**Long chain building block - Agents**

**6) Agents**

* **Tools**: search, calculator, code, HTTP, SQL, custom tools.
* **Policies/Types**: tool-calling (functions), ReAct, plan-and-execute.
* **Runtime**: AgentExecutor (+ create\_tool\_calling\_agent / ReAct builders).

Perfect! Let’s deep dive into **LangChain Agents**—this is the part that really makes LLMs “do” things instead of just talking. I’ll break it down step by step, with examples and practical insights.

**1️⃣ What is an Agent?**

An **Agent** is an LLM-powered system that can **decide which actions to take and when**, using a set of tools. It’s more autonomous than a simple chain because it can:

* Pick the right tool dynamically.
* Plan multiple steps.
* Reason based on intermediate results.

Think of it as **GPT + toolbox + reasoning policy**.

**2️⃣ Tools**

Tools are the “abilities” your agent can call.

**Common tools:**

| **Tool Type** | **Description** |
| --- | --- |
| **Search** | Query Google, Bing, or custom search APIs. |
| **Calculator** | Do arithmetic or math operations. |
| **Code** | Execute Python/JS or other scripts dynamically. |
| **HTTP** | Call REST APIs to get live data. |
| **SQL** | Query databases. |
| **Custom Tools** | Anything you define, like WeatherFetcher, EmailSender. |

**Example: Custom Python Tool**

python

CopyEdit

from langchain.tools import Tool

def greet\_user(name: str) -> str:

return f"Hello {name}, welcome!"

greet\_tool = Tool(

name="GreetUser",

func=greet\_user,

description="Greets a user by their name."

)

**3️⃣ Agent Policies / Types**

Agents differ in **how they decide which tool to call**.

**A. Tool-Calling (Function Agents)**

* Predefined mapping of tools → functions.
* LLM just decides **which function to call and with what args**.
* Often used with **OpenAI function calling**.

python

CopyEdit

from langchain.agents import create\_tool\_calling\_agent

agent = create\_tool\_calling\_agent(

llm=llm,

tools=[greet\_tool]

)

**B. ReAct Agents**

* Short for **Reason + Act**.
* Agent alternates between **thinking** and **acting**.
* Useful for **multi-step reasoning**.
* Output includes:
  + **Thought**: reasoning about what to do.
  + **Action**: tool to call.
  + **Observation**: result from the tool.

text

CopyEdit

Thought: I need to check the current temperature.

Action: WeatherFetcher(city="New York")

Observation: 27°C

Thought: I can now recommend clothing based on weather.

**C. Plan-and-Execute**

* Agent creates a **plan first** (like a roadmap of steps).
* Then executes steps **one by one**, possibly with intermediate reasoning.
* Good for **complex workflows**.

**4️⃣ Runtime: AgentExecutor**

AgentExecutor is the runtime that **runs an agent with tools**, handles reasoning loops, and manages tool calls.

python

CopyEdit

from langchain.agents import AgentExecutor

executor = AgentExecutor(agent=agent, tools=[greet\_tool])

response = executor.run("Greet Rajeev")

print(response) # Hello Rajeev, welcome!

**Key points:**

* Handles multiple steps automatically.
* Streams intermediate reasoning if needed.
* Can integrate ReAct policies or function-based agents.

**5️⃣ Full Example: Agent with Multiple Tools**

python

CopyEdit

from langchain.llms import OpenAI

from langchain.agents import initialize\_agent, Tool

# LLM

llm = OpenAI(temperature=0)

# Tools

def add(a: int, b: int) -> int:

return a + b

calculator\_tool = Tool(

name="Calculator",

func=add,

description="Adds two numbers"

)

def greet(name: str) -> str:

return f"Hello {name}!"

greet\_tool = Tool(

name="GreetUser",

func=greet,

description="Greets a user"

)

# Initialize ReAct agent

agent = initialize\_agent(

tools=[calculator\_tool, greet\_tool],

llm=llm,

agent="react",

verbose=True

)

# Run agent

agent.run("Greet Rajeev and then add 23 + 45")

**Output:**  
The agent will reason step by step:

1. Greet Rajeev
2. Compute 23 + 45
3. Return final combined answer

**✅ Key Takeaways**

1. **Tools are the agent’s abilities**—you define them.
2. **Agent policies** decide **how to think and act**:
   * Tool-calling → simplest
   * ReAct → dynamic reasoning
   * Plan-and-execute → structured workflows
3. **AgentExecutor** runs everything, handles loops, tool calls, and reasoning.
4. Agents allow **multi-step problem solving** using LLMs.

If you want, I can **build a full stack React + FastAPI UI** showing an **interactive LangChain Agent** that calls multiple tools like search, calculator, and API, with streaming reasoning. It’ll be a runnable GenAI demo!

Do you want me to do that next?

**You said:**

lets deep dive into 2. Tools

**ChatGPT said:**

Perfect! Let’s deep dive into **LangChain Tools**—this is where the agent’s real “abilities” live. Understanding tools well is crucial because **agents are only as smart as the tools they can use**.

**1️⃣ What is a Tool?**

In LangChain, a **Tool** is essentially a callable function with some metadata:

* **name** → Unique name of the tool.
* **func** → The Python function (or async function) that does the actual work.
* **description** → Explains what the tool does; the agent reads this to decide when to use it.

Tools can wrap:

* **Simple functions** (add numbers, greet users)
* **APIs** (search engines, weather APIs)
* **Code execution** (Python code, SQL queries)
* **Custom actions** (sending emails, querying a database)

**2️⃣ Built-in Tool Types**

**A. Search Tools**

Used to query online or internal knowledge sources.

python

CopyEdit

from langchain.tools import SerpAPIWrapper, Tool

from langchain.agents import initialize\_agent

from langchain.llms import OpenAI

search = SerpAPIWrapper()

search\_tool = Tool(

name="Search",

func=search.run,

description="Use this tool to search the web for factual information."

