Tools

MUKESH KUMAR

- Tools
- Tools Calling

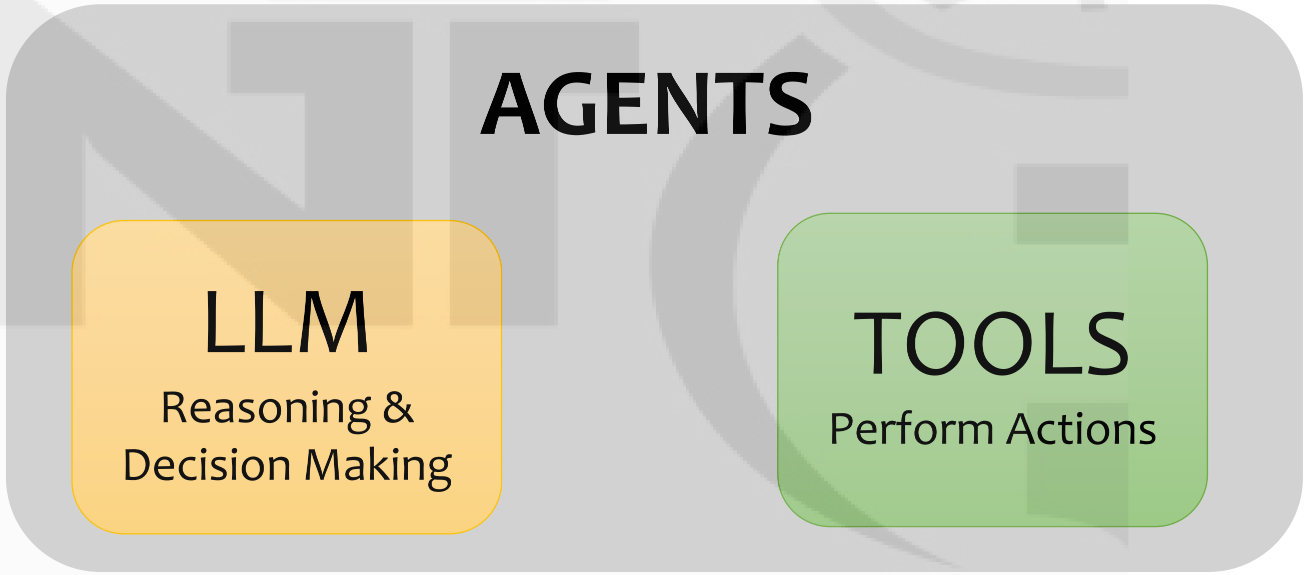
LLMs

- LLM can think and respond

- LLM cannot :
- Execute any tasks like :
- Fetch live data
- Update database

Tools

- Tools are functions that an agent can use
- Think of tools as capabilities: search, calculator, API calls
- Used by Agents to interact with the external world


AGENTS
LLM
Reasoning & Decision Making
TOOLS
Perform Actions

Tools types

- Builtin tools
- Custom tools

Built-in Tools:

- Ready-to-use implementations for common functionalities.
- Examples: Tavily Search (web search), Python REPL (code execution), Wikipedia (information retrieval), YouTube Search.

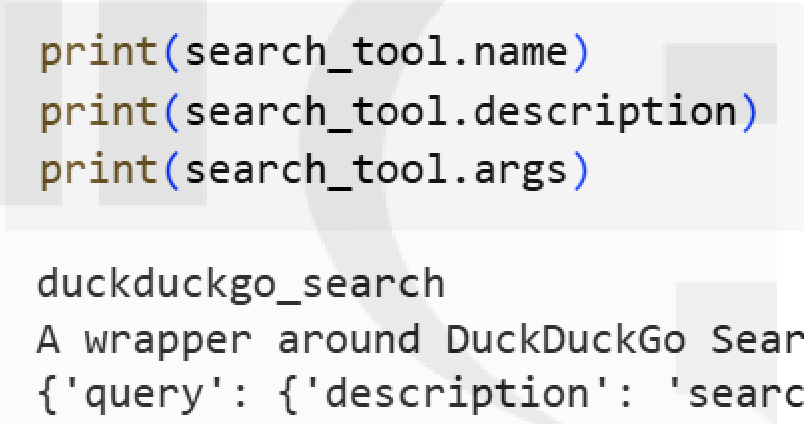
- **Toolkits:**
- Collections of tools designed to be used together for specific tasks.
- Examples: Toolkits for SQL databases, Gmail, GitHub, PDF processing, etc.

