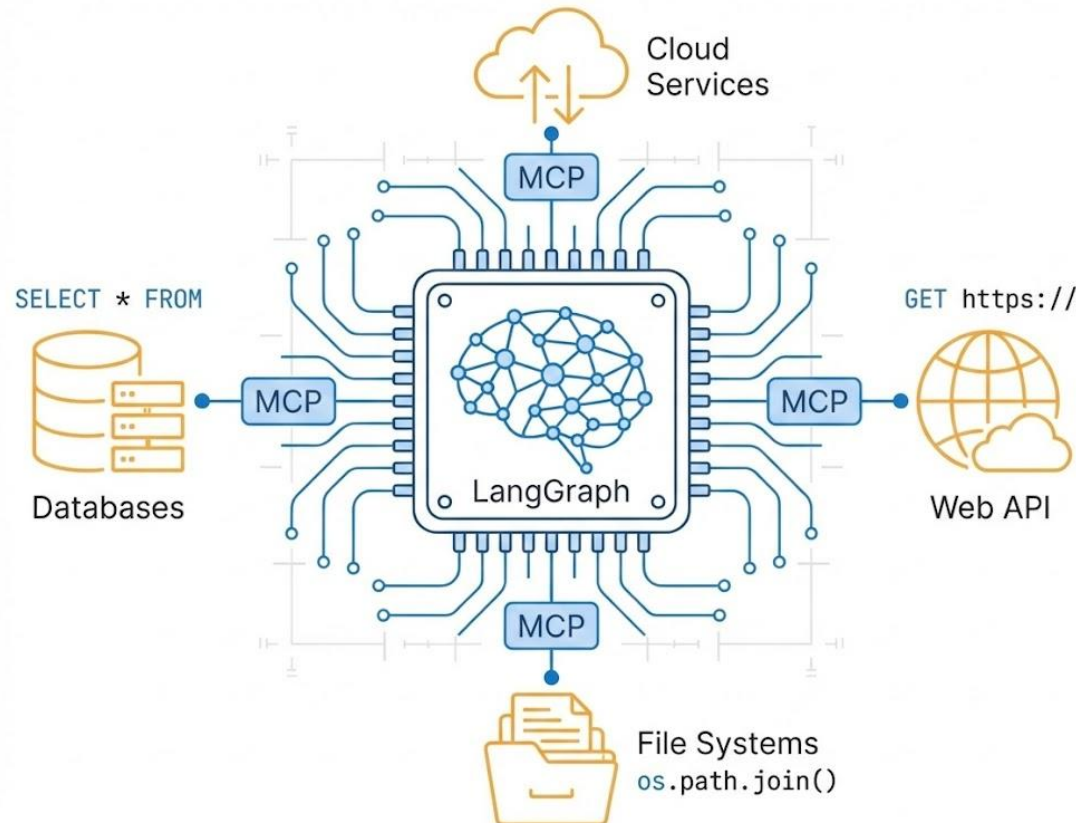


NeuroSymbolic Artificial Intelligence at Scale

Parte: 6.1 (2025-26) From RAG LLM to Agentic LLM (Part 1)

Paolo Nesi, paolo.nesi@unifi.it

<https://www.disit.org/>



Gianni Pantaleo

gianni.pantaleo@unifi.it

DISIT LAB

Google Scholar

<https://scholar.google.com/citations?user=ohyLOOUAAAAJ&hl=it>

Structure of the course up to now

- Overview
- What is Symbolic, Classification
- Hybrid solutions
- Physically Informed, PINN
- Deep Reinforced Learning and Symbolic at Scale
- Knowledge \leftrightarrow NN
- **From RAG LLM to Agentic LLM** ←
- MLOps at Scale



Outline

- AI Agents: Definitions & Motivations
- The Roadmap to AI Agents
 - ❖ *Reasoning*
 - ❖ *Acting: Tool calling*
 - ❖ *Reasoning + Acting: the ReAct framework*
- Taxonomies and Design Patterns of AI Agents
- The Model Context Protocol: MCP
- Orchestration Frameworks: LangGraph

Outline

➤ AI Agents: Definitions & Motivations



➤ The Roadmap to AI Agents

- ❖ *Reasoning*
- ❖ *Acting: Tool calling*
- ❖ *Reasoning + Acting: the ReAct framework*

➤ Taxonomies and Design Patterns of AI Agents

➤ The Model Context Protocol: MCP

➤ Orchestration Frameworks: LangGraph

AI Agents: Definitions & Motivations



LLM/RAG Limitations → Motivation for Agents

- **LLM Knowledge Cut-off:** *The knowledge of LLMs is limited and constrained to pretrained data, which is a great limitation when we need to work on updated data or knowledge. RAG can mitigate this issue. However...*
- **No action beyond text generation:** LLM/RAG based chatbots can only process text in input and produce text as output. They *can NOT execute code, call external APIs, write on databases, etc...* Without tools, the only interface between the model and the world is the context window.
- **No iterative planning or reasoning:** LLM/RAG based chatbots produce a single response per request; they *can NOT decompose a complex goal into subtasks, execute steps sequentially or iteratively, observe intermediate results and adapt their strategy* accordingly.
- **No persistent state:** LLMs are inherently stateless: each inference request is independent, so LLMs do not preserve a persistent state (“memory”) across different interactions. Chatbots based on LLMs *may appear stateful only by application-level design*: conversation history is re-injected in the context window at every inference request.
- **No self-correction:** LLMs may hallucinate, and they *have no mechanism to evaluate the quality of their own output or detect failures*; there is no feedback channel back into any reasoning process.

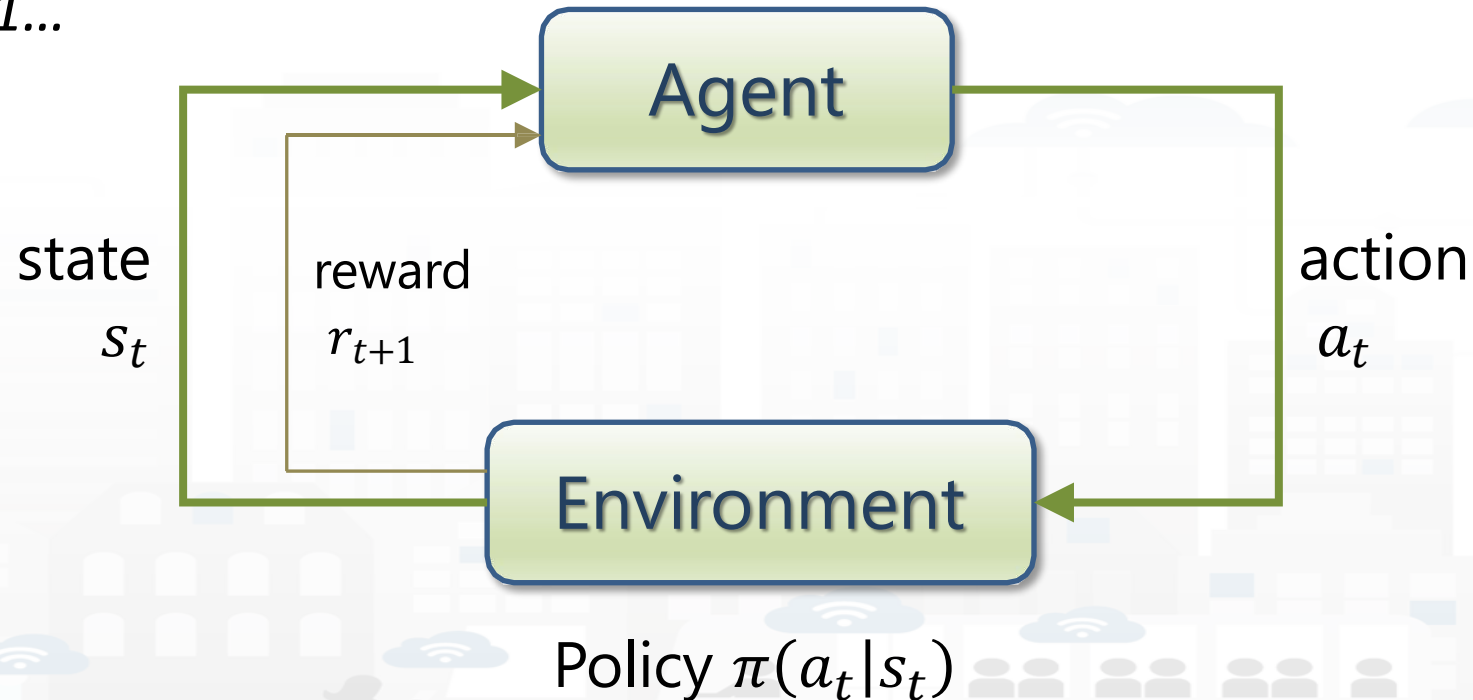
All These limitations motivate the transition to AGENTIC systems!

What is an «AI Agent»

Everybody is nowadays talking about AI Agents based on LLMs...