)

* Can use Google, Bing, Wikipedia, or custom knowledge bases.
* The agent reads the description to know **when to use this tool**.

**B. Calculator Tools**

For **math operations**, arithmetic, or complex calculations.

python

CopyEdit

def calculator(a: int, b: int) -> int:

return a + b

calculator\_tool = Tool(

name="Calculator",

func=calculator,

description="Adds two numbers."

)

* Can wrap Python functions or external calculator APIs.
* Useful in ReAct for step-by-step reasoning.

**C. Code Tools**

Allows the agent to **execute code dynamically**.

python

CopyEdit

from langchain.tools.python.tool import PythonREPLTool

python\_tool = PythonREPLTool()

* Executes Python code safely in a sandbox.
* Perfect for complex calculations, string manipulations, or data transformations.

**D. HTTP Tools**

Calls APIs or HTTP endpoints.

python

CopyEdit

import requests

from langchain.tools import Tool

def get\_joke() -> str:

res = requests.get("https://official-joke-api.appspot.com/jokes/random")

return res.json()["setup"] + " " + res.json()["punchline"]

http\_tool = Tool(

name="GetJoke",

func=get\_joke,

description="Fetches a random joke from the internet."

)

* Can call REST APIs, GraphQL endpoints, or custom internal services.
* Agent can dynamically fetch live data.

**E. SQL / Database Tools**

For **querying structured data**.

python

CopyEdit

from langchain.sql\_database import SQLDatabase

from langchain.tools import tool

db = SQLDatabase.from\_uri("sqlite:///mydata.db")

sql\_tool = tool(

name="SQLDatabaseTool",

func=db.run,

description="Run SQL queries against the database."

)

* Can be **read-only** (SELECT queries) or allow updates.
* Often paired with **ReAct agents** for multi-step database queries.

**F. Custom Tools**

Anything you define—your own “superpowers” for the agent.

python

CopyEdit

def greet\_user(name: str) -> str:

return f"Hello {name}, welcome to the platform!"

greet\_tool = Tool(

name="GreetUser",

func=greet\_user,

description="Greets a user by their name."

)

* Agent uses the description to know **when it’s appropriate** to call the tool.
* You can make tools **complex**, e.g., integrate AI models, sensors, or internal services.

**3️⃣ Tool Metadata and Importance**

**Why description is critical:**

* Agents **read tool descriptions** to decide which tool to use.
* In ReAct, the LLM thinks:

“I need to perform X, which tool can help? Ah, the Search tool says it fetches factual information.”

**Other optional metadata:**

* return\_direct → If True, output goes directly back to the user.
* args\_schema → If using function-calling agents, specifies the argument types.

**4️⃣ Tool Best Practices**

1. **Keep tools small & focused** → One tool = one responsibility.
2. **Write clear descriptions** → The agent relies on them heavily.
3. **Combine tools wisely** → Too many tools can confuse the agent.
4. **Use async if needed** → For HTTP or long-running operations.
5. **Security first** → Be careful exposing tools that can modify your system.

**5️⃣ Example: Multi-Tool Agent**

python

CopyEdit

from langchain.agents import initialize\_agent

from langchain.llms import OpenAI

llm = OpenAI(temperature=0)

tools = [calculator\_tool, greet\_tool, http\_tool] # multiple tools

agent = initialize\_agent(

tools=tools,

llm=llm,

agent="react",

verbose=True

)

response = agent.run("Greet Rajeev and then fetch a random joke")

print(response)

* The agent chooses tools **step by step** based on what’s needed.
* Shows **tool orchestration** in action.

✅ **Key Takeaways**

1. **Tools are the core capabilities of an agent.**
2. LangChain has **built-in tools** (Search, Calculator, Python, HTTP, SQL).
3. You can **define custom tools** for any functionality.
4. **Descriptions and metadata** guide agent reasoning.
5. Multi-tool orchestration enables **complex workflows and reasoning**.

If you want, next I can do a **deep dive into ReAct agent** showing **how it chooses tools step by step**, with **streaming reasoning** so you see the agent “think.”

Do you want me to do that next?

**You said:**

yes please - I can do a deep dive into ReAct agent showing how it chooses tools step by step, with streaming reasoning so you see the agent “think.”

**ChatGPT said:**

Perfect! Let’s do a **deep dive into ReAct agents** in LangChain, showing **step-by-step tool selection** with **streaming reasoning**—so you can actually see the agent “think.”

**1️⃣ What is a ReAct Agent?**

**ReAct = Reason + Act**

It’s a **policy for an agent** where it alternates between:

1. **Thought** → LLM reasons about what needs to be done.
2. **Action** → Chooses a tool to call and specifies arguments.
3. **Observation** → Receives the result from the tool.
4. **Repeat** → Loops until the task is complete.

Think of it as **“thinking out loud” while solving a problem**.

**2️⃣ Key Components**

| **Component** | **Role** |
| --- | --- |
| **LLM** | Generates reasoning and decides actions. |
| **Tools** | Capabilities the agent can call (calculator, search, API, custom). |
| **AgentExecutor** | Runtime that runs the ReAct loop and manages tool calls. |
| **Verbose mode** | Shows reasoning steps (Thought → Action → Observation). |

**3️⃣ Step-by-Step ReAct Flow**

Suppose we have **3 tools**:

1. GreetUser(name) → Greets the user.
2. Calculator(a, b) → Adds numbers.
3. GetJoke() → Fetches a random joke.

**Task:**

“Greet Rajeev, add 23 + 45, and fetch a joke.”

**Step 1: LLM thinks**

vbnet

CopyEdit

Thought: I need to greet Rajeev first to make the response friendly.

