.messages import HumanMessage, SystemMessage  from langgraph.prebuilt import create\_react\_agent  from langchain\_core.tools import tool  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0,  )  # Tools defination  @tool  def add(op1: int, op2: int) -> int:      """      Adds two integers and returns their sum.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The sum of op1 and op2.      """      """Add two numbers."""      return op1 + op2  @tool  def subtract(op1: int, op2: int) -> int:      """      Subtracts the second integer from the first and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The result of op1 minus op2.      """      """Subtract two numbers."""      return op1 - op2  @tool  def multiply(op1: int, op2: int) -> int:      """      Multiplies two integers and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The product of op1 and op2.      """      """Multiply two numbers."""      return op1 \* op2  @tool  def divide(op1: int, op2: int) -> float:      """      Divides the first integer by the second and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          float: The result of op1 divided by op2.      """      """Divide two numbers."""      return op1 / op2  tools = [add, subtract, multiply, divide]  llm\_with\_tools = llm.bind\_tools(tools)  # System prompt for the agent  AGENT\_PROMPT\_TXT = """  You are a calculator agent. You can perform the following operations:  1. Addition  2. Subtraction  3. Multiplication  4. Division  Please provide your input in the following format:  {      "operation": "add|subtract|multiply|divide",      "operands": [operand1, operand2]  }  """  AGENT\_SYS\_PROMPT = SystemMessage(content=AGENT\_PROMPT\_TXT)  calculator\_agent = create\_react\_agent(      model=llm\_with\_tools, tools=tools, prompt=AGENT\_SYS\_PROMPT  )  def call\_agent(agent, query, verbose=False):      # Stream the agent's execution for the given query      for event in agent.stream(          {"messages": [HumanMessage(content=query)]},  # Input prompt          stream\_mode="values",  # Stream output as intermediate values      ):          # If verbose is enabled, print each intermediate message          if verbose:              event["messages"][-1].pretty\_print()      # Return the final message content for optional downstream use      return event["messages"][-1].content  # Example usage  query = "What is the result of (15 + 5) \* (10 - 2) / 4 ?"  result = call\_agent(calculator\_agent, query)  print(result) |
| OUTPUT  The result of the expression \((15 + 5) \* (10 - 2) / 4\) is \(40.0\). |

#### HEALTH ASSISTANT

**Objective**:

* Build a ReAct-based Tool-Use Agent using LangGraph with built-in utility functions like **create\_react\_agent()**
* Equip the agent with multiple tools (**web search, PubMed search, doctor recommendation**)
* Handle user queries end-to-end—from interpreting intent to delivering a useful, cited, and grounded response
* Simulate a multi-step reasoning and tool-using workflow

**Goal**: Working agent that can research a health query, gather relevant information, and offer advice including suggesting a doctor—all through structured decision-making and tool usage.

**Agent Architecture**

The following figure shows the agent architecture, including all components and the overall workflow:

A diagram of a health care system

AI-generated content may be incorrect.