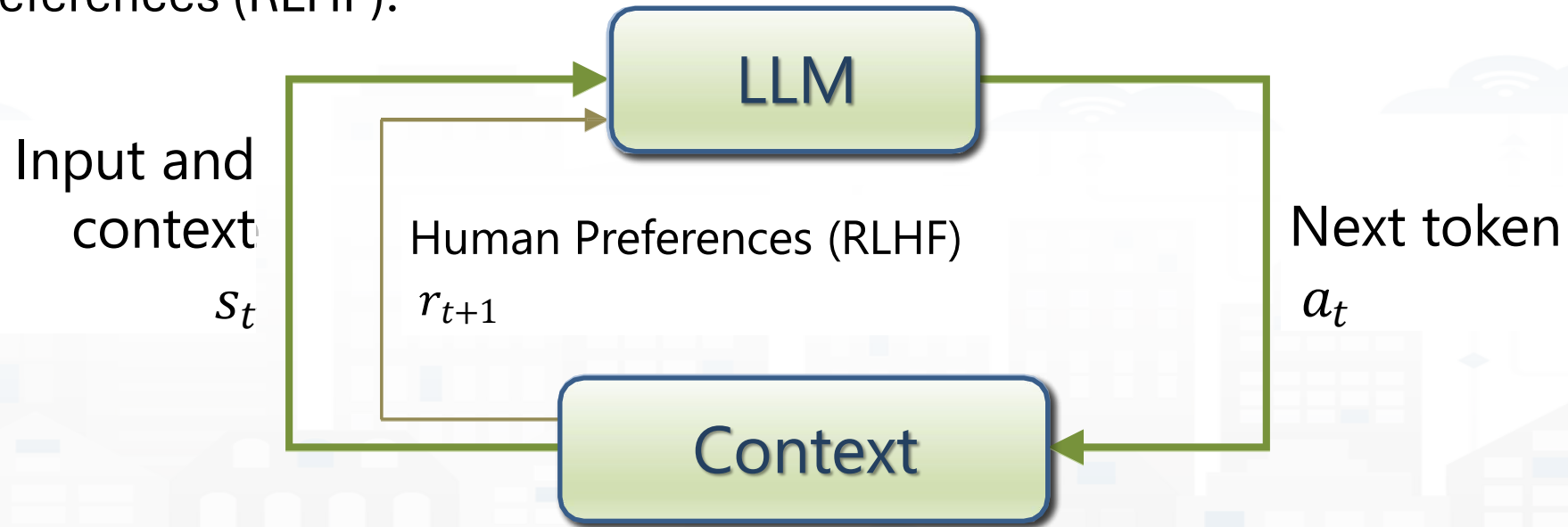
...*but* WHAT is an «Agent» in this sense?

Recalling PART 4-1...



What is an «AI Agent»

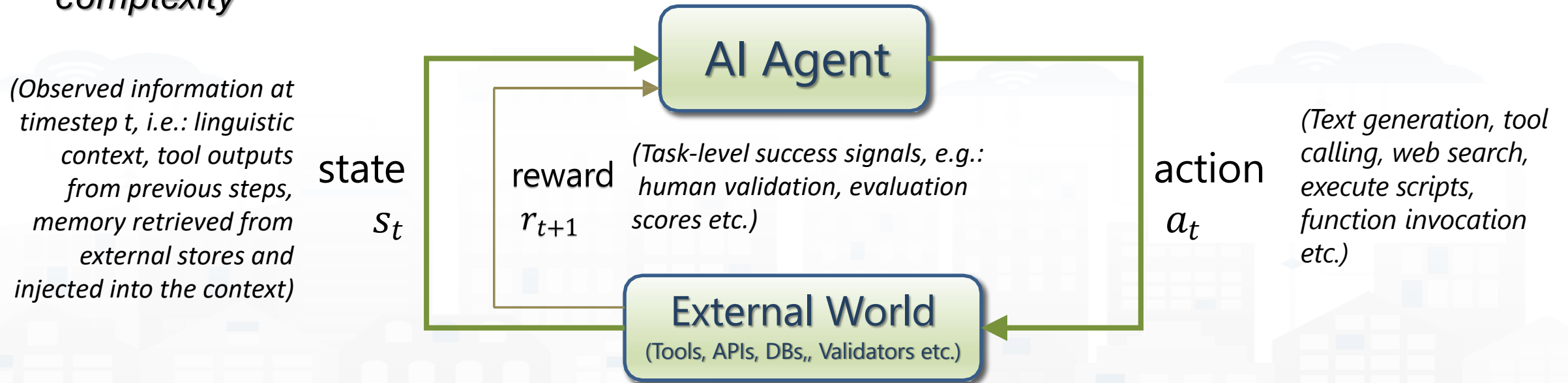
- In the LLM case, the policy $\pi(a_t | s_t)$ is the model probability distribution over the generation of the next token (a_t), conditioned on the input sequence and context generated so far (s_t), with the objective of maximizing a cumulative reward, i.e., aligning token generation with human preferences (RLHF).



Policy: Model probability distribution for next token generation $\pi(a_t | s_t)$

What is an «AI Agent»

- An AI Agent is a system with a policy $\pi(a_t | s_t)$ that selects and executes an action a_t conditioned on observed state s_t , with the objective of maximizing a cumulative reward. *Compared to the LLM case, the action, environment, state and reward involves higher complexity*



Decision Policy: Implemented via Model, Reasoning and Memory $\pi(a_t | s_t)$

What is an «AI Agent»

- Definitions of Agent before the advent of LLMs:

*An agent is anything that can be viewed as **perceiving** its environment through sensors and **acting** upon that environment through actuators [Russell & Norvig, 1995]*

*An Agent is “a computing system situated in an environment, capable of **flexible** and **autonomous** action to meet its design objectives” [Wooldridge & Jennings, 1995]*

- Definitions of LLM-based Agent:

*Agents [...] are systems where **LLMs dynamically direct their own processes and tool usage**, maintaining control over how they accomplish tasks [Anthropic, 2024a]*

*AI agents are systems that can perceive and act upon their environment with a degree of autonomy, using tools as needed to achieve specific goals and **adapt to changing inputs and contexts** [OECD, 2026]*

[Russell & Norvig, 1995] Russell, S. J., & Norvig, P. (1995). Artificial Intelligence: A Modern Approach. Prentice Hall

[Wooldridge & Jennings, 1995] Wooldridge, M. and Jennings, N. R., “Intelligent agents: theory and practice”, The Knowledge Engineering Review, Vol. 10:2, 1995

[Anthropic, 2024a] “Building effective agents”. Online: <https://www.anthropic.com/engineering/building-effective-agents>

[OECD, 2026] “The Agentic AI Landscape and its Conceptual Foundations”, OECD Artificial Intelligence Papers, No. 5, February 2026

https://www.oecd.org/content/dam/oecd/en/publications/reports/2026/02/the-agentic-ai-landscape-and-its-conceptual-foundations_a9d4b451/396cf758-en.pdf

What is an «AI Agent»

- Key features and properties of AI Agents:
 1. Tool use: agents interact with external systems;
 2. Planning: agents decompose goals into sub-tasks;
 3. Memory: agents maintain state across interactions;
 4. Autonomy: agents may act without step-by-step human instruction (but still with human-in-the-loop);
 5. Self-reflection: agents evaluate and correct their own outputs.
- The LLM is the cognitive engine; the agent is the system that gives it a plan, tools, memory and evaluation solutions.

The Five Main Components of AI Agents

1) **LLM.** The LLM serves as the core inference engine of an Agentic AI system. It is responsible for:

- **Goal Interpretation:** Parsing and grounding user intent into a structured task representation.
- **Planning:** Decomposing the goal into an ordered sequence of sub-tasks or actions. The *plan is not pre-coded or rule-based*, but it *emerges dynamically* by the model reasoning at inference time, conditioned on the user input, the available tools, and the intermediate results observed so far.
- **Reasoning:** Evaluating intermediate outputs, resolving ambiguities, and selecting optimal next actions.
- **Tool Selection:** Identifying and invoking the appropriate external tool for each sub-task.
- **Communication:** Generating structured outputs for users (natural language responses), other tools, and peer agents in multi-agent systems.

The Five Main Components of AI Agents

2) Orchestrator. The Orchestrator is the control-flow layer that translates the LLM-generated plan into concrete execution steps. It coordinates tool calls, manages branching logic, and enforces execution policies:

- **Task Sequencing:** Dispatches sub-tasks in the order provided by the plan, respecting dependencies.
- **Conditional Routing:** Implements fallback paths on tool failure or unexpected output (e.g., LinkedIn API unavailable → fallback to Indeed).
- **Retry Logic:** Automatically re-invokes a tool upon transient errors, with configurable back-off policies.
- **Looping / Iteration:** Supports iterative execution cycles when the termination condition has not yet been met (e.g., polling at a fixed interval).

The Five Main Components of AI Agents

3) Tools. Tools are executable interfaces that extend the agent's capabilities beyond the LLM's parametric knowledge. The LLM selects and invokes tools via structured function calls. Main categories:

- **Action Tools:** Functions, REST APIs, shell commands, RPA scripts — enabling side-effects in external systems.
- **Knowledge Retrieval Tools:** Search engines, vector stores, SQL/NoSQL databases — providing grounded, up-to-date context at inference time.

The Five Main Components of AI Agents

4) Memory. Memory provides the agent with persistence across interaction turns and tasks. Three main scopes:

- **Short-Term Memory (In-Context):** The active context window — conversation history, intermediate results, tool outputs within the current session.
- **Long-Term Memory (External Store):** Persistent facts, user preferences, and learned patterns stored in a vector DB or key-value store, retrieved on demand.
- **State Tracking:** Maintains a structured representation of task progress, completed steps, and remaining goals.