Action: GreetUser(name="Rajeev")

**Step 2: Agent executes the tool**

makefile

CopyEdit

Observation: Hello Rajeev, welcome!

**Step 3: LLM continues reasoning**

vbnet

CopyEdit

Thought: Now I need to calculate 23 + 45.

Action: Calculator(a=23, b=45)

**Step 4: Agent executes the tool**

makefile

CopyEdit

Observation: 68

**Step 5: LLM continues reasoning**

kotlin

CopyEdit

Thought: Finally, to make the response fun, I can fetch a joke.

Action: GetJoke()

**Step 6: Agent executes the tool**

vbnet

CopyEdit

Observation: Why did the chicken cross the road? To get to the other side!

**Step 7: LLM aggregates results and returns**

vbnet

CopyEdit

Response: Hello Rajeev, welcome! The sum of 23 + 45 is 68. Here's a joke for you: Why did the chicken cross the road? To get to the other side!

**4️⃣ ReAct Agent Code Example**

python

CopyEdit

from langchain.llms import OpenAI

from langchain.agents import initialize\_agent, Tool

# 1️⃣ Define LLM

llm = OpenAI(temperature=0)

# 2️⃣ Define tools

def greet\_user(name: str) -> str:

return f"Hello {name}, welcome!"

def add\_numbers(a: int, b: int) -> int:

return a + b

def get\_joke() -> str:

import requests

res = requests.get("https://official-joke-api.appspot.com/jokes/random")

joke = res.json()

return f"{joke['setup']} {joke['punchline']}"

greet\_tool = Tool(name="GreetUser", func=greet\_user, description="Greets a user by name")

calculator\_tool = Tool(name="Calculator", func=add\_numbers, description="Adds two numbers")

joke\_tool = Tool(name="GetJoke", func=get\_joke, description="Fetches a random joke")

tools = [greet\_tool, calculator\_tool, joke\_tool]

# 3️⃣ Initialize ReAct agent with verbose streaming

agent = initialize\_agent(

tools=tools,

llm=llm,

agent="react",

verbose=True

)

# 4️⃣ Run agent

response = agent.run("Greet Rajeev, add 23 + 45, and fetch a joke")

print("\nFinal Response:\n", response)

**What happens here:**

* verbose=True → prints **Thought → Action → Observation** in real time.
* Agent **chooses tools dynamically** based on reasoning.
* Agent **aggregates results** and returns the final response.

**5️⃣ Tips for ReAct Agents**

1. **Clear tool descriptions** → LLM depends on them to choose correctly.
2. **Multiple steps** → ReAct shines in multi-step workflows.
3. **Debugging** → Use verbose=True to watch reasoning.
4. **Limit loops** → Prevent infinite thinking cycles using max\_iterations.
5. **Streaming reasoning** → Can stream thought/action/observation to a UI in real-time.

✅ **Key Takeaways**

* ReAct agents **think and act** iteratively.
* They can **use multiple tools step by step**.
* verbose=True reveals the agent's **“inner monologue”**.
* Perfect for **multi-step reasoning tasks**, dynamic workflows, or tool orchestration.

If you want, I can next show a **full stack React + FastAPI demo** where the **ReAct agent streams its reasoning to the UI**, so you can literally **watch the agent think and act in real-time**.

Do you want me to build that next?

**You said:**

yes please - I can next show a full stack React + FastAPI demo where the ReAct agent streams its reasoning to the UI, so you can literally watch the agent think and act in real-time.

**ChatGPT said:**

Perfect! Let’s build a **full-stack React + FastAPI demo** where a **ReAct agent streams its reasoning** to the UI in real-time. I’ll break it down step by step with a clear architecture and runnable code.

**1️⃣ Architecture Overview**

css

CopyEdit

[React Frontend] <--WebSocket--> [FastAPI Backend] <--LangChain ReAct Agent--> [Tools]

* **Frontend**:
  + React 18 / Vite or Create React App.
  + Connects via WebSocket to receive streaming Thought → Action → Observation steps.
  + Displays the agent’s “inner monologue” in real-time.
* **Backend**:
  + FastAPI with WebSocket endpoint.
  + Runs LangChain ReAct agent.
  + Streams each reasoning step to the frontend.
* **Tools**:
  + Calculator
  + GreetUser
  + GetJoke

**2️⃣ FastAPI Backend with Streaming ReAct Agent**

python

CopyEdit

# backend/main.py

import asyncio

import json

from fastapi import FastAPI, WebSocket

from fastapi.middleware.cors import CORSMiddleware

from langchain.llms import OpenAI

from langchain.agents import initialize\_agent, Tool

app = FastAPI()

# Allow frontend localhost

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_methods=["\*"],

allow\_headers=["\*"],

)

# Tools

def greet\_user(name: str) -> str:

return f"Hello {name}, welcome!"

def add\_numbers(a: int, b: int) -> int:

return a + b

def get\_joke() -> str:

import requests

res = requests.get("https://official-joke-api.appspot.com/jokes/random")

joke = res.json()

return f"{joke['setup']} {joke['punchline']}"

greet\_tool = Tool(name="GreetUser", func=greet\_user, description="Greets a user by name")

calculator\_tool = Tool(name="Calculator", func=add\_numbers, description="Adds two numbers")

joke\_tool = Tool(name="GetJoke", func=get\_joke, description="Fetches a random joke")

tools = [greet\_tool, calculator\_tool, joke\_tool]

# LLM

llm = OpenAI(temperature=0)

# ReAct agent

agent = initialize\_agent(

tools=tools,

llm=llm,

agent="react",

verbose=True # We'll capture the verbose output

)

@app.websocket("/ws/agent")

async def websocket\_agent(websocket: WebSocket):

await websocket.accept()

try:

while True:

data = await websocket.receive\_text()

# Run the agent asynchronously

loop = asyncio.get\_event\_loop()

def run\_agent():