|  |
| --- |
| **from langchain\_openai import AzureOpenAIEmbeddings, AzureChatOpenAI**  **import os**  **from dotenv import load\_dotenv**  **import json**  **from doctors import doctors\_db**  **from IPython.display import display, Markdown**  **from langchain\_core.messages import HumanMessage, SystemMessage**  **from langgraph.prebuilt import create\_react\_agent**  **# Import Chroma vector database for storing and searching embeddings**  **from langchain\_chroma import Chroma**  **from langchain\_core.tools import tool**  **load\_dotenv()**  **# Set your Azure OpenAI credentials**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]**  **embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]**  **# Initialize the Azure OpenAI LLM**  **llm = AzureChatOpenAI(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **temperature=0,**  **)**  **# Initialize the Azure OpenAI Embeddings**  **embeddings\_client = AzureOpenAIEmbeddings(**  **azure\_deployment=embedding\_deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **# Read the JSON File**  **def load\_json\_data\_source():**  **try:**  **current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))**  **json\_file\_path = os.path.join(current\_dir, "search.json")**  **if not os.path.exists(json\_file\_path):**  **print(f"Error: search.json not found at {json\_file\_path}")**  **return None**  **with open(json\_file\_path, "r") as f:**  **search\_docs = json.load(f)**  **return search\_docs**  **except json.JSONDecodeError as e:**  **print(f"Error decoding JSON: {e}")**  **return None**  **except Exception as e:**  **print(f"Unexpected error: {e}")**  **return None**  **# Load the JSON data**  **search\_docs = load\_json\_data\_source()**  **if search\_docs is None:**  **print("Failed to load search data")**  **else:**  **print(f"Major Document Types: {list(search\_docs.keys())}")**  **# Store the publications and web Articles in in vector Db in different namespaces**  **web\_search\_db = Chroma.from\_texts(**  **search\_docs["search\_pages"],**  **collection\_name="web\_search\_db\_demo1",**  **embedding=embeddings\_client,**  **)**  **pubmed\_db = Chroma.from\_texts(**  **search\_docs["pubmed\_pages"],**  **collection\_name="pubmed\_db\_demo1",**  **embedding=embeddings\_client,**  **)**  **### Tools Definations**  **@tool**  **def search\_web(query: str) -> list:**  **"""Search the web for a query. Useful for retrieving general or up-to-date healthcare information."""**  **# Perform semantic similarity search over the web search vector database**  **results = web\_search\_db.similarity\_search(query)**  **docs = [doc.page\_content for doc in results]**  **return docs**  **@tool**  **def search\_pubmed(query: str) -> list:**  **"""Search PubMed for scientific articles related to the query."""**  **# Perform semantic similarity search over the PubMed vector database**  **results = pubmed\_db.similarity\_search(query)**  **docs = [doc.page\_content for doc in results]**  **return docs**  **# Tool for recommending a doctor based on user symptoms or health-related queries**  **@tool**  **def recommend\_doctor(query: str) -> dict:**  **"""Recommend the most suitable doctor based on the user's symptoms."""**  **doctors\_list = str(doctors\_db)**  **# Use the LLM to reason over the list and identify the best match for the user's concern**  **prompt = f"""You are an assistant helping recommend a doctor based on a patient's health issues.**  **Here is the list of available doctors:**  **{doctors\_list}**  **Given the user's query: "{query}"**  **Choose the most suitable doctor from the list. Only pick one doctor.**  **Return only the selected doctor's information in JSON format.**  **If unsure, recommend the General Physician.**  **"""**  **response = llm.invoke(prompt)**  **return {"recommended\_doctor": response.content}**  **tools = [search\_web, search\_pubmed, recommend\_doctor]**  **# Instruction prompt for the overall Agent**  **AGENT\_PROMPT\_TXT = """You are an agent designed to act as an expert in researching medical symptoms**  **and recommending relevant doctors for booking appointments. Also, remember the current year is 2025.**  **Given a user query, call the relevant tools and provide the most appropriate response.**  **Follow these guidelines to help you make more informed decisions:**  **- If the user's query specifically asks for a doctor recommendation, recommend an appropriate doctor.**  **- If the user is researching specific aspects related to symptoms, treatments, or other areas of healthcare,**  **use both the search\_web and search\_pubmed tools to gather detailed information and provide a well-structured response.**  **- If the user is just looking for general healthcare information, web search alone is sufficient.**  **- Use the search\_pubmed tool only if the query relates to information typically found in PubMed articles.**  **- Responses should include cited source links and/or PubMed article titles and publication dates, if available.