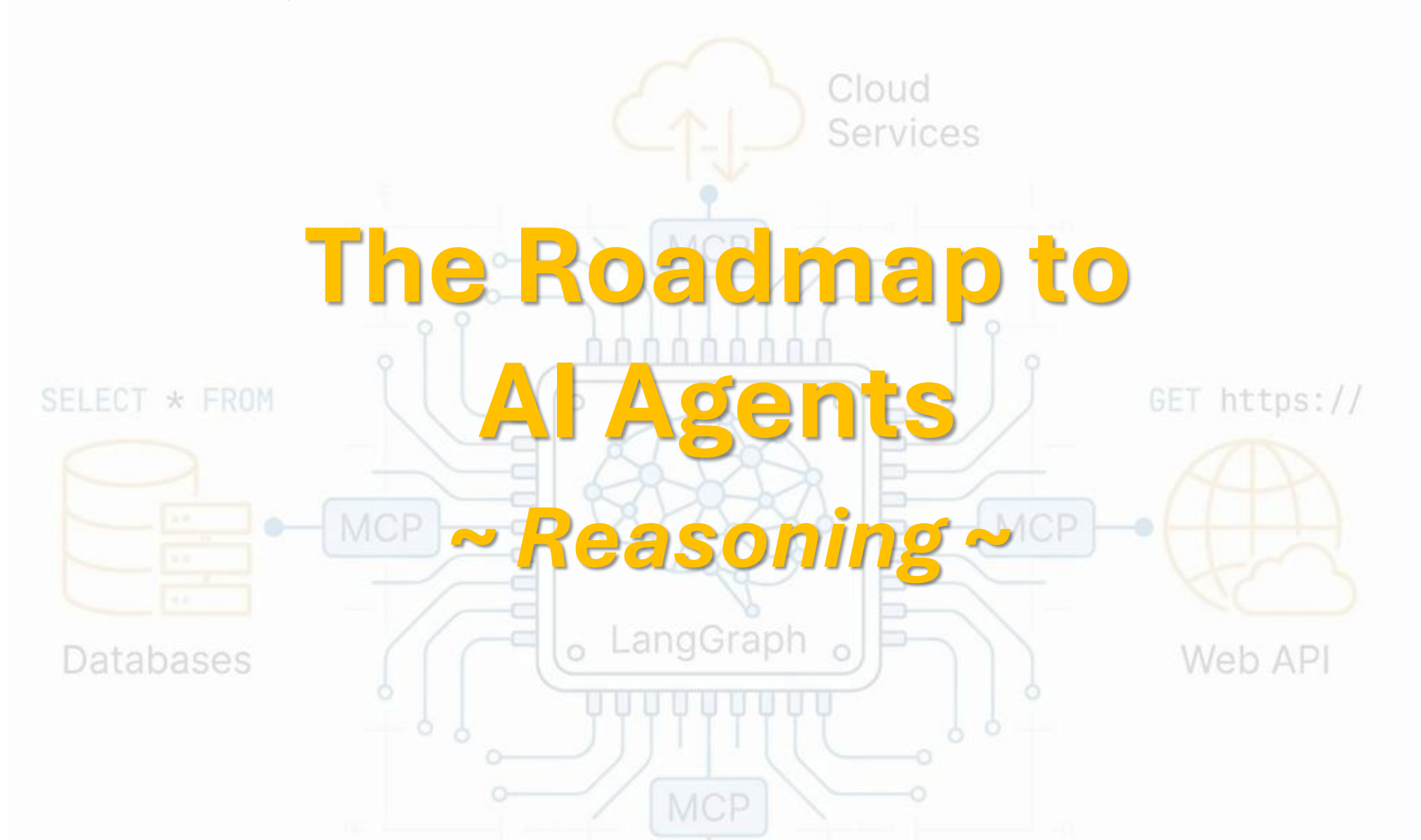
The Five Main Components of AI Agents

5) Supervisor (Human-in-the-Loop). The Supervisor component implements HITL (Human-in-the-Loop) oversight, providing explicit checkpoints at which human judgment is injected into the agentic workflow. Key roles:

- Authorization gate for sensitive or irreversible actions.
- Error correction and plan refinement before execution resumes.
- Enforcement of safety, compliance, and policy constraints.
- Configurable autonomy level (full automation ↔ fully supervised).

Outline

- AI Agents: Definitions & Motivations
- **The Roadmap to AI Agents**
 - ❖ **Reasoning** ←
 - ❖ *Acting: Tool calling*
 - ❖ *Reasoning + Acting: the ReAct framework*
- Taxonomies and Design Patterns of AI Agents
- The Model Context Protocol: MCP
- Orchestration Frameworks: LangGraph



The Reasoning problem in vanilla LLMs: Flat Scaling Curve

- Increasing the LLM model dimension does not improve performances in solving complex tasks that require multi-hop reasoning, advanced math, symbolic reasoning etc.
- This issue has been outlined by [Wei et al., 2022] as the LLMs **Flat Scaling Curve**.
- Mitigation to the “Flat Scaling Curve” issue: the **Chain-of-Thought (CoT) Prompting** [Wei et al., 2022].

[Wei et al., 2022] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D., “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”, in Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS’22). Article No.: 1800, Pages 24824 – 24837, 2022. arXiv:2201.11903, 2022.

Chain-of-Thought (CoT) Prompting

- The Chain-of-Thought is a prompting technique that generates a series of intermediate reasoning steps (thoughts) before the final answer, providing them to the LLM as few-shot examples by directly injecting them in the context.



CoT Key results (GSM8K math benchmark):

PaLM 540B + CoT: **56.9%**

VS

PaLM 540B alone: **17.9%**

CoT has demonstrated to triple performances on complex arithmetic tasks

Chain-of-Thought (CoT) Prompting

- The Zero-shot CoT variant, proposed by [Kojima et al., 2022], shows that simply adding “Let's think step by step” to the prompt is enough to elicit chain reasoning without examples.
- This shows that the ability to reason in steps is already implicit in models large enough, and prompting activates it.

Let's Try It !

[Kojima et al., 2022] Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., & Iwasawa, Y., “Large Language Models are Zero-Shot Reasoners”, in Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS'22). Article No.: 1613, Pages 22199 - 22213. arXiv:2205.11916, 2022.

Chain-of-Thought (CoT) Prompting

Ollama is an open-source platform that allows users to run LLMs locally.

Install Ollama: <https://ollama.com/download>

```
# Linux (Install and run the service)
```

```
curl -fsSL https://ollama.com/install.sh | sh
```

```
# macOS
```

```
# Download the .dmg file and drag it in /Applications
```

```
# Windows
```

```
# Download the .exe file and install it
```

```
# Download (pull) an LLM
```

```
# (e.g.: qwen3.5 with 4 Billion parameters can easily run on GPUs with 8 GB of VRAM)
```

```
ollama pull qwen3.5:4b
```

```
# Verify that ollama server is up and running
```

```
ollama list
```

Chain-of-Thought (CoT) Prompting Example

- Requirements: `pip install ollama`

```
import ollama
```

```
# CoT: add 'Think step by step' to the prompt
```

```
# and the LLM generates reasoning steps without additional Fine-Tuning
```

```
task = (  
    "A train travels 120 km in 1.5 h.\n"  
    "A car covers the same distance in 2 h.\n"  
    "How much faster is the train (km/h) than the car?"  
)  
prompt = task + "\n\nThink step by step, then give the Answer."
```

```
model="qwen3.5:4b"
```

```
response = ollama.chat(  
    model=model,  
    messages=[{"role": "user", "content": prompt}]  
)
```

```
print(response["message"]["content"])
```

Chain-of-Thought (CoT) Prompting Example

- **Output:**

To find out how much faster the train is than the car, we need to calculate the speed of each vehicle and then find the difference.

****Step 1: Calculate the speed of the train.****

- * $\text{Speed} = \text{Distance} / \text{Time}$
- * $\text{Distance} = 120 \text{ km}$
- * $\text{Time} = 1.5 \text{ hours}$
- * $\text{Train Speed} = \$120 \ \div \ 1.5 = 80\$ \text{ km/h}$

****Step 2: Calculate the speed of the car.****

- * $\text{Speed} = \text{Distance} / \text{Time}$
- * $\text{Distance} = 120 \text{ km}$
- * $\text{Time} = 2 \text{ hours}$
- * $\text{Car Speed} = \$120 \ \div \ 2 = 60\$ \text{ km/h}$

****Step 3: Calculate the difference in speed.****

- * $\text{Difference} = \text{Train Speed} - \text{Car Speed}$
- * $\text{Difference} = \$80 - 60 = 20\$ \text{ km/h}$

The train is 20 km/h faster than the car.

****Answer:**** 20

Chain-of-Thought (CoT) Prompting

Advantages

- ✓ Serialization of complex reasoning into interpretable steps;
- ✓ No model update or fine-tuning required: CoT is a prompt-only technique that works on any LLM without model weights update or fine-tuning.

Limitations

- ✗ Emerging behavior only at large model scale (tens to hundreds of billions of parameters);
- ✗ Strictly linear: no branching or backtracking; cannot explore alternative solutions;
- ✗ Wrong steps propagate through the rest of the chain.

Tree-of-Thoughts (ToT)

- [Yao et al., 2023a] introduce the **Tree-of-Thoughts (ToT)** approach, i.e., a prompting framework where an LLM generates a tree of multiple reasoning paths, instead of following a linear chain.
- Starting from the input problem as root node, at each node, the model produces multiple candidate thoughts, evaluates them, and selects the most promising ones to expand further, instead of following a single linear chain.
- The LLM exploits *Breadth-First Search (BFS)* and *Depth-First Search (DFS)* algorithms to explore the alternative solutions and evaluates them, pruning unpromising branches and backtracking when needed.

ToT Key results (Game of 24):

GPT-4 + ToT: **74%**

VS

GPT-4 + CoT: **4%**

ToT has shown huge performance improvements on complex problem-solving game tasks with respect to CoT.

[Yao et al., 2023a] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y., & Narasimhan, K., “Tree of Thoughts: Deliberate Problem Solving with Large Language Models”, in Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS 2023), Article No.: 517, Pages 11809 – 11822. arXiv:2305.10601, 2023.

Tree-of-Thoughts (ToT)

Advantages

- ✓ Supports backtracking and weak branch pruning: more robust to early-error propagation compared to CoT;
- ✓ Significant improvements for complex planning, search problems and problem-solving tasks.

Limitations

- ✗ Higher computational costs: $O(b^d)$ complexity, where b is the (mean) branching factor (i.e., how many candidate thoughts are generated for each node) and d is the depth;
- ✗ Does not support merging branches which may yield valid reusable insights.

Graph-of-Thoughts (GoT)

- A further step ahead is represented by the **Graph-of-Thoughts (GoT)** paradigm, presented by [Besta et al., 2024].
- GoT models the LLM reasoning process as an arbitrary graph. CoT and ToT are special cases of GoT, where reasoning is constrained to a linear chain or a tree-structured graph, respectively.
- GoT allows richer reasoning structures, supporting merging of thoughts branches, therefore enabling the aggregation and distillation of different insights for the same problem/task.
- GoT supports the refinement of thoughts (graph nodes) through feedback loops.

[Besta et al., 2024] Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Gianinazzi, L., Gajda, J., Lehmann, T., Podstawski, M., Niewiadomski, H., Nyczyk, P., & Hoefler, T., “Graph of Thoughts: Solving Elaborate Problems with Large Language Models”, in Proceedings of the AAAI Conference on Artificial Intelligence, 38(16), 17682–17690. arXiv:2308.09687, 2024.