# Capture verbose output by temporarily patching print

steps = []

import builtins

original\_print = builtins.print

def mock\_print(\*args, \*\*kwargs):

msg = " ".join(str(a) for a in args)

steps.append(msg)

builtins.print = mock\_print

try:

result = agent.run(data)

finally:

builtins.print = original\_print

return steps, result

steps, result = await loop.run\_in\_executor(None, run\_agent)

# Stream steps one by one

for step in steps:

await websocket.send\_text(json.dumps({"type": "step", "content": step}))

await asyncio.sleep(0.1) # simulate streaming

# Send final response

await websocket.send\_text(json.dumps({"type": "final", "content": result}))

except Exception as e:

await websocket.close()

**3️⃣ React Frontend (Streaming UI)**

jsx

CopyEdit

// frontend/src/App.jsx

import { useEffect, useState, useRef } from "react";

export default function App() {

const [messages, setMessages] = useState([]);

const [input, setInput] = useState("");

const wsRef = useRef(null);

useEffect(() => {

wsRef.current = new WebSocket("ws://localhost:8000/ws/agent");

wsRef.current.onmessage = (event) => {

const data = JSON.parse(event.data);

setMessages((prev) => [...prev, data]);

};

return () => wsRef.current.close();

}, []);

const handleSend = () => {

if (input && wsRef.current) {

wsRef.current.send(input);

setInput("");

}

};

return (

<div style={{ padding: "20px", fontFamily: "sans-serif" }}>

<h1>ReAct Agent Streaming Demo</h1>

<div

style={{

border: "1px solid #ccc",

padding: "10px",

height: "400px",

overflowY: "scroll",

background: "#f9f9f9",

}}

>

{messages.map((msg, idx) => (

<div key={idx}>

{msg.type === "step" && <p>💭 {msg.content}</p>}

{msg.type === "final" && <p>✅ {msg.content}</p>}

</div>

))}

</div>

<input

type="text"

value={input}

onChange={(e) => setInput(e.target.value)}

placeholder="Ask the agent..."

style={{ width: "70%", padding: "8px", marginRight: "10px" }}

/>

<button onClick={handleSend} style={{ padding: "8px 16px" }}>

Send

</button>

</div>

);

}

**4️⃣ How It Works**

1. User types a prompt in React and clicks **Send**.
2. WebSocket sends the prompt to FastAPI.
3. FastAPI runs **ReAct agent**, capturing verbose Thought → Action → Observation.
4. Each step is **streamed to frontend** in near real-time.
5. Final response is sent when the agent finishes.
6. React UI shows the agent “thinking” dynamically.

**5️⃣ Demo Example**

**Input:**

“Greet Rajeev, add 23 + 45, and fetch a joke”

**Streaming output in UI:**

vbnet

CopyEdit

💭 Thought: I need to greet Rajeev first.

💭 Action: GreetUser(name="Rajeev")

💭 Observation: Hello Rajeev, welcome!

💭 Thought: Now I will calculate 23 + 45

💭 Action: Calculator(a=23, b=45)

💭 Observation: 68

💭 Thought: Let's fetch a joke

💭 Action: GetJoke()

💭 Observation: Why did the chicken cross the road? To get to the other side!

✅ Final: Hello Rajeev, welcome! The sum of 23 + 45 is 68. Here's a joke: Why did the chicken cross the road? To get to the other side!

* You literally **watch the agent think, choose tools, and act** in real-time.

**6️⃣ Key Points / Tips**

1. **Streaming makes LLM reasoning transparent**.
2. **WebSocket is perfect** for live agent feedback.
3. **Verbose mode = essential** for ReAct agents.
4. **Tools can be swapped or added dynamically** without changing UI.
5. **Frontend can highlight thoughts, actions, and observations differently** for clarity.

If you want, I can next **upgrade this demo** to include:

* **Multiple ReAct agents** running concurrently
* **Async tool execution** for HTTP/DB
* **Interactive UI** with expandable reasoning trees

This would turn it into a **full production-ready GenAI agent dashboard**.

Do you want me to do that next?

**You said:**

lets deep dive int to - Agent Policies / Types

**ChatGPT said:**

Perfect! Let’s do a **deep dive into Agent Policies / Types** in LangChain. This is where the agent’s **behavior and decision-making strategy** are defined. Understanding this is key to building effective autonomous agents.

**1️⃣ What Are Agent Policies?**

**Agent Policies (or Agent Types)** determine **how an agent decides which tool to use, when, and in what order**.

Think of them as **strategies for reasoning + acting**. Different policies are suited for different types of tasks:

* Simple single-step tasks → Tool-calling agents.
* Multi-step reasoning → ReAct or Plan-and-Execute agents.
* Complex workflows with sub-tasks → Plan-and-Execute or hierarchical agents.

**2️⃣ Main Agent Policies / Types in LangChain**

| **Policy / Type** | **Behavior / Strategy** | **Use Case** |
| --- | --- | --- |
| **Tool-Calling (Function) Agent** | LLM chooses a tool to call by matching function descriptions. Single-step. | Simple queries, predefined function calls |
| **ReAct Agent** | Alternates between **Thought → Action → Observation**. Multi-step reasoning. | Complex reasoning, multi-tool orchestration |
| **Plan-and-Execute** | Creates a **plan of steps first**, then executes them sequentially, possibly looping. | Structured workflows, multi-step pipelines |
| **Self-Ask / Tree of Thought** | Agent breaks down a question into sub-questions, solves them recursively. | Research-style problem solving, reasoning chains |
| **Zero-Shot / Few-Shot Agents** | Uses prompt examples to guide tool selection without explicit reasoning steps. | Quick tasks, low reasoning requirement |

**3️⃣ Deep Dive: Each Type**

**A. Tool-Calling Agent**

* **Behavior:** Chooses which tool to use directly based on the LLM understanding of the tool’s description.
* **Characteristics:**
  + Simple & lightweight.
  + Works well with OpenAI function calling.
* **Example:**

python

CopyEdit

from langchain.agents import create\_tool\_calling\_agent

agent = create\_tool\_calling\_agent(

llm=llm,

tools=[greet\_tool, calculator\_tool]

)

agent.run("Greet Rajeev and add 3 + 4")

* The LLM **directly calls GreetUser** then Calculator.