**  **- If recommending doctors, use the recommend\_doctor tool and display detailed information in a clear, structured format.**  **Also, encourage the user to book an appointment via email.**  **- Politely decline to answer any queries that are not related to medical or healthcare information.**  **The final response should contain the following:**  **- The main output content.**  **- At the end, include an Agent Reasoning: section that covers the following in bullet points:**  **- Your step-by-step reasoning process for arriving at the final response.**  **- The tools you called, in the specific order, with their names.**  **- Observations from the tool call results that helped you construct the final response.**  **"""**  **AGENT\_SYS\_PROMPT = SystemMessage(content=AGENT\_PROMPT\_TXT)**  **# Create the agent using tools and LLM**  **healthbuddy\_agent = create\_react\_agent(model=llm, tools=tools, prompt=AGENT\_SYS\_PROMPT)**  **# Utility function to call the agent and stream its step-by-step reasoning**  **def call\_agent(agent, query, verbose=False):**  **# Stream the agent's execution for the given query**  **for event in agent.stream(**  **{"messages": [HumanMessage(content=query)]},  # Input prompt**  **stream\_mode="values",  # Stream output as intermediate values**  **):**  **# If verbose is enabled, print each intermediate message**  **if verbose:**  **event["messages"][-1].pretty\_print()**  **# Display the final response from the agent as Markdown**  **print("\n\nFinal Response:\n")**  **print(event["messages"][-1].content)**  **# Return the final message content for optional downstream use**  **return event["messages"][-1].content**  **# Example usage**  **query = (**  **"what are the latest methods for diabetes management and recommend a doctor please"**  **)**  **result = call\_agent(healthbuddy\_agent, query, verbose=True)**  **# # Example usage**  **# query = "I am having panic attacks, what could I do?"**  **# result = call\_agent(healthbuddy\_agent, query, verbose=True)**  **# # Example usage without printing detailed log messages**  **# query = "I am having panic attacks, please recommend a right doctor"**  **# result = call\_agent(healthbuddy\_agent, query, verbose=False)**  **# # Agent limitation**  **# query = "Great can you book an appointment please"**  **# result = call\_agent(healthbuddy\_agent, query, verbose=True)** |
| **OUTPUT**  **================================ Human Message =================================**  **what are the latest methods for diabetes management and recommend a doctor please**  **================================== Ai Message ==================================**  **Tool Calls:**  **search\_web (call\_H8l3urRrtpPi9ppoFfNyW5A8)**  **Call ID: call\_H8l3urRrtpPi9ppoFfNyW5A8**  **Args:**  **query: latest methods for diabetes management 2025**  **recommend\_doctor (call\_85eAACdF2sJ9FxATSziHdpS0)**  **Call ID: call\_85eAACdF2sJ9FxATSziHdpS0**  **Args:**  **query: diabetes management**  **================================= Tool Message =================================**  **Name: recommend\_doctor**  **{"recommended\_doctor": "```json\n{\n \"name\": \"Dr. Janet Dyne\",\n \"specialization\": \"Endocrinology (Diabetes Care)\",\n \"available\_timings\": \"10:00 AM - 1:00 PM\",\n \"location\": \"City Health Clinic\",\n \"contact\": \"janet.dyne@healthclinic.com\"\n}\n```"}**  **================================== Ai Message ==================================**  **### Latest Methods for Diabetes Management**  **In 2025, diabetes management has seen significant advancements, particularly in personalized medicine and technology integration. Here are some of the latest methods:**  **1. \*\*Personalized Medicine\*\*: This approach tailors treatment plans based on individual genetic makeup, lifestyle factors, and health characteristics. Genetic screening can predict diabetes susceptibility and optimize medication choices, enhancing treatment effectiveness.**  **2. \*\*Continuous Glucose Monitoring (CGM)\*\*: CGM devices provide real-time glucose data, allowing for proactive adjustments in insulin therapy. These devices can be integrated with insulin pumps to create closed-loop systems, often referred to as artificial pancreas systems.**  **3. \*\*Telemedicine and Remote Monitoring\*\*: These technologies facilitate remote patient monitoring and virtual consultations, improving accessibility to care and enabling timely adjustments to treatment plans based on real-time data.**  **4. \*\*Lifestyle Interventions\*\*: Personalized dietary plans and physical activity recommendations are crucial. Tailored interventions have shown to improve glycemic control and promote weight management effectively.**  **5. \*\*Pharmacogenomics\*\*: This field studies how genetic variations affect individual responses to diabetes medications, allowing for more effective and safer drug therapies.**  **6. \*\*Behavioral Interventions\*\*: Techniques such as cognitive-behavioral therapy (CBT) and motivational interviewing are being used to enhance treatment adherence and empower patients in their self-management.