CoT vs ToT vs GoT

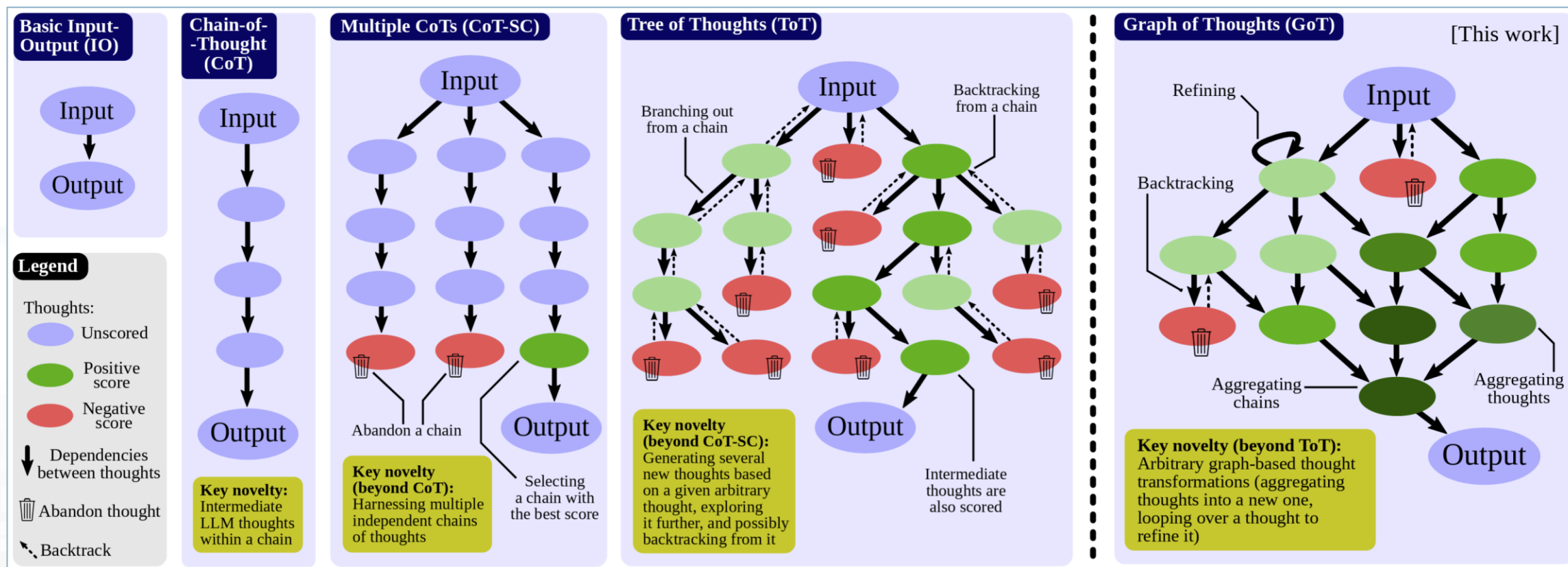


Figure reference: [Besta et al., 2024]

Outline

- AI Agents: Definitions & Motivations
- **The Roadmap to AI Agents**
 - ❖ Reasoning
 - ❖ **Acting: Tool calling** ←
 - ❖ Reasoning + Acting: the ReAct framework
- Taxonomies and Design Patterns of AI Agents
- The Model Context Protocol: MCP
- Orchestration Frameworks: LangGraph



Tool Calling

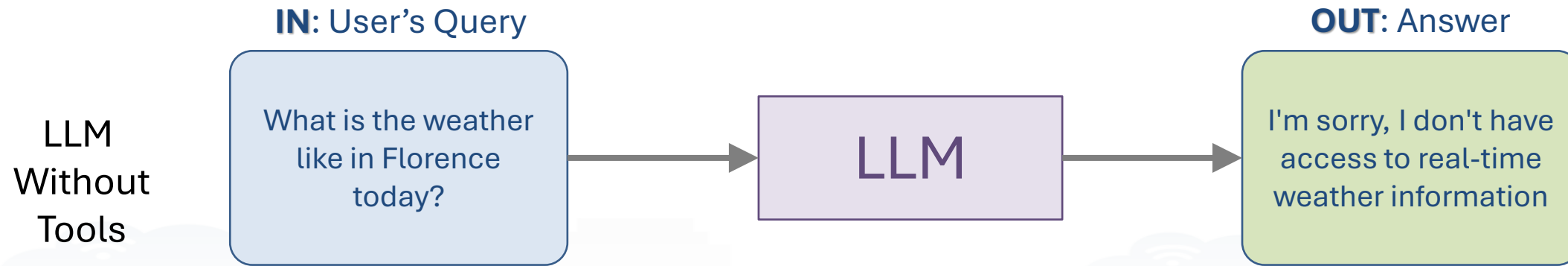
- Reasoning has been proven to be not enough to achieve what we intend as “agentic features”...
- ...Actually, *“reasoning is a static black box, in that the model uses its own internal representations to generate thoughts, and is not grounded in the external world, which limits its ability to reason reactively or update its knowledge”* [Yao et al., 2023b].
- Therefore, **action** is needed!

[Yao et al., 2023b] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y., “ReAct: Synergizing reasoning and acting in language models”, in Proceedings of the 11th International Conference on Learning Representations (ICLR 2023). arXiv preprint arXiv:2210.03629v3, 2023.

Tool Calling

- **“Tool calling**, sometimes referred to as function calling, is a key enabler of agentic AI. It allows autonomous systems to complete complex tasks by dynamically accessing and acting upon external resources.” [IBM, 2025]
- In RAG frameworks (vanilla LLM without tools):
 - relevant document chunks fetched by the retriever from external knowledge base or corpora are injected in the LLM prompt as context;
 - this is a **passive** retrieval, which does not enable an **active** behavior of the system (i.e., execution of functions, tools etc.).
- Tool calling represent the active approach:
 - The signature, definition and description of one or more available functions, API etc. (i.e., the **tools**) is injected in the LLM prompt;
 - the LLM decides autonomously whether and which tool to invoke by interpreting the user’s intent and the following reasoning steps (i.e., from the context);
 - the LLM extracts the parameters required as tool input from the context;
 - the runtime executes the function (outside the LLM’s ecosystem) returning the result as a structured information in the LLM context.

Tool Calling



Tool Calling

IN: Tool description + User's Query

LLM
With
Tools

```
<get_weather(city :str)
function description >
+
What is the weather like in
Florence today?
```

LLM

OUT: Final Answer

In Florence today
there are 22°C with
partly cloudy skies
and 65% humidity.

LLM finds a relevant function call and extract the arguments to be passed as input parameters

```
Call the function get_weather
with arguments:
{city: "Florence"}
```

LLM deduces the final answer from the tool output (returned function result)

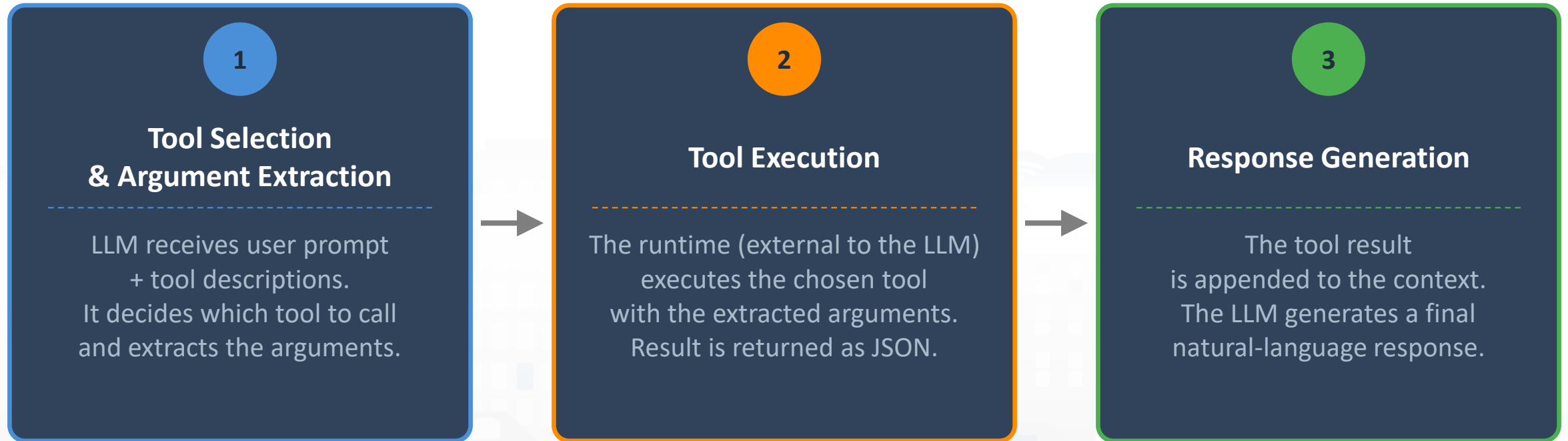
```
{ temperature=22.0,
condition="Partly cloudy",
humidity=65 }
```

Backend

```
get_weather(city):
(function/API) to retrieve
weather information
from city
```

The tool produces a result that is appended to the context

Tool Calling – How it Works



Prompt + [Tool description] → LLM → tool_call(args) → execute() → append result → LLM → response

Tool Calling – How it Works

```
from dataclasses import dataclass  
import requests
```

```
@dataclass
```

```
class WeatherInfo:  
    city: str  
    temperature: float  
    condition: str  
    humidity: int
```

Tool signature; type hints
expose the interface in a
machine-readable way

```
def get_weather(city: str) -> WeatherInfo:
```

```
    """  
    Returns the current weather conditions for the given city.  
    Use when the user asks about current weather, temperature,  
    or atmospheric conditions in a city.  
    Args:  
        city (str): City name, e.g. "Florence", "Rome", "Milan".  
    Returns:  
        WeatherInfo: Object with temperature (°C), condition and humidity.  
    """
```

Tool description (docstring): description of what
the function does and list of the
required/optional parameters. This represents
the source of the semantic contract with the LLM

```
url = f"https://wttr.in/{city}?format=j1"  
r = requests.get(url, timeout=5)  
r.raise_for_status()  
d = r.json()["current_condition"][0]
```

Tool logic (transparent to the LLM)

```
return WeatherInfo(  
    city=city,  
    temperature=float(d["temp_C"]),  
    condition=d["weatherDesc"][0]["value"],  
    humidity=int(d["humidity"])  
)
```