**B. ReAct Agent**

* **Behavior:** Alternates **Thought → Action → Observation**.
* **Characteristics:**
  + Dynamic multi-step reasoning.
  + Streams intermediate steps if verbose=True.
  + Can handle unexpected or multi-step queries.
* **Example Thought/Action loop:**

vbnet

CopyEdit

Thought: I need to greet the user first.

Action: GreetUser(name="Rajeev")

Observation: Hello Rajeev!

Thought: Now calculate 3+4

Action: Calculator(a=3, b=4)

Observation: 7

* Perfect for **step-by-step decision making**.

**C. Plan-and-Execute Agent**

* **Behavior:** Breaks the task into a **plan first**, then executes each step sequentially.
* **Characteristics:**
  + Structured multi-step execution.
  + Can revisit steps if needed.
* **Example:**
  + Plan: ["Greet user", "Add numbers", "Fetch joke"]
  + Execute: Step-by-step, reporting results.
* Great for **complex workflows**, e.g., document analysis + SQL + API calls.

**D. Self-Ask / Tree of Thought**

* **Behavior:**
  + Decomposes questions into **sub-questions**.
  + Solves each recursively and aggregates answers.
* **Use Case:**
  + Large reasoning chains, fact-checking, multi-step QA.

**E. Zero-Shot / Few-Shot Agents**

* **Behavior:** Uses examples in the prompt to **guide tool usage**.
* **Characteristics:**
  + Lightweight, no multi-step reasoning needed.
  + Good for predictable workflows.

**4️⃣ When to Use Which Agent Policy**

| **Task Complexity** | **Recommended Policy** |
| --- | --- |
| Simple function call | Tool-Calling Agent |
| Multi-step reasoning | ReAct Agent |
| Structured workflows | Plan-and-Execute |
| Recursive question solving | Tree of Thought |
| Quick predictable task | Zero/Few-Shot Agent |

**5️⃣ AgentExecutor**

* Regardless of agent type, **AgentExecutor** runs the agent:
  + Handles **loops**, **tool calls**, and **state management**.
  + Supports **streaming output** for ReAct or verbose agents.

python

CopyEdit

from langchain.agents import AgentExecutor

executor = AgentExecutor(agent=agent, tools=[calculator\_tool, greet\_tool])

response = executor.run("Greet Rajeev and add 23 + 45")

* For **ReAct agents**, verbose=True streams the **Thought → Action → Observation**.

✅ **Key Takeaways**

1. Agent type = reasoning + tool-calling strategy.
2. **Tool-Calling** → simplest, direct.
3. **ReAct** → step-by-step reasoning, dynamic tool use.
4. **Plan-and-Execute** → pre-planned multi-step execution.
5. **Tree of Thought / Self-Ask** → recursive reasoning.
6. **Zero/Few-Shot** → lightweight, prompt-guided.

I can next create a **comparison table + code demo** showing **all agent types in action** on the **same task**, so you can literally see **how each type behaves differently**.

Do you want me to do that next?

**You said:**

lets deep dive into - Runtime: AgentExecutor

**ChatGPT said:**

Perfect! Let’s deep dive into **Runtime: AgentExecutor** in LangChain. This is the **engine that runs your agents**, manages tool calls, reasoning loops, and streaming outputs. Understanding it is critical for building production-ready agent pipelines.

**1️⃣ What is AgentExecutor?**

**AgentExecutor** is the **runtime environment** for LangChain agents. It:

* Executes an agent (any policy: Tool-Calling, ReAct, Plan-and-Execute).
* Handles **tool calls**, including argument passing.
* Manages **iterations** for multi-step reasoning.
* Streams **Thought → Action → Observation** if verbose.
* Maintains **state between steps**.

In short, it’s the **bridge between your agent’s logic and the actual tools**.

**2️⃣ Key Responsibilities**

| **Responsibility** | **Description** |
| --- | --- |
| **Agent orchestration** | Runs the agent according to its policy (ReAct, Tool-Calling, Plan-and-Execute). |
| **Tool execution** | Calls the appropriate tool function with arguments. |
| **Iteration control** | Handles loops until termination conditions (max steps, final answer). |
| **Streaming output** | Emits intermediate reasoning steps (Thoughts, Actions, Observations). |
| **Error handling** | Captures exceptions from tools or LLM outputs. |

**3️⃣ Core Parameters**

When creating an AgentExecutor:

| **Parameter** | **Description** |
| --- | --- |
| agent | The agent instance (ReAct, Tool-Calling, etc.) |
| tools | List of tools the agent can use |
| max\_iterations | Max reasoning steps before termination (default: 15) |
| return\_intermediate\_steps | If True, returns each Thought → Action → Observation |
| handle\_parsing\_errors | Manage LLM output parsing errors gracefully |
| verbose | If True, prints or streams reasoning steps |

**4️⃣ How AgentExecutor Works Step by Step**

Let’s take a **ReAct agent** as an example with 3 tools: GreetUser, Calculator, GetJoke.

**Input Prompt:** “Greet Rajeev, add 23 + 45, and fetch a joke”

**Step 1: Initialization**

* AgentExecutor receives prompt.
* Initializes agent state (conversation memory, tool mapping).

**Step 2: Agent thinks**

* LLM generates a **Thought**: “I should greet Rajeev first.”