**  **7. \*\*Innovative Technologies\*\*: New advancements in diabetes technology, including more comfortable CGM sensors and mobile health applications, are making diabetes management more user-friendly and effective.**  **For more detailed information, you can refer to the [American Diabetes Association's Standards of Care 2025](https://diabetes.org/newsroom/press-releases/american-diabetes-association-releases-standards-care-diabetes-2025) and the article on [Advances in the Management of Diabetes Mellitus](https://pmc.ncbi.nlm.nih.gov/articles/PMC10505357/).**  **### Recommended Doctor for Diabetes Management**  **I recommend booking an appointment with:**  **\*\*Dr. Janet Dyne\*\***  **- \*\*Specialization\*\*: Endocrinology (Diabetes Care)**  **- \*\*Available Timings\*\*: 10:00 AM - 1:00 PM**  **- \*\*Location\*\*: City Health Clinic**  **- \*\*Contact\*\*: [janet.dyne@healthclinic.com](mailto:janet.dyne@healthclinic.com)**  **Feel free to reach out to Dr. Dyne via email to schedule your appointment.**  **---**  **### Agent Reasoning:**  **- I first searched for the latest methods in diabetes management using the `search\_web` tool, focusing on current advancements and guidelines.**  **- Simultaneously, I used the `recommend\_doctor` tool to find a suitable doctor specializing in diabetes management.**  **- The results provided comprehensive insights into personalized medicine, technology integration, and lifestyle interventions, which are crucial for effective diabetes management.**  **- The recommended doctor, Dr. Janet Dyne, is specialized in endocrinology, making her a suitable choice for diabetes care.**  **Final Response:**  **### Latest Methods for Diabetes Management**  **In 2025, diabetes management has seen significant advancements, particularly in personalized medicine and technology integration. Here are some of the latest methods:**  **1. \*\*Personalized Medicine\*\*: This approach tailors treatment plans based on individual genetic makeup, lifestyle factors, and health characteristics. Genetic screening can predict diabetes susceptibility and optimize medication choices, enhancing treatment effectiveness.**  **2. \*\*Continuous Glucose Monitoring (CGM)\*\*: CGM devices provide real-time glucose data, allowing for proactive adjustments in insulin therapy. These devices can be integrated with insulin pumps to create closed-loop systems, often referred to as artificial pancreas systems.**  **3. \*\*Telemedicine and Remote Monitoring\*\*: These technologies facilitate remote patient monitoring and virtual consultations, improving accessibility to care and enabling timely adjustments to treatment plans based on real-time data.**  **4. \*\*Lifestyle Interventions\*\*: Personalized dietary plans and physical activity recommendations are crucial. Tailored interventions have shown to improve glycemic control and promote weight management effectively.**  **5. \*\*Pharmacogenomics\*\*: This field studies how genetic variations affect individual responses to diabetes medications, allowing for more effective and safer drug therapies.**  **6. \*\*Behavioral Interventions\*\*: Techniques such as cognitive-behavioral therapy (CBT) and motivational interviewing are being used to enhance treatment adherence and empower patients in their self-management.**  **7. \*\*Innovative Technologies\*\*: New advancements in diabetes technology, including more comfortable CGM sensors and mobile health applications, are making diabetes management more user-friendly and effective.**  **For more detailed information, you can refer to the [American Diabetes Association's Standards of Care 2025](https://diabetes.org/newsroom/press-releases/american-diabetes-association-releases-standards-care-diabetes-2025) and the article on [Advances in the Management of Diabetes Mellitus](https://pmc.ncbi.nlm.nih.gov/articles/PMC10505357/).**  **### Recommended Doctor for Diabetes Management**  **I recommend booking an appointment with:**  **\*\*Dr. Janet Dyne\*\***  **- \*\*Specialization\*\*: Endocrinology (Diabetes Care)**  **- \*\*Available Timings\*\*: 10:00 AM - 1:00 PM**  **- \*\*Location\*\*: City Health Clinic**  **- \*\*Contact\*\*: [janet.dyne@healthclinic.com](mailto:janet.dyne@healthclinic.com)**  **Feel free to reach out to Dr. Dyne via email to schedule your appointment.**  **---**  **### Agent Reasoning:**  **- I first searched for the latest methods in diabetes management using the `search\_web` tool, focusing on current advancements and guidelines.**  **- Simultaneously, I used the `recommend\_doctor` tool to find a suitable doctor specializing in diabetes management.**  **- The results provided comprehensive insights into personalized medicine, technology integration, and lifestyle interventions, which are crucial for effective diabetes management.**  **- The recommended doctor, Dr. Janet Dyne, is specialized in endocrinology, making her a suitable choice for diabetes care.** |