- Custom Tools:
- Developers can build their own tools tailored to specific needs using Python functions and decorators (@tool).
- This provides maximum flexibility for integrating any external system or API.

Working with built in tools

- https://python.langchain.com/docs/integrations/tools/

Tool Attributes


```
print(search_tool.name)
print(search_tool.description)
print(search_tool.args)
```

```
duckduckgo_search
A wrapper around DuckDuckGo Sear
{'query': {'description': 'searc
```

Working with custom tools

- Create a function for your tool (doc string is good practice- not mandatory)
- Add type hints (recommended step but not mandatory)
- Add tool decorator (makes it special function so LLM can use it)
- To use the tool call invoke and pass param as dictonary

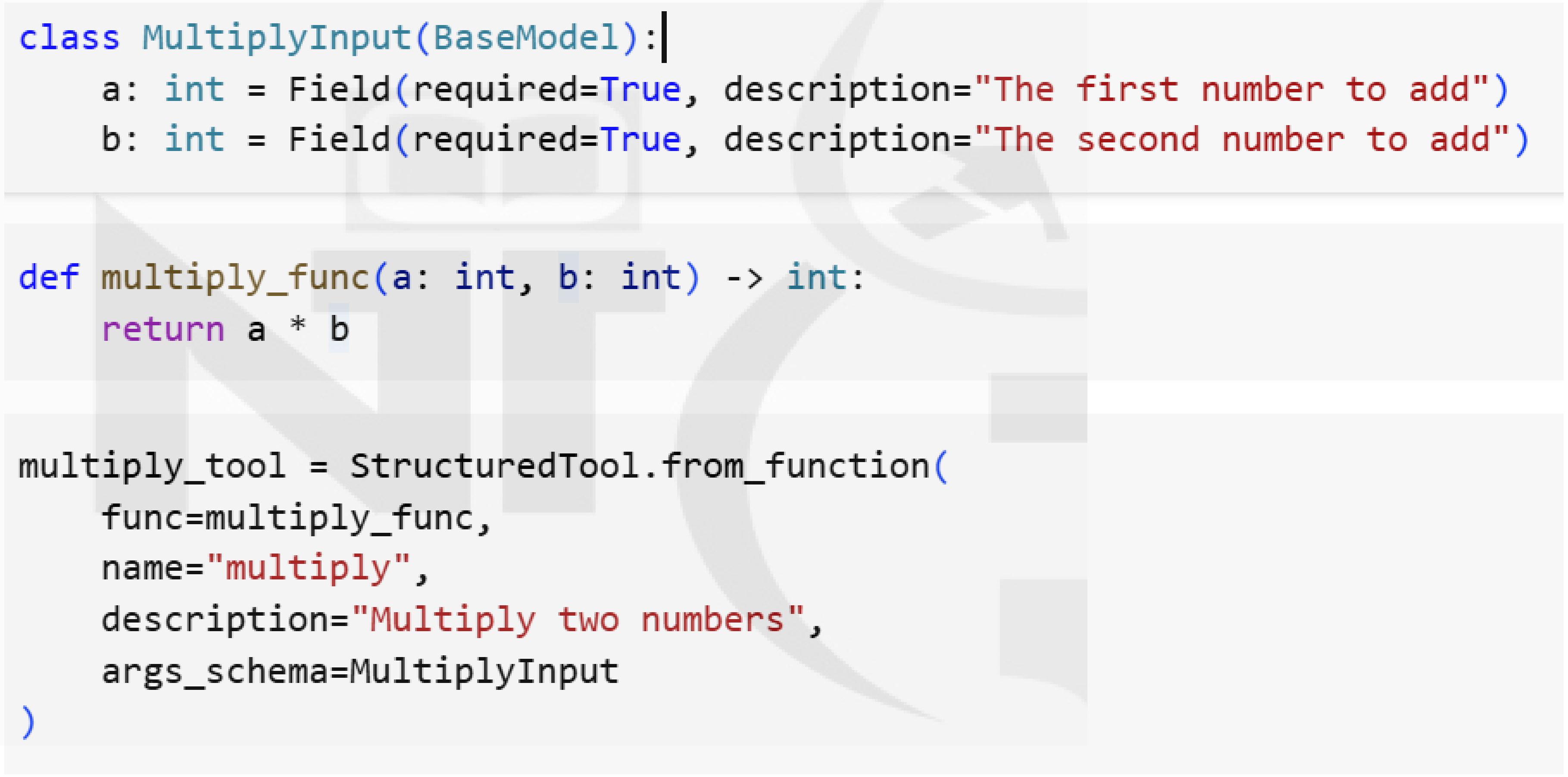
Ways to create tools

- Using @tool decorator
- Using StructuredTool and pydantic
- Using BaseTool class

```python
@tool
def multiply(a: int, b:int) -> int:
    """Multiply two numbers"""
    return a*b
```

Using Structured Tool

- Create method
- Define pydantic class
- Create the tool using StructuredTool.from_function



```python
class MultiplyInput(BaseModel):
    a: int = Field(required=True, description="The first number to add")
    b: int = Field(required=True, description="The second number to add")

def multiply_func(a: int, b: int) -> int:
    return a * b

multiply_tool = StructuredTool.from_function(
    func=multiply_func,
    name="multiply",
    description="Multiply two numbers",
    args_schema=MultiplyInput
)
```

Using BaseTool

```python
from langchain.tools import BaseTool
from typing import Type

# arg schema using pydantic

class MultiplyInput(BaseModel):
    a: int = Field(required=True, description="The first number to add")
    b: int = Field(required=True, description="The second number to add")

class MultiplyTool(BaseTool):
    name: str = "multiply"
    description: str = "Multiply two numbers"

    args_schema: Type[BaseModel] = MultiplyInput

    def _run(self, a: int, b: int) -> int:
        return a * b
```

Working with Toolkit

- Define related tools using @tool
- Create a class for toolkit
- Define list of tools inside toolkit class

```
from langchain_core.tools import tool

# Custom tools
@tool
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

@tool
def multiply(a: int, b: int) -> int:
    """Multiply two numbers"""
    return a * b
```

```
class MathToolkit:
    def get_tools(self):
        return [add, multiply]
```