**Step 3: Tool selection**

* AgentExecutor maps **Thought → Tool**: chooses GreetUser(name="Rajeev").

**Step 4: Tool execution**

* Calls the Python function for GreetUser.
* Captures the **Observation**: “Hello Rajeev, welcome!”

**Step 5: Feedback loop**

* Sends Observation back to agent.
* LLM generates next Thought: “Now calculate 23+45”

**Step 6: Iterates**

* Repeats **Tool selection → Execution → Observation** until final answer is generated.

**Step 7: Termination**

* AgentExecutor stops after either:
  + LLM returns a **final answer**, or
  + max\_iterations is reached.

**5️⃣ Code Example: AgentExecutor in Action**

python

CopyEdit

from langchain.agents import AgentExecutor, initialize\_agent, Tool

from langchain.llms import OpenAI

# LLM

llm = OpenAI(temperature=0)

# Tools

def greet\_user(name: str) -> str:

return f"Hello {name}!"

def add\_numbers(a: int, b: int) -> int:

return a + b

def get\_joke() -> str:

import requests

res = requests.get("https://official-joke-api.appspot.com/jokes/random")

joke = res.json()

return f"{joke['setup']} {joke['punchline']}"

greet\_tool = Tool(name="GreetUser", func=greet\_user, description="Greets a user by name")

calculator\_tool = Tool(name="Calculator", func=add\_numbers, description="Adds two numbers")

joke\_tool = Tool(name="GetJoke", func=get\_joke, description="Fetches a random joke")

tools = [greet\_tool, calculator\_tool, joke\_tool]

# ReAct Agent

agent = initialize\_agent(tools=tools, llm=llm, agent="react", verbose=True)

# AgentExecutor

executor = AgentExecutor(agent=agent, tools=tools, max\_iterations=10, verbose=True)

# Run

final\_result = executor.run("Greet Rajeev, add 23 + 45, and fetch a joke")

print("Final Response:", final\_result)

**Output (verbose)**:

yaml

CopyEdit

💭 Thought: I need to greet Rajeev first

💭 Action: GreetUser(name="Rajeev")

💭 Observation: Hello Rajeev, welcome!

💭 Thought: Now I will calculate 23 + 45

💭 Action: Calculator(a=23, b=45)

💭 Observation: 68

💭 Thought: Fetch a joke

💭 Action: GetJoke()

💭 Observation: Why did the chicken cross the road? To get to the other side!

✅ Final: Hello Rajeev, welcome! The sum of 23 + 45 is 68. Here's a joke: Why did the chicken cross the road? To get to the other side!

**6️⃣ Advanced Features of AgentExecutor**

1. **Streaming to frontend** → perfect for WebSocket real-time display.
2. **Intermediate steps return** → useful for debugging or multi-step reasoning visualization.
3. **Error handling** → can continue even if one tool fails.
4. **Dynamic tool injection** → swap or add tools at runtime without restarting agent.
5. **Max iterations** → prevents infinite loops.

✅ **Key Takeaways**

* **AgentExecutor = runtime + orchestrator** for LangChain agents.
* Manages **multi-step reasoning**, **tool selection**, and **loop execution**.
* Supports **streaming verbose outputs**, making it ideal for **interactive UIs**.
* Works with **all agent types** (Tool-Calling, ReAct, Plan-and-Execute).

Next, I can do a **full diagram + flow animation** showing **AgentExecutor in action with a ReAct agent**, mapping **Thought → Action → Observation → Final Answer**, so you visually see the runtime orchestration.

Do you want me to do that next?

**You said:**

yes please - I can do a full diagram + flow animation showing AgentExecutor in action with a ReAct agent, mapping Thought → Action → Observation → Final Answer, so you visually see the runtime orchestration. Do you want me to do that next?

**ChatGPT said:**

Perfect! Let’s create a **full diagram + flow animation** to visualize **AgentExecutor running a ReAct agent**, showing the full cycle from **Thought → Action → Observation → Final Answer**.

Here’s a step-by-step conceptual flow:

**1️⃣ Diagram Overview**

pgsql

CopyEdit

User Prompt

|

v

+---------------------+

| AgentExecutor |

|---------------------|

| 1. Receive Prompt |

| 2. Initialize State |

+---------------------+

|

v

+---------------------+

| LLM (Agent) |

|---------------------|

| Generates Thought |

+---------------------+

|

v

+---------------------+

| Tool Selector |

|---------------------|

| Maps Thought → Tool |

+---------------------+

|

v

+---------------------+

| Tool Execution |

|---------------------|

| Calls tool function |

| Returns Observation |

+---------------------+

|

v

+---------------------+

| Feedback to LLM |

|---------------------|

| Observation added to |

| agent memory |

+---------------------+

|

v

[Repeat loop until final answer]

|

v

+---------------------+

| Final Response |

|---------------------|

| Aggregates results |

| Returns to user |

+---------------------+

**2️⃣ Flow Animation Concept (Step-by-Step)**

1. **Step 1 – User Prompt:**

“Greet Rajeev, add 23 + 45, and fetch a joke”

1. **Step 2 – Thought Generation:**  
   LLM thinks:

vbnet

CopyEdit

Thought: I need to greet Rajeev first

1. **Step 3 – Tool Selection:**  
   AgentExecutor selects GreetUser(name="Rajeev")
2. **Step 4 – Tool Execution:**

makefile

CopyEdit

Observation: Hello Rajeev, welcome!

1. **Step 5 – Feedback Loop:**  
   LLM receives Observation → generates next Thought:

makefile

CopyEdit

Thought: Now calculate 23 + 45

1. **Step 6 – Tool Execution:**

makefile

CopyEdit

Action: Calculator(a=23, b=45)

Observation: 68

1. **Step 7 – Continue Loop:**

vbnet

CopyEdit

Thought: Fetch a joke

Action: GetJoke()

Observation: Why did the chicken cross the road? To get to the other side!