Returned structured information

Tool example:

- Uses wttr.in – a free, open weather API (no API key required)
- Takes a city name as input
- Returns temperature, condition and humidity
 - The docstring is the semantic contract:

it helps the LLM to understand when and why to invoke the tool

Tool Calling – How it Works

①a – The LLM receives the user's prompt and the tool description

```
System: <function API>
def get_weather(city: str) -> WeatherInfo:
    """Returns current weather. Use when user asks about weather."""
User: What is the weather like in Florence right now?
```

①b – The LLM extract the arguments and decide which tool to call

```
{
  "name": "get_weather",
  "arguments": { "city": "Florence" }
}
```

② – The Runtime executes the tool and returns a result

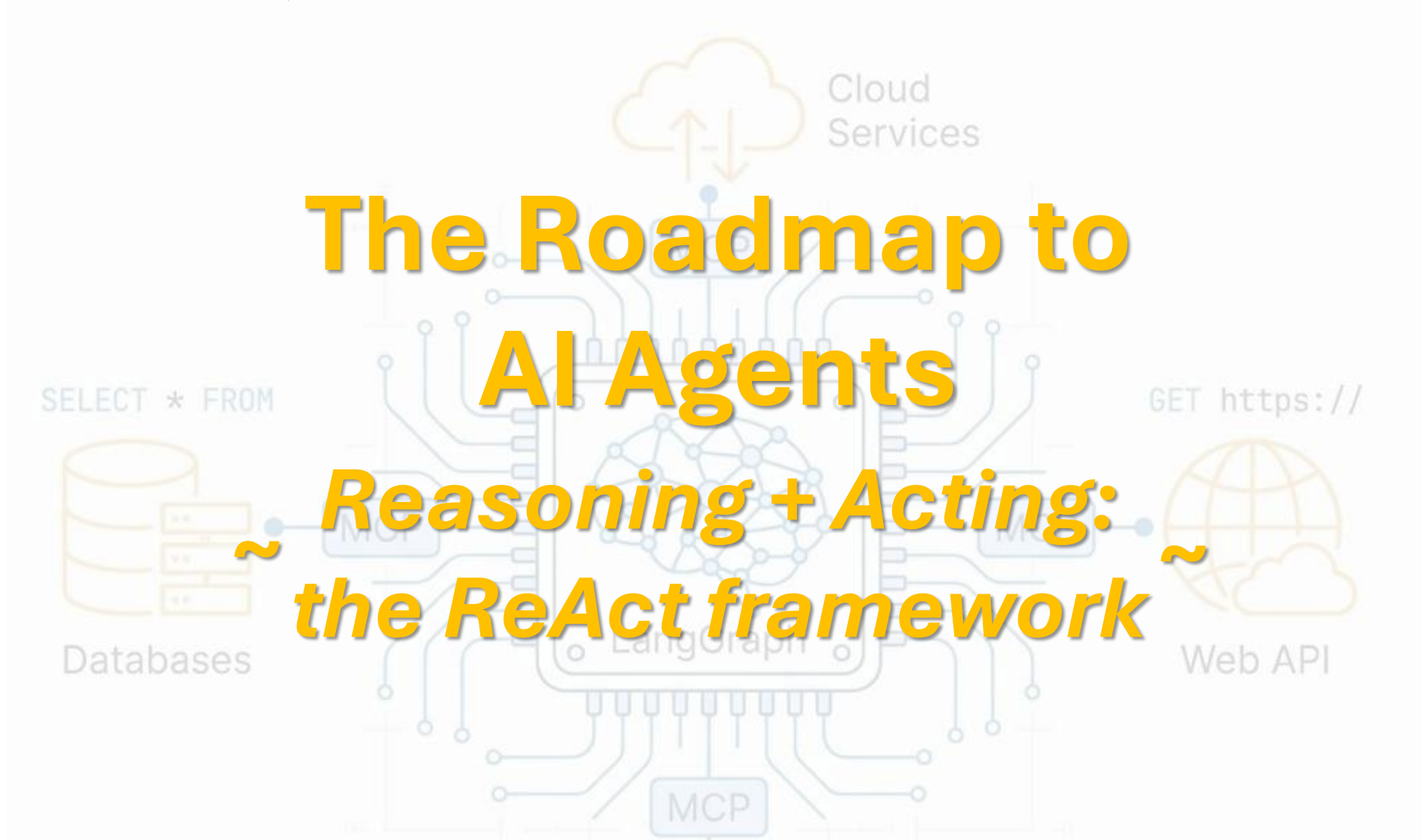
```
Tool result: WeatherInfo(city="Florence", temperature=22.0,
  condition="Partly cloudy", humidity=65)
```

③ – The tool result (observation) is appended to the context; the LLM parses it and generates the final natural-language answer

```
"In Florence today it is 22°C with partly cloudy skies and 65% humidity."
```

Outline

- AI Agents: Definitions & Motivations
- The Roadmap to AI Agents
 - ❖ Reasoning
 - ❖ Acting: Tool calling
 - ❖ **Reasoning + Acting: the ReAct framework** ←
- Taxonomies and Design Patterns of AI Agents
- The Model Context Protocol: MCP
- Orchestration Frameworks: LangGraph



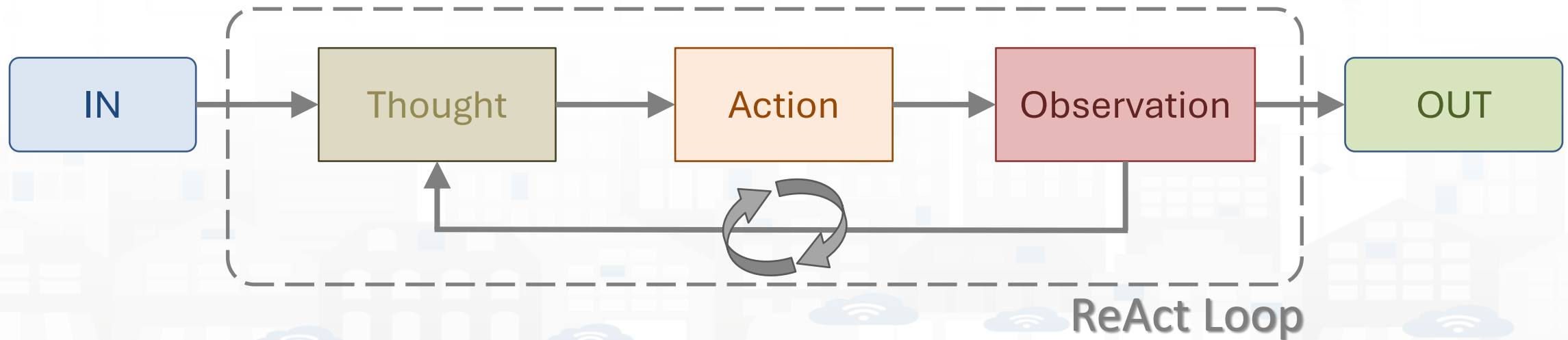
ReAct = Reasoning + Acting

- Complex real-world tasks are generally difficult to be solved in a single LLM call (single-shot)
- A better approach is to decompose the initial goal / user's intent / task into a procedural plan of actionable steps (sub-tasks), where the next action often depends on the result of the previous one.
- A single LLM call (even when using tools) produces one response and terminates: it cannot observe the outcome, recover from failures, or adapt its plan based on intermediate results.
- To solve these issues, [Yao et al., 2023b] have proposed the ReAct framework.

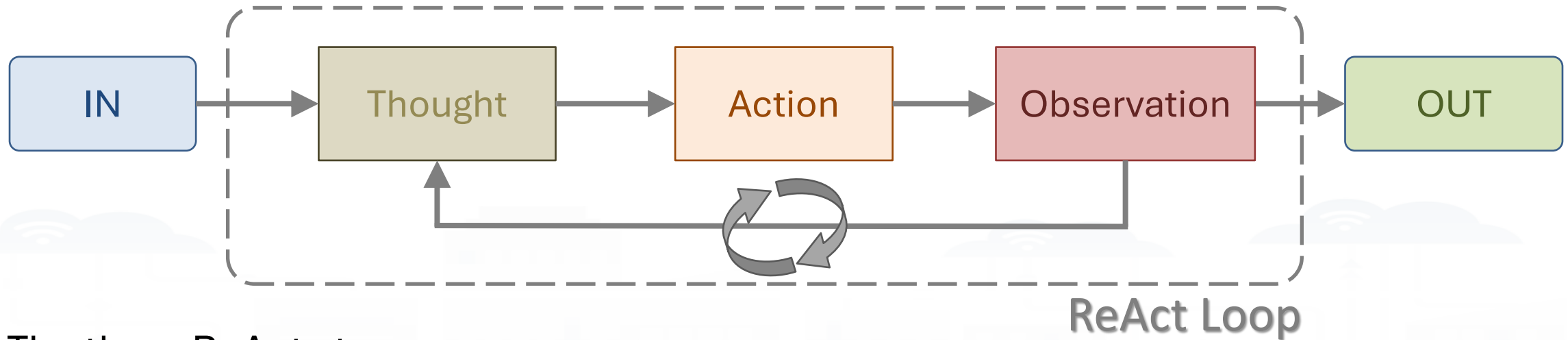
[Yao et al., 2023b] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y., "ReAct: Synergizing reasoning and acting in language models", in Proceedings of the 11th International Conference on Learning Representations (ICLR 2023). arXiv preprint arXiv:2210.03629v3, 2023.

The ReAct Framework

- ReAct formalizes an iterative loop of **Thought** → **Action** → **Observation** in which the model explicitly reasons about what to do next, acts via a tool call, observes the result, and continues until the goal is reached.
- This approach transforms a stateless text generator into a goal-directed agent.



The ReAct Framework



- The three ReAct steps:
 - **Thought**: reason about the current context and decide what to do next
 - **Action**: generate a structured tool call with the right arguments
 - **Observation**: receive the tool result and append it to the context

The ReAct Framework

- What ReAct adds over plain tool calling:
 - Explicit Thought before each action: the model verbalizes its reasoning step
 - Thought updates after each Observation: the plan adapts to new information
 - Interpretable trace: Thought → Action → Observation is human-readable and easy to debug
- The **Thought** component of ReAct is represented by **CoT**: the explicit step-by-step reasoning that decomposes a complex goal into manageable sub-steps is exactly Chain-of-Thought, applied at each iteration of the loop
- *Key difference*: CoT reasons entirely within the model's internal representations and terminates after one pass, while ReAct interrupts the reasoning at each step to act, observe the real world, and resume with updated information.