### ReAct AGENT BUILT FROM SCRATCH

#### SIMPLE CALCULATOR

|  |
| --- |
| from langchain\_openai import AzureChatOpenAI  import os  from dotenv import load\_dotenv  from langchain\_core.messages import HumanMessage, SystemMessage  from langchain\_core.tools import tool  from langgraph.graph import StateGraph, START, END  from langgraph.prebuilt import ToolNode, tools\_condition  from typing import Annotated  from typing\_extensions import TypedDict  from langgraph.graph.message import add\_messages  load\_dotenv()  # Set your Azure OpenAI credentials  subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]  endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]  api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]  deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]  embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]  # Initialize the Azure OpenAI LLM  llm = AzureChatOpenAI(      azure\_deployment=deployment,      api\_version=api\_version,      azure\_endpoint=endpoint,      api\_key=subscription\_key,      temperature=0,  )  # Tools defination  @tool  def add(op1: int, op2: int) -> int:      """      Adds two integers and returns their sum.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The sum of op1 and op2.      """      """Add two numbers."""      return op1 + op2  @tool  def subtract(op1: int, op2: int) -> int:      """      Subtracts the second integer from the first and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The result of op1 minus op2.      """      """Subtract two numbers."""      return op1 - op2  @tool  def multiply(op1: int, op2: int) -> int:      """      Multiplies two integers and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          int: The product of op1 and op2.      """      """Multiply two numbers."""      return op1 \* op2  @tool  def divide(op1: int, op2: int) -> float:      """      Divides the first integer by the second and returns the result.      Args:          op1 (int): The first operand.          op2 (int): The second operand.      Returns:          float: The result of op1 divided by op2.      """      """Divide two numbers."""      return op1 / op2  tools = [add, subtract, multiply, divide]  llm\_with\_tools = llm.bind\_tools(tools)  # Utility function to call the agent and stream its step-by-step reasoning  def call\_agent(agent, query, verbose=False):      # Stream the agent's execution for the given query      for event in agent.stream(          {"messages": [HumanMessage(content=query)]},  # Input prompt          stream\_mode="values",  # Stream output as intermediate values      ):          # If verbose is enabled, print each intermediate message          if verbose:              event["messages"][-1].pretty\_print()      # Display the final response from the agent as Markdown      print("\n\nFinal Response:\n")      print(event["messages"][-1].content)      # Return the final message content for optional downstream use      return event["messages"][-1].content  # System prompt for the agent  AGENT\_PROMPT\_TXT = """  You are a calculator agent. You can perform the following operations:  1. Addition  2. Subtraction  3. Multiplication  4. Division  Please provide your input in the following format:  {      "operation": "add|subtract|multiply|divide",      "operands": [operand1, operand2]  }  """  AGENT\_SYS\_PROMPT = SystemMessage(content=AGENT\_PROMPT\_TXT)  # Define the agent's state schema for storing the message history  class State(TypedDict):      messages: Annotated[list, add\_messages]  # Create the node function that handles reasoning and planning using the LLM  def tool\_calling\_llm(state: State) -> State:      # Extract the current conversation history from the state      current\_state = state["messages"]      # Prepend the system instructions to the current message history      state\_with\_instructions = [AGENT\_SYS\_PROMPT] + current\_state      # Call the LLM to generate a new message (either a response or a tool call request)      response = [llm\_with\_tools.invoke(state\_with\_instructions)]      # Return the updated state containing the new message      return {"messages": response}  # Build the graph  builder = StateGraph(State)  # Add nodes  builder.add\_node("tool\_calling\_llm", tool\_calling\_llm)  builder.add\_node("tools", ToolNode(tools=tools))  # Add edges  builder.add\_edge(START, "tool\_calling\_llm")  # Conditional edge  builder.add\_conditional\_edges(      "tool\_calling\_llm",      tools\_condition,  # conditional routing function      # If the latest message (result) from LLM is a tool call request -> tools\_condition routes to tools      # If the latest message (result) from LLM is a not a tool call -> tools\_condition routes to END      ["tools", END],  )  builder.add\_edge(      "tools", "tool\_calling\_llm"  )  # this is the key feedback loop in the agentic system  # Compile Agent Graph  calculator\_agent = builder.compile()  # Example usage  query = "What is the result of (15 + 5) \* (10 - 2) / 4 ?"  result = call\_agent(calculator\_agent, query, verbose=True) |
| ================================ Human Message =================================  What is the result of (15 + 5) \* (10 - 2) / 4 ?  ================================== Ai Message ==================================  Tool Calls:  add (call\_gbhSI4Awo4diuFCZQnx9eoGT)  Call ID: call\_gbhSI4Awo4diuFCZQnx9eoGT  Args:  op1: 15  op2: 5  subtract (call\_wPiw3jamp2dz3Sy5qo0buKdE)  Call ID: call\_wPiw3jamp2dz3Sy5qo0buKdE  Args:  op1: 10  op2: 2  ================================= Tool Message =================================  Name: subtract  8  ================================== Ai Message ==================================  Tool Calls:  multiply (call\_eEKJxO2PgSSGihaD34TMXFU5)  Call ID: call\_eEKJxO2PgSSGihaD34TMXFU5  Args:  op1: 20  op2: 8  ================================= Tool Message =================================  Name: multiply  160  ================================== Ai Message ==================================  Tool Calls:  divide (call\_8LEeuX2kqFDnLtChWIPS1H0c)  Call ID: call\_8LEeuX2kqFDnLtChWIPS1H0c  Args:  op1: 160  op2: 4  ================================= Tool Message =================================  Name: divide  40.0  ================================== Ai Message ==================================  The result of the expression \((15 + 5) \* (10 - 2) / 4\) is \(40.0\).  Final Response:  The result of the expression \((15 + 5) \* (10 - 2) / 4\) is \(40.0\). |