AI Agent

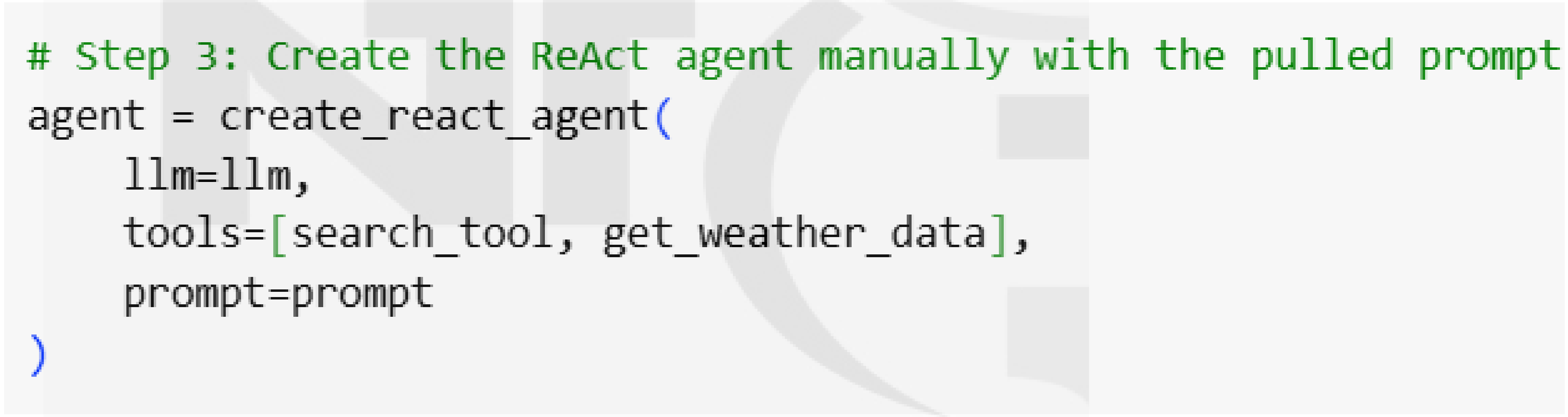
- An AI Agent is an autonomous system that can think, reason, and act to achieve goals.
- LangChain agents use language models to decide which tools to use and when.
- Agents help automate complex workflows involving tool usage and decision making.

Types of LangChain Agents

- ZeroShotAgent: Uses a prompt template with tool descriptions.
- ReAct Agent: Interleaves reasoning and acting steps.
- Plan-and-Execute: First plans steps, then executes each.
- Custom agents can be defined using `AgentExecutor`.

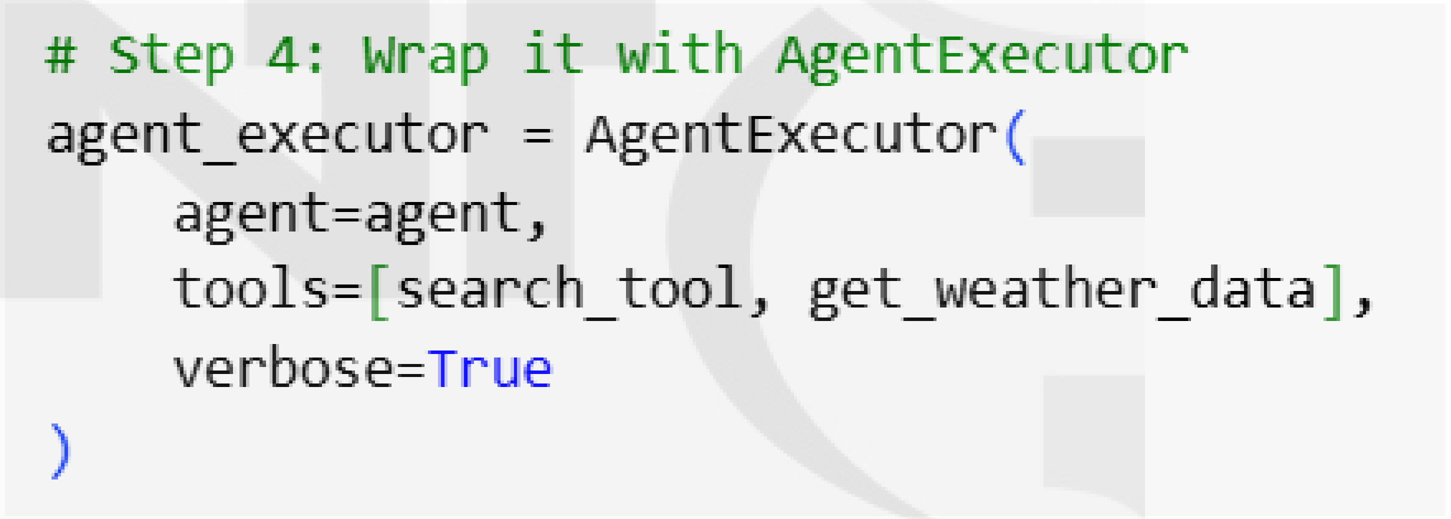
Create_react_Agent

- Define the agent



```python
# Step 3: Create the ReAct agent manually with the pulled prompt
agent = create_react_agent(
    llm=llm,
    tools=[search_tool, get_weather_data],
    prompt=prompt
)
```