The ReAct Framework

(2) AlfWorld

You are in the middle of a room. Looking quickly around you, you see a cabinet 6, a cabinet 1, a coffee machine 1, a countertop 3, a stove burner 1, and a toaster 1.
Your task is to: Put some pepper shaker on a drawer.

(2b) ReAct (Reason + Act)

Act 1: **Think**[First I need to find a pepper shaker...more likely to appear in cabinets (1-6), countertops (1-3), ...]
Act 2: Go to cabinet 1
Obs 2: On the cabinet 1, you see a vase 2.
(Here the agent go to cabinet 1, then cabinet 2, 3, then countertop 1 and 2)
Act 6: Go to countertop 3
Obs 6: On the countertop 3, you see a apple 1, a bread 1, a pepper shaker 1, and a vase 3.
Act 7: Take pepper shaker 1 from countertop 3
Obs 7: You pick up the pepper shaker 1 from the countertop 3.
Act 8: **Think**[Now I find a pepper shaker 1. Next, I need to put it in/on drawer 1.]
Act 9: Go to drawer 1
Obs 9: Drawer 1 is closed.
Act 10: Open drawer 1
Obs 10: You open Drawer 1 ...
Act 11: Put pepper shaker 1 in/on drawer 1
Obs 11: You put pepper shaker 1 in/on the drawer 1.

(2a) Act-Only

Act 1: Go to drawer 1
Obs 1: The drawer 1 is closed.
Act 2: Open drawer 1
Obs 2: You open the drawer 1. The drawer 1 is open. In it, you see a dish sponge 2, and a spoon 1.
Act 3: Go to sinkbasin 1
Obs 3: On the sinkbasin 1, you see a dish sponge 3, a spatula 1, and a spoon 2.
Act 4: Take peppershaker 1 from sinkbasin 1
Obs 4: Nothing happens.
Act 5: Take peppershaker 1 from sinkbasin 1
Obs 5: Nothing happens. ✗

Figure Reference: [Yao et al., 2023b]

ReAct Key results (ALFWorld):

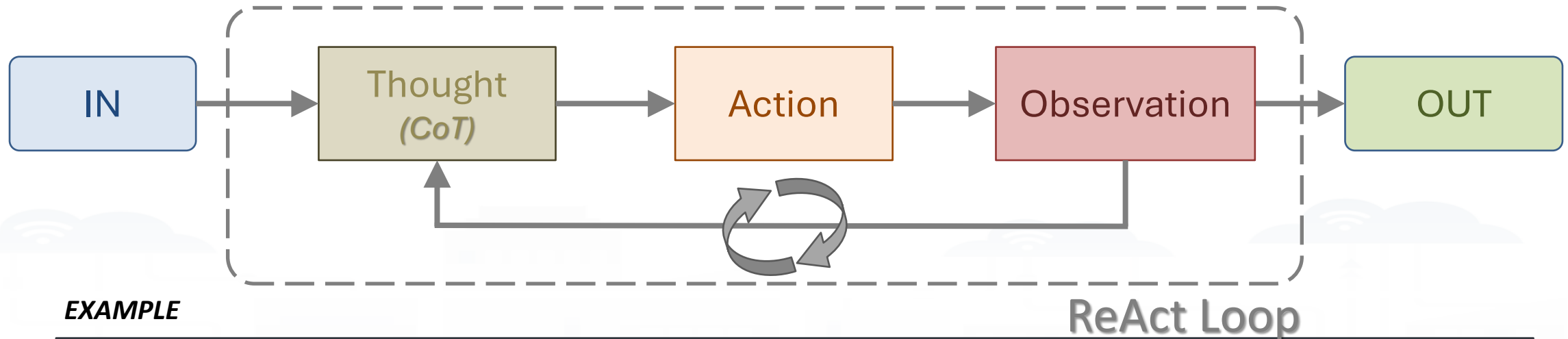
PaLM-540 + ReAct: **71%**

VS

PaLM-540 Act-Only: **45%**

ReAct has shown substantial performance improvement over “Act only” approach on ALFWorld, a text-based interactive environment where an agent must complete multi-step household tasks with natural language commands.

The ReAct Framework



EXAMPLE

```

Input: "What's the weather near me?"
Thought: I need the user's location first. I'll call get_user_location().
Action: get_user_location() → {"city": "Prato", "lat": 43.88, "lon": 11.10}
Observation: LocationInfo(city='Prato', lat=43.88, lon=11.10)
Thought: I have the coordinates. Now I'll call get_weather(43.88, 11.10).
Action: get_weather(lat=43.88, lon=11.10)
Observation: WeatherInfo(temperature=18.0, condition='Overcast', humidity=72)
Output: "Near you in Prato: 18°C, overcast, 72% humidity."
  
```

ReAct: Multiple Tool Calling

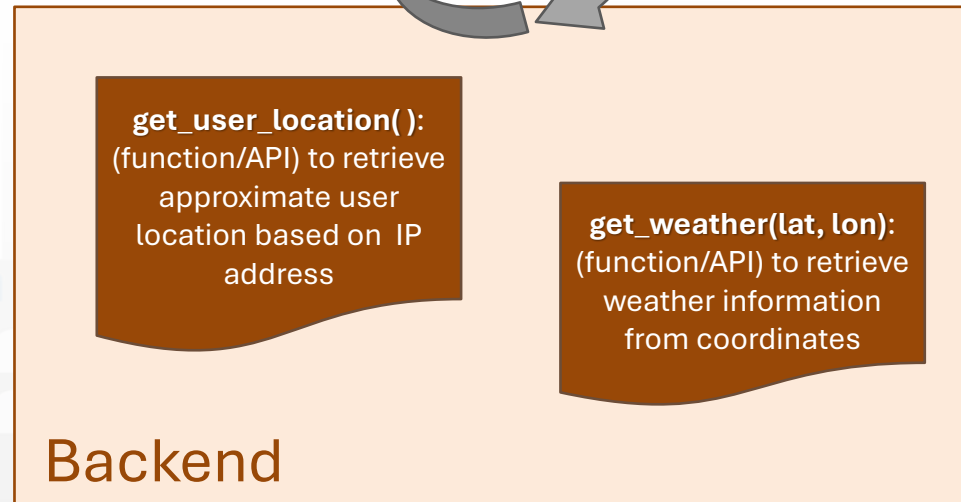
IN: Tool description + User's Query

<LIST of tools
descriptions>
+
What is the weather like near
me?

LLM

OUT: Final Answer

The weather near you in Prato,
Italy, is currently showing as
patchy rain nearby with a
temperature of 21.0°C and 72%
humidity.



Let's Try It !

ReAct: Multiple Tool Calling – How it Works

Tool 1: get_user_location

```
@dataclass
class LocationInfo:
    city: str
    country: str
    lat: float
    lon: float

def get_user_location() -> LocationInfo:
    """
    Returns the approximate current location of the user
    based on their IP address (city, country, coordinates).
    Use when the user refers to their current position
    with expressions like 'near me', 'here', 'where I am'.
    No arguments required.
    """
    r = requests.get("https://ipinfo.io/json", timeout=5)
    r.raise_for_status()
    d = r.json()
    lat, lon = d["loc"].split(",")
    return LocationInfo(
        city = d.get("city", "Unknown"),
        country = d.get("country", "Unknown"),
        lat = float(lat),
        lon = float(lon)
    )
```

Tool 1 JSON Schema: it is generated (automatically or manually) from the Tool type hints, descriptions, parameters and injected in the LLM context by the runtime

```
{
  "type": "function",
  "function": {
    "name": "get_user_location",
    "description": (
      "Returns the approximate current location of the user "
      "based on IP address (city, country, coordinates). "
      "Use when the user refers to their current position "
      "with expressions like 'near me', 'here', 'where I am'."
    ),
    "parameters": {
      "type": "object",
      "properties": {},
      "required": []
    }
  }
}
```

ReAct: Multiple Tool Calling – How it Works

Tool 2: get_weather

```
@dataclass
class WeatherInfo:
    temperature: float
    condition: str
    humidity: int

def get_weather(city: str = None, lat: float = None, lon: float = None) -> WeatherInfo:
    """
    Returns the current weather conditions for a location.
    Use when the user asks about current weather, temperature,
    or atmospheric conditions.
    Args:
        city (str): City name, e.g. "Tokyo", "Rome". Use when the city
            is known by name. Omit if coordinates are available.
        lat (float): Latitude. Use when coordinates are available (e.g.
            from get_user_location). Omit if city name is known.
        lon (float): Longitude. Required if lat is provided.
    """
    location = f"{lat},{lon}" if lat is not None and lon is not None else city
    r = requests.get(f"https://wttr.in/{location}?format=j1", timeout=5)
    r.raise_for_status()
    d = r.json()["current_condition"][0]
    return WeatherInfo(
        temperature = float(d["temp_C"]),
        condition = d["weatherDesc"][0]["value"],
        humidity = int(d["humidity"])
    )
```

ReAct: Multiple Tool Calling – How it Works

— *Tool Registry* —

```
TOOLS = {"get_user_location": get_user_location,  
         "get_weather": get_weather}
```

```
TOOLS_SCHEMA = [  
  {  
    "type": "function",  
    "function": {  
      "name": "get_user_location",  
      "description": (  
        "Returns the approximate current location of the user "  
        "based on IP address (city, country, coordinates). "  
        "Use when the user refers to their current position "  
        "with expressions like 'near me', 'here', 'where I am'."  
      ),  
      "parameters": {  
        "type": "object",  
        "properties": {},  
        "required": []  
      }  
    }  
  ],  
  ]
```

ReAct: Multiple Tool Calling – How it Works

```
{
  "type": "function",
  "function": {
    "name": "get_weather",
    "description": (
      "Returns the current weather conditions for a location. Use when the user asks about current weather, temperature, "
      "or atmospheric conditions."
    ),
    "parameters": {
      "type": "object",
      "properties": {
        "city": {
          "type": "string",
          "description": "City name, e.g. 'Tokyo', 'Rome'. Use when the city is known by name. "
            "Omit if coordinates are available."
        },
        "lat": {
          "type": "number",
          "description": "Latitude. Use when coordinates are available (e.g. from get_user_location). Omit if city name is known."
        },
        "lon": {
          "type": "number",
          "description": "Longitude. Required if lat is provided."
        }
      },
      "required": []
    }
  }
}
```

ReAct: Multiple Tool Calling – How it Works

— ReAct Loop —

```
def run(user_message: str, model: str = "qwen3.5:4b") -> str:
    context = [{"role": "user", "content": user_message}]
    while True:
        msg = ollama.chat(model=model, messages=context, tools=TOOLS_SCHEMA)["message"]
        if not msg.get("tool_calls"):
            return msg["content"]
        context.append(msg)
        for call in msg["tool_calls"]:
            name, args = call["function"]["name"], call["function"]["arguments"]
            print(f"[Action] {name}({args})")
            result = TOOLS[name](**args)
            print(f"[Observation] {result}")
            context.append({"role": "tool", "content": str(result)})
```

— Entry point —

```
if __name__ == "__main__":
    print(run("What's the weather like near me?"))
    print(run("What's the weather like in Tokyo?"))
```

When Ollama operates in tool-calling mode, the model encodes its reasoning directly into the structured tool call object, leaving the content field empty.