#### HEALTH ASSISTANT

|  |
| --- |
| **import json**  **import os**  **from langchain\_openai import AzureChatOpenAI**  **from langchain\_openai import AzureOpenAIEmbeddings**  **from langchain.vectorstores import Chroma**  **from langchain\_core.tools import tool**  **from typing import Annotated**  **from typing\_extensions import TypedDict**  **from langgraph.graph.message import add\_messages**  **from langchain\_core.messages import SystemMessage, HumanMessage**  **from langgraph.graph import StateGraph, START, END**  **from langgraph.prebuilt import ToolNode, tools\_condition**  **from IPython.display import display, Image, Markdown**  **from dotenv import load\_dotenv**  **from doctors import doctors\_db**  **from IPython.display import display, Markdown**  **load\_dotenv()**  **# Set your Azure OpenAI credentials**  **subscription\_key = os.environ["AZURE\_OPENAI\_API\_KEY"]**  **endpoint = os.environ["AZURE\_OPENAI\_ENDPOINT"]**  **api\_version = os.environ["AZURE\_OPENAI\_API\_VERSION"]**  **deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT"]**  **embedding\_deployment = os.environ["AZURE\_OPENAI\_DEPLOYMENT\_FOR\_EMBEDDING"]**  **# Initialize the Azure OpenAI LLM**  **llm = AzureChatOpenAI(**  **azure\_deployment=deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **temperature=0,**  **)**  **# Initialize the Azure OpenAI Embeddings**  **embeddings\_client = AzureOpenAIEmbeddings(**  **azure\_deployment=embedding\_deployment,**  **api\_version=api\_version,**  **azure\_endpoint=endpoint,**  **api\_key=subscription\_key,**  **)**  **# Read the JSON File**  **def load\_json\_data\_source():**  **try:**  **current\_dir = os.path.dirname(os.path.abspath(\_\_file\_\_))**  **json\_file\_path = os.path.join(current\_dir, "search.json")**  **if not os.path.exists(json\_file\_path):**  **print(f"Error: search.json not found at {json\_file\_path}")**  **return None**  **with open(json\_file\_path, "r") as f:**  **search\_docs = json.load(f)**  **return search\_docs**  **except json.JSONDecodeError as e:**  **print(f"Error decoding JSON: {e}")**  **return None**  **except Exception as e:**  **print(f"Unexpected error: {e}")**  **return None**  **# Load the JSON data**  **search\_docs = load\_json\_data\_source()**  **if search\_docs is None:**  **print("Failed to load search data")**  **else:**  **print(f"Major Document Types: {list(search\_docs.keys())}")**  **# Store the publications and web Articles in in vector Db in different namespaces**  **web\_search\_db = Chroma.from\_texts(**  **search\_docs["search\_pages"],**  **collection\_name="web\_search\_db\_demo1",**  **embedding=embeddings\_client,**  **)**  **pubmed\_db = Chroma.from\_texts(**  **search\_docs["pubmed\_pages"],**  **collection\_name="pubmed\_db\_demo1",**  **embedding=embeddings\_client,**  **)**  **### Tools Definations**  **@tool**  **def search\_web(query: str) -> list:**  **"""Search the web for a query. Useful for retrieving general or up-to-date healthcare information."""**  **# Perform semantic similarity search over the web search vector database**  **results = web\_search\_db.similarity\_search(query)**  **docs = [doc.page\_content for doc in results]**  **return docs**  **@tool**  **def search\_pubmed(query: str) -> list:**  **"""Search PubMed for scientific articles related to the query."""**  **# Perform semantic similarity search over the PubMed vector database**  **results = pubmed\_db.similarity\_search(query)**  **docs = [doc.page\_content for doc in results]**  **return docs**  **# Tool for recommending a doctor based on user symptoms or health-related queries**  **@tool**  **def recommend\_doctor(query: str) -> dict:**  **"""Recommend the most suitable doctor based on the user's symptoms."""**  **doctors\_list = str(doctors\_db)**  **# Use the LLM to reason over the list and identify the best match for the user's concern**  **prompt = f"""You are an assistant helping recommend a doctor based on a patient's health issues.**  **Here is the list of available doctors:**  **{doctors\_list}**  **Given the user's query: "{query}"**  **Choose the most suitable doctor from the list. Only pick one doctor.**  **Return only the selected doctor's information in JSON format.**  **If unsure, recommend the General Physician.**  **"""**  **response = llm.invoke(prompt)**  **return {"recommended\_doctor": response.content}**  **tools = [search\_web, search\_pubmed, recommend\_doctor]**  **# Instruction prompt for the overall Agent**  **AGENT\_PROMPT\_TXT = """You are an agent designed to act as an expert in researching medical symptoms**  **and recommending relevant doctors for booking appointments. Also, remember the current year is 2025.