Agent Executor



```python
# Step 4: Wrap it with AgentExecutor
agent_executor = AgentExecutor(
    agent=agent,
    tools=[search_tool, get_weather_data],
    verbose=True
)
```

ReAct Pattern (Reasoning + Acting)

- Thought: Internal reasoning step.
- Action: Decide on and execute a tool.
- Observation: Capture tool result.
- Loop continues until Final Answer is reached.

What is ReAct

ReAct Framework

- The ReAct framework, implemented within LangChain, enables LLMs to reason and act based on a given situation. It mimics human reactions to problems using external tools, thus enhancing the model's ability to respond to queries.

How ReAct Agents Work

- **Reasoning:** The agent analyzes the input and determines the necessary steps to take.
- **Action:** Based on the reasoning, the agent interacts with external tools or data sources.
- **Observation:** The agent observes the results of its actions and uses them to refine its reasoning.

- ReAct Agent continues this loop : **Thought >Action>Observation** until it finds the final answer.

Agent & Agent Executor

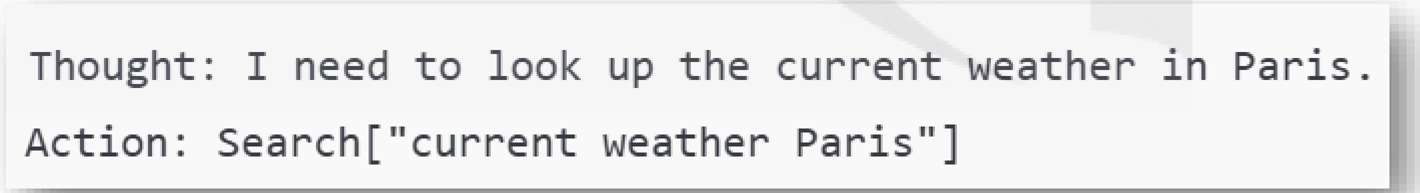
Agent

Role: "Think"

- The **Agent** is the **brain**. It decides **what to do next** based on the input, current context, and intermediate results.
- **Responsibilities:**
- Parse user input or task.
- Generate reasoning steps (e.g., using ReAct: Thought → Action).
- Choose the **next action/tool** to call.
- Determine when to **stop** and return a final answer.

Example

- Given a question like:
- “What is the weather in Paris?”
- The Agent may produce:



```
Thought: I need to look up the current weather in Paris.
Action: Search["current weather Paris"]
```

AgentExecutor

Role: "Do"

- The **AgentExecutor** is the **engine**. It **runs the loop**, invoking the agent repeatedly, managing the tools, and feeding back observations.
- **Responsibilities:**
- Execute the tool/action chosen by the Agent.
- Capture and store the result (**observation**).
- Feed it back into the Agent.
- Repeat the loop (Thought → Action → Observation) until done.
- Optionally, manage memory and context across steps.

Agent & Agent Executor Summary

Component	Role	Analogy
Agent	Thinker / Planner	Strategist or Pilot
AgentExecutor	Doer / Coordinator	Ground Crew / Engine

Use Cases of LangChain Agents

- Customer support automation
- Multi-step reasoning tasks
- RPA (Robotic Process Automation)
- Research assistants
- Data enrichment and scraping

Summary

- LangChain agents combine reasoning and tool use.
- They use LLMs, tools, and memory to perform tasks.
- ReAct is a powerful pattern for decision making.
- Flexible, composable, and useful in production AI systems.