1. **Step 8 – Termination:**

vbnet

CopyEdit

Final Answer: Hello Rajeev, welcome! The sum of 23 + 45 is 68. Here's a joke: Why did the chicken cross the road? To get to the other side!

**3️⃣ Key Points Visualized**

* **Looping:** Thought → Action → Observation repeats until final answer.
* **Streaming:** Each Observation can be sent **in real-time** to frontend.
* **Intermediate Steps:** Can be logged or displayed for debugging.
* **Max Iterations:** Prevents infinite loops.

I can next **generate an actual animated flow diagram** that visually shows:

* Boxes moving: Thought → Action → Observation → back to LLM
* Streaming arrows
* Final aggregation into user response

This will be a **full visual animation you can use in presentations or interactive UI**.

Do you want me to generate that animation next?

**You said:**

now give me full realistic full code examples in fast api full stack using all learnings - about - 1) Models • Language models -> LLMs / Chat models: ChatOpenAI, ChatAnthropic, local LLMs (HF). • Embeddings: OpenAIEmbeddings, HuggingFaceEmbeddings. • (Optional) Other modalities: vision, audio models (varies by provider). 2) Prompts • Templates: PromptTemplate, ChatPromptTemplate (system/human/ai messages). • Examples: few-shot + example selectors. • Output control: structured output / output parsers (JSON schema, regex). 3) Indexes (RAG plumbing) • Loaders: PDFs/CSV/HTML/web/DB via langchain\_community.document\_loaders. • Splitters: RecursiveCharacterTextSplitter, token/semantic splitters. • Vector stores: Chroma, FAISS, Pinecone, Weaviate, Milvus. • Retrievers: .as\_retriever() with k, MMR; multi-query, ensemble, compression/rerank. 4) Chains • Basic: LLMChain (prompt → model). • Retrieval: RetrievalQA, ConversationalRetrievalChain. • Doc chains: map-reduce / refine / stuff. • LCEL (Runnable graph): RunnableSequence, branching, parallel, retries, fallbacks, streaming. 5) Memory • ConversationBufferMemory (all turns), BufferWindowMemory (last k). • ConversationSummaryMemory (token-efficient). • EntityMemory, VectorStoreRetrieverMemory (long-term recall). • Backends: in-memory, Redis, SQL, etc. 6) Agents • Tools: search, calculator, code, HTTP, SQL, custom tools. • Policies/Types: tool-calling (functions), ReAct, plan-and-execute. • Runtime: AgentExecutor (+ create\_tool\_calling\_agent / ReAct builders).

**ChatGPT said:**

Absolutely! Let’s create a **realistic, full-stack FastAPI + React project** that integrates all the key **LangChain learnings** you mentioned:

* **Models** → LLMs, embeddings
* **Prompts** → Templates, few-shot examples, structured output
* **Indexes (RAG)** → Document loaders, splitters, vector stores, retrievers
* **Chains** → LLMChain, RetrievalQA, LCEL
* **Memory** → ConversationBufferMemory, VectorStoreRetrieverMemory
* **Agents** → Tools, ReAct agent, AgentExecutor

We’ll structure this as a **multi-purpose GenAI assistant** that can:

* Answer questions from documents (RAG)
* Execute tools like calculator or joke fetch
* Maintain memory
* Stream reasoning steps to a React UI

**1️⃣ Backend (FastAPI)**

**Directory structure**

css

CopyEdit

backend/

├─ main.py

├─ tools.py

├─ chains.py

├─ memory.py

├─ models.py

├─ prompts.py

├─ rag.py

**models.py – LLMs & embeddings**

python

CopyEdit

from langchain.chat\_models import ChatOpenAI

from langchain.embeddings.openai import OpenAIEmbeddings

# Language Model

llm = ChatOpenAI(temperature=0, model\_name="gpt-4")

# Embeddings for RAG

embeddings = OpenAIEmbeddings()

**prompts.py – Prompt Templates & Few-shot**

python

CopyEdit

from langchain.prompts import PromptTemplate, ChatPromptTemplate, SystemMessagePromptTemplate, HumanMessagePromptTemplate

# Simple template

question\_template = PromptTemplate(

input\_variables=["question"],

template="Answer the following question concisely:\n\n{question}"

)

# Few-shot chat template

chat\_template = ChatPromptTemplate.from\_messages([

SystemMessagePromptTemplate.from\_template("You are a helpful AI assistant."),

HumanMessagePromptTemplate.from\_template("{question}")

])

**memory.py – Conversation / Vector Memory**

python

CopyEdit

from langchain.memory import ConversationBufferMemory, VectorStoreRetrieverMemory

from langchain.vectorstores import Chroma

# In-memory conversation

conv\_memory = ConversationBufferMemory(memory\_key="chat\_history", return\_messages=True)

# Vector store memory for long-term recall

vectordb = Chroma(persist\_directory="./chroma", embedding\_function=None)

vector\_memory = VectorStoreRetrieverMemory(retriever=vectordb.as\_retriever())

**rag.py – Loaders, Splitters, Vector Store**

python

CopyEdit

from langchain.document\_loaders import PyPDFLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain.vectorstores import Chroma

from langchain.chains import RetrievalQA

# Load PDF

loader = PyPDFLoader("docs/sample.pdf")

docs = loader.load()

# Split documents

splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=50)

chunks = splitter.split\_documents(docs)

# Create vector store

vectordb = Chroma.from\_documents(chunks, embedding\_function=None, persist\_directory="./chroma")

# Retriever

retriever = vectordb.as\_retriever(search\_type="mmr", k=3)

# Retrieval QA chain

retrieval\_chain = RetrievalQA.from\_chain\_type(

llm=llm,

chain\_type="stuff",

retriever=retriever,

return\_source\_documents=True

)