**  **Given a user query, call the relevant tools and provide the most appropriate response.**  **Follow these guidelines to help you make more informed decisions:**  **- If the user's query specifically asks for a doctor recommendation, recommend an appropriate doctor.**  **- If the user is researching specific aspects related to symptoms, treatments, or other areas of healthcare,**  **use both the search\_web and search\_pubmed tools to gather detailed information and provide a well-structured response.**  **- If the user is just looking for general healthcare information, web search alone is sufficient.**  **- Use the search\_pubmed tool only if the query relates to information typically found in PubMed articles.**  **- Responses should include cited source links and/or PubMed article titles and publication dates, if available.**  **- If recommending doctors, use the recommend\_doctor tool and display detailed information in a clear, structured format.**  **Also, encourage the user to book an appointment via email.**  **- Politely decline to answer any queries that are not related to medical or healthcare information.**  **The final response should contain the following:**  **- The main output content.**  **- At the end, include an Agent Reasoning: section that covers the following in bullet points:**  **- Your step-by-step reasoning process for arriving at the final response.**  **- The tools you called, in the specific order, with their names.**  **- Observations from the tool call results that helped you construct the final response.**  **"""**  **AGENT\_SYS\_PROMPT = SystemMessage(content=AGENT\_PROMPT\_TXT)**  **# List of all tools that the LLM should be aware of**  **# These tools were defined earlier using the @tool decorator**  **tools = [search\_web, search\_pubmed, recommend\_doctor]**  **# Bind the tools to the LLM so it can invoke them when necessary**  **# Enables tool-calling functionality in LangChain**  **llm\_with\_tools = llm.bind\_tools(tools=tools)**  **# Utility function to call the agent and stream its step-by-step reasoning**  **def call\_agent(agent, query, verbose=False):**  **# Stream the agent's execution for the given query**  **for event in agent.stream(**  **{"messages": [HumanMessage(content=query)]},  # Input prompt**  **stream\_mode="values",  # Stream output as intermediate values**  **):**  **# If verbose is enabled, print each intermediate message**  **if verbose:**  **event["messages"][-1].pretty\_print()**  **# Display the final response from the agent as Markdown**  **print("\n\nFinal Response:\n")**  **print(event["messages"][-1].content)**  **# Return the final message content for optional downstream use**  **return event["messages"][-1].content**  **# Define the agent's state schema for storing the message history**  **class State(TypedDict):**  **messages: Annotated[list, add\_messages]**  **# Create the node function that handles reasoning and planning using the LLM**  **def tool\_calling\_llm(state: State) -> State:**  **# Extract the current conversation history from the state**  **current\_state = state["messages"]**  **# Prepend the system instructions to the current message history**  **state\_with\_instructions = [AGENT\_SYS\_PROMPT] + current\_state**  **# Call the LLM to generate a new message (either a response or a tool call request)**  **response = [llm\_with\_tools.invoke(state\_with\_instructions)]**  **# Return the updated state containing the new message**  **return {"messages": response}**  **# Build the graph**  **builder = StateGraph(State)**  **# Add nodes**  **builder.add\_node("tool\_calling\_llm", tool\_calling\_llm)**  **builder.add\_node("tools", ToolNode(tools=tools))**  **# Add edges**  **builder.add\_edge(START, "tool\_calling\_llm")**  **# Conditional edge**  **builder.add\_conditional\_edges(**  **"tool\_calling\_llm",**  **tools\_condition,  # conditional routing function**  **# If the latest message (result) from LLM is a tool call request -> tools\_condition routes to tools**  **# If the latest message (result) from LLM is a not a tool call -> tools\_condition routes to END**  **["tools", END],**  **)**  **builder.add\_edge(**  **"tools", "tool\_calling\_llm"**  **)  # this is the key feedback loop in the agentic system**  **# Compile Agent Graph**  **healthbuddy\_agent = builder.compile()**  **# Example usage**  **query = (**  **"what are the latest methods for diabetes management and recommend a doctor please"**  **)**  **result = call\_agent(healthbuddy\_agent, query, verbose=True)** |
| OUTPUT WILL BE SAME AS ABOVE |