Therefore, intermediate Thought steps are not explicitly visible in the output. This is a runtime behavior specific of the Ollama's tool-calling implementation, not a limitation of the ReAct paradigm.

ReAct: Multiple Tool Calling – How it Works

- Output:

```
[Action] get_user_location({})
```

```
[Observation] → LocationInfo(city='Prato', country='IT', lat=43.8805, lon=11.097)
```

```
[Action] get_weather({'lat': 43.8805, 'lon': 11.097})
```

```
[Observation] → WeatherInfo(temperature=20.0, condition='Patchy rain nearby', humidity=64)
```

Based on your current location in Prato, Italy, the weather is currently 20.0°C with patchy rain nearby and humidity at 64%. It looks like there might be light rainfall occurring in the area. You may want to check if you need an umbrella or rain gear depending on your activities!

```
[Action] get_weather({'city': 'Tokyo'})
```

```
[Observation] → WeatherInfo(temperature=18.0, condition='Partly cloudy', humidity=83)
```

The weather in Tokyo is currently partly cloudy with a temperature of 18°C (64°F). The humidity is also quite high at 83%.

ReAct: Multiple Tool Calling – How it Works

- The `TOOLS_SCHEMA` is the explicit JSON contract between the LLM runtime and the model, and it is generated from the function signatures and docstrings (which are human readable for developers).
- The `TOOLS_SCHEMA` must be in a well-structured format (JSON) so that the runtime can parse it efficiently and inject into the prompt. *As currently written, the `TOOLS_SCHEMA` is redundant with the docstrings.*
- This issue may be solved by introducing an **orchestration framework** that can automatically generate the schema directly from the function signature and docstring, and that can also manage the runtime loop, thus making the docstring the single source of truth. *We will see this in the Orchestration Framework section...!*
- Therefore, the orchestrator may act both as a schema inference layer and a conversation execution engine.

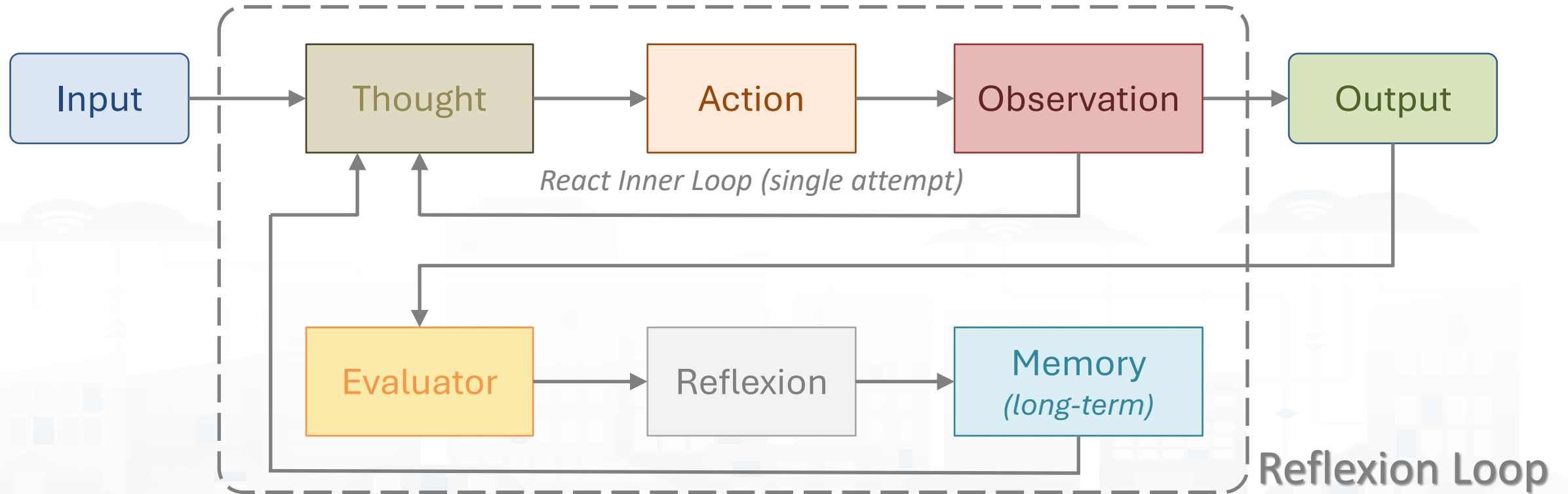
ReAct: Limitations & Issues

- The ReAct *Thought* component directly inherits the CoT main limitations:
 - Structural limitation: the ReAct framework does not know when it fails: the Thought → Action → Observation loop repeats until the model produces a textual response, without verifying that the response is correct.
 - If the model makes an incorrect inference during a certain Thought step, all subsequent steps will be based on that erroneous premise.
 - ReAct lacks a mechanism to self-evaluate its own output.

Reflexion

- The Reflexion framework has been presented by [Shinn et al., 2023].
- Intuition: Humans learn from two mechanisms: **external feedback** and **internal reflection**
- Reflexion wraps ReAct with an outer loop. After each attempt (whether successful or not) the agent evaluates its own output:
 - The agent generates a verbal self-critique in natural language;
 - the agent stores the evaluation in memory;
 - finally, the agent injects the evaluation into the next attempt's context as an explicit instruction.
- The self-critique is stored in an episodic memory and injected into the context of the next attempt as an explicit instruction — no weight updates, no retraining.
- This is called "verbal reinforcement learning": instead of a numeric reward signal which updates model weights, a natural language reflection updates the model's behavior in-context.

The Reflexion Loop



Reflexion Key results (HotPotQA):

ReAct + Reflexion outperforms the ReAct-only baseline by **20%** on HotPotQA, a Wikipedia-based multi-hop Question & Answer dataset requiring reasoning over multiple documents to answer complex questions.

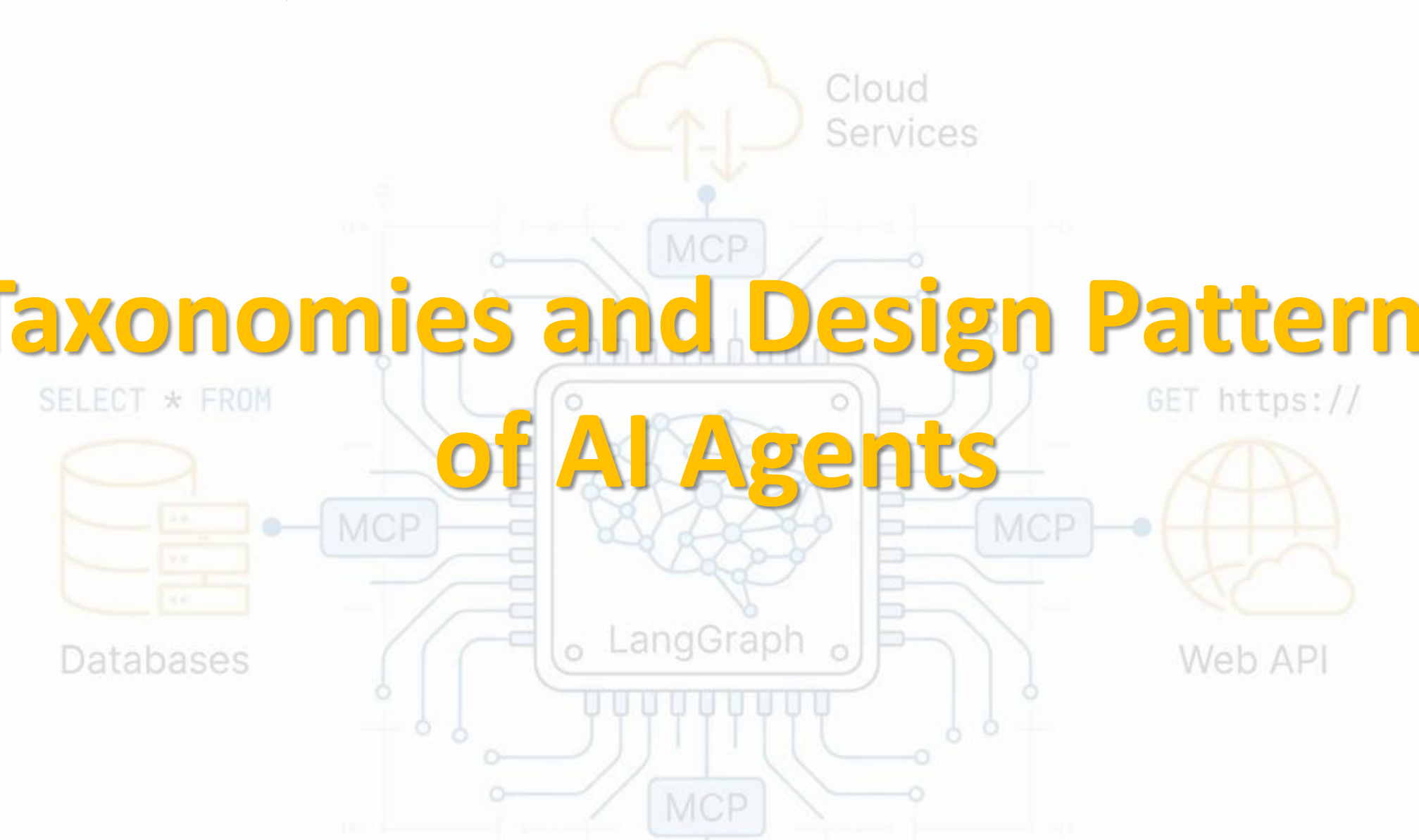
Challenges and Issues

- If there are too many available tools, the context may increase rapidly and the LLM may not be able to correctly retrieve the relevant tools. This can soon lead to performance degradation.
- For this reason, a proper mechanism of tools selection is needed in order to scale.

Outline

- AI Agents: Definitions & Motivations
- The Roadmap to AI Agents
 - ❖ *Reasoning*
 - ❖ *Acting: Tool calling*
 - ❖ *Reasoning + Acting: the ReAct framework*
- **Taxonomies and Design Patterns of AI Agents** ←
- The Model Context Protocol: MCP
- Orchestration Frameworks: LangGraph

Taxonomies and Design Patterns of AI Agents



Taxonomy of LLM-based Agentic Systems



Tool-Calling Agents

- **LLM + Tool Calling**
- **Single reasoning step:**
- Decide tool → Call → Return results/answers
- No planning, no iteration, no memory beyond context

➤ *Suitable for: single Q&A with simple automations*



Reasoning Agents

- **Chain-of-Thought (CoT) + Tool calling**
- **ReAct:** Thought → Action → Observation loop
- Iterative refinement based on tool outputs

➤ *Suitable for: multi-step Q&A and tasks*



Planning Agents

- **Tree/Graph of Thought (ToT/GoT) + Tool calling:** explore multiple reasoning paths
- **Plan-and-Execute:** generate a plan, then execute each step
- Parallel tool calls

➤ *Suitable for: complex multi-step workflows*



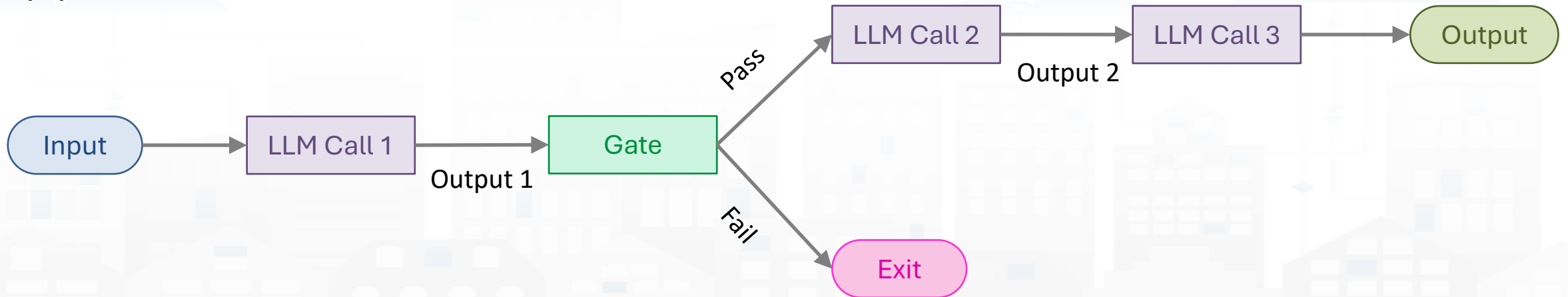
Reflection (Self-Evaluating) Agents

- **Reflexion + Tool calling:** Self-evaluation & retry loop
- Reinforcement from execution feedback

➤ *Suitable for: execution and validation of long-horizon heavy tasks*

Agentic Design Patterns: Prompt Chaining

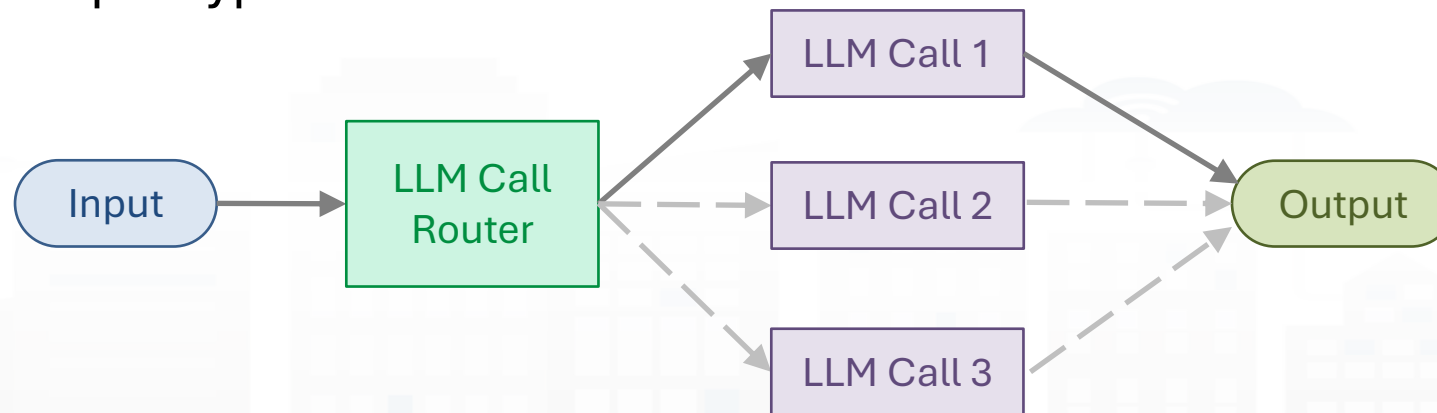
- **5 Agentic Design Patterns** proposed by [Anthropic, 2024a].
- **Prompt Chaining Workflow:** the task is decomposed into a sequence of steps, where each LLM call processes the output of the previous one. Programmatic checks (Gates) can be inserted on any intermediate step to validate outputs and conditionally end or redirect the pipeline.



- This workflow is suitable when the task can be easily decomposed into fixed subtasks.

Agentic Design Patterns: Routing

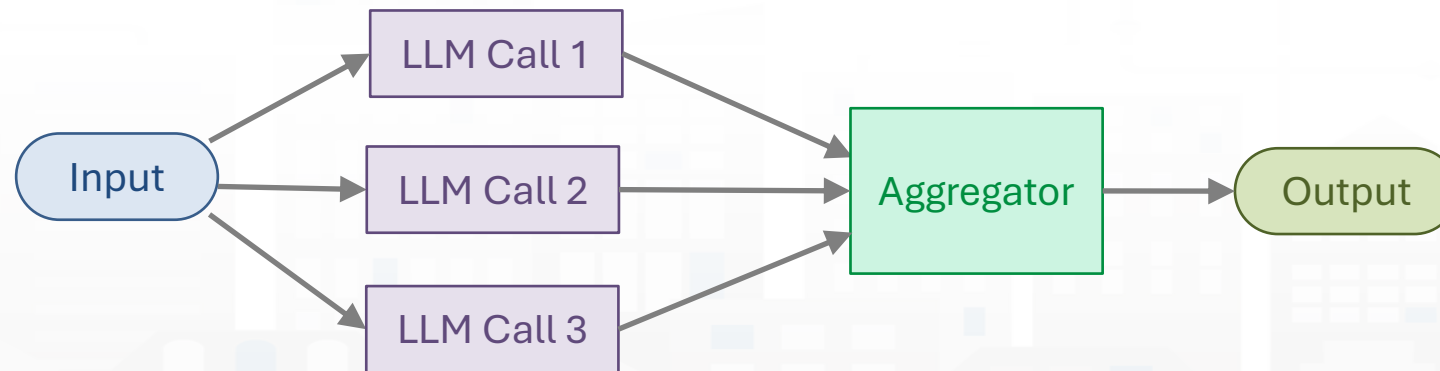
- **Routing:** the LLM Router classifies the input and redirect it to the most suitable specialized LLM call. This pattern enforces separation of concerns and enables prompt optimization. Without routing, a single generalist prompt should trade off performance across heterogeneous input types.



- Routing is suitable for complex tasks presenting heterogeneous input types that may benefit from dedicated handling and prompt optimization. A single generalist prompt cannot be simultaneously optimized for all of them without performance trade-offs.

Agentic Design Patterns: Parallelization

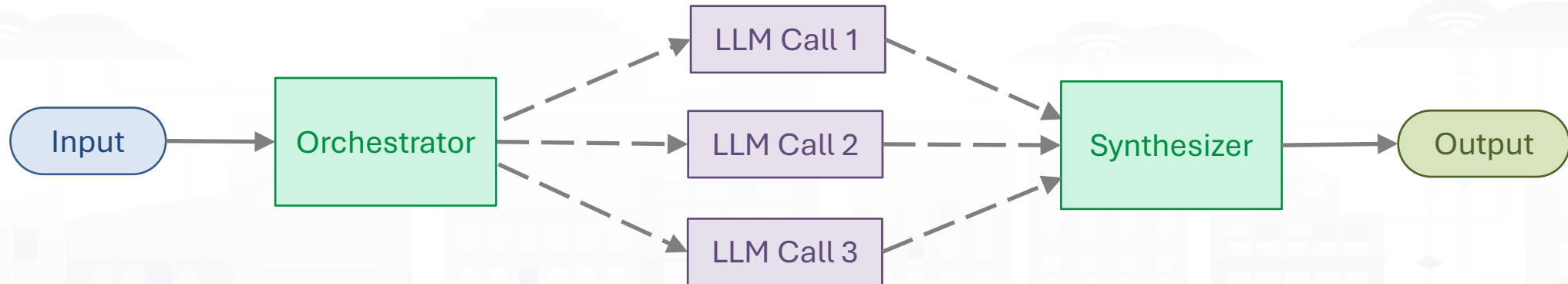