**tools.py – Custom Agent Tools**

python

CopyEdit

from langchain.agents import Tool

import requests

def greet\_user(name: str) -> str:

return f"Hello {name}, welcome!"

def add\_numbers(a: int, b: int) -> int:

return a + b

def get\_joke() -> str:

res = requests.get("https://official-joke-api.appspot.com/jokes/random")

joke = res.json()

return f"{joke['setup']} {joke['punchline']}"

greet\_tool = Tool(name="GreetUser", func=greet\_user, description="Greets a user by name")

calc\_tool = Tool(name="Calculator", func=add\_numbers, description="Adds two numbers")

joke\_tool = Tool(name="GetJoke", func=get\_joke, description="Fetches a joke")

tools = [greet\_tool, calc\_tool, joke\_tool]

**chains.py – LCEL / Multi-step / Streaming**

python

CopyEdit

from langchain.chains import LLMChain

from langchain.chat\_models import ChatOpenAI

from langchain.prompts import PromptTemplate

from langchain.schema import BaseOutputParser

# Example: structured output parser

class JSONParser(BaseOutputParser):

def parse(self, text: str):

import json

try:

return json.loads(text)

except:

return {"raw": text}

template = PromptTemplate(

input\_variables=["question"],

template="Answer the question and output JSON with keys: answer, summary\n\nQuestion: {question}"

)

llmchain = LLMChain(llm=ChatOpenAI(temperature=0), prompt=template, output\_parser=JSONParser())

**main.py – FastAPI with ReAct Agent + RAG + Memory + Streaming**

python

CopyEdit

from fastapi import FastAPI, WebSocket

from fastapi.middleware.cors import CORSMiddleware

from langchain.agents import initialize\_agent, AgentExecutor

from models import llm

from tools import tools

from rag import retrieval\_chain

from memory import conv\_memory

app = FastAPI()

# Allow React frontend

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"],

allow\_methods=["\*"],

allow\_headers=["\*"],

)

# Initialize ReAct Agent

agent = initialize\_agent(tools=tools, llm=llm, agent="react", verbose=True)

executor = AgentExecutor(agent=agent, tools=tools, max\_iterations=10, verbose=True)

@app.websocket("/ws/agent")

async def websocket\_agent(ws: WebSocket):

await ws.accept()

try:

while True:

data = await ws.receive\_text()

# Run RAG QA first

rag\_result = retrieval\_chain.run(data)

# Run agent executor with streaming steps

import asyncio

loop = asyncio.get\_event\_loop()

def run\_agent():

steps = []

import builtins

orig\_print = builtins.print

def mock\_print(\*args, \*\*kwargs):

msg = " ".join(str(a) for a in args)

steps.append(msg)

builtins.print = mock\_print

try:

final\_res = executor.run(data)

finally:

builtins.print = orig\_print

return steps, final\_res

steps, final\_res = await loop.run\_in\_executor(None, run\_agent)

# Stream intermediate steps

import json

for s in steps:

await ws.send\_text(json.dumps({"type":"step","content":s}))

# Send final agent response + RAG result

await ws.send\_text(json.dumps({"type":"final","content":{"agent":final\_res,"rag":rag\_result}}))

except Exception as e:

await ws.close()

**2️⃣ Frontend (React)**

jsx

CopyEdit

// frontend/src/App.jsx

import { useEffect, useState, useRef } from "react";

export default function App() {

const [messages, setMessages] = useState([]);

const [input, setInput] = useState("");

const wsRef = useRef(null);

useEffect(() => {

wsRef.current = new WebSocket("ws://localhost:8000/ws/agent");

wsRef.current.onmessage = (event) => {

const data = JSON.parse(event.data);

setMessages((prev) => [...prev, data]);

};

return () => wsRef.current.close();

}, []);

const handleSend = () => {

if (input && wsRef.current) {

wsRef.current.send(input);

setInput("");

}

};

return (

<div style={{ padding: "20px", fontFamily: "sans-serif" }}>

<h1>GenAI Full-Stack Demo</h1>

<div

style={{

border: "1px solid #ccc",

padding: "10px",

height: "400px",

overflowY: "scroll",

background: "#f9f9f9",

}}

>

{messages.map((msg, idx) => (

<div key={idx}>

{msg.type === "step" && <p>💭 {msg.content}</p>}

{msg.type === "final" && (

<p>✅ Agent: {msg.content.agent}<br/>📚 RAG: {msg.content.rag}</p>

)}

</div>

))}

</div>

<input

type="text"

value={input}

onChange={(e) => setInput(e.target.value)}

placeholder="Ask something..."

style={{ width: "70%", padding: "8px", marginRight: "10px" }}

/>

<button onClick={handleSend} style={{ padding: "8px 16px" }}>

Send

</button>

</div>

);

}

**✅ Features Covered in This Full Stack Example**

1. **Models** → ChatOpenAI LLM + OpenAI embeddings
2. **Prompts** → PromptTemplate + JSON output parser
3. **RAG Index** → PDF loader, splitter, Chroma vector store, retriever
4. **Chains** → RetrievalQA + LLMChain (structured JSON output)
5. **Memory** → ConversationBufferMemory + optional VectorStoreRetrieverMemory
6. **Agents** → ReAct agent with multiple tools + AgentExecutor
7. **Streaming** → Thought → Action → Observation → Final Answer to React UI

This setup is **production-ready and modular**:

* Add more tools → tools.py
* Add more document sources → rag.py
* Add multiple agents → main.py with multiple executors
* Add multi-turn conversation → memory.py

I can next make a **diagram showing the full data flow** from:

* **Frontend → WebSocket → AgentExecutor → RAG → Tools → Memory → LLM → Final Response**

This will **visually map all 6 areas you asked for** in one single architecture.