|  |
| --- |
| **from langgraph.graph import StateGraph, END**  **from typing import TypedDict, List**  **from langchain\_openai import ChatOpenAI**  **llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)**  **class AgentState(TypedDict):**  **question: str**  **steps: List[str]**  **final\_answer: str**  **# Nodes**  **def reason\_node(state: AgentState):**  **q = state["question"]**  **if "Einstein" in q:**  **return {"steps": ["Wikipedia: Einstein"]}**  **if "5\*12" in q:**  **return {"steps": ["Calculator: 5\*12"]}**  **return {"final\_answer": "I don't know"}**  **def wikipedia\_node(state: AgentState):**  **return {"steps": [search\_wikipedia("Albert Einstein")]}**  **def calculator\_node(state: AgentState):**  **return {"steps": [calculator("5\*12")]}**  **# Build graph**  **graph = StateGraph(AgentState)**  **graph.add\_node("reason", reason\_node)**  **graph.add\_node("wiki", wikipedia\_node)**  **graph.add\_node("calc", calculator\_node)**  **graph.add\_edge("reason", "wiki")**  **graph.add\_edge("wiki", "calc")**  **graph.add\_edge("calc", END)**  **graph.set\_entry\_point("reason")**  **compiled = graph.compile(reducers={"steps": lambda a, b: (a or []) + b})**  **print(compiled.invoke({"question": "Who is Albert Einstein and what is 5\*12?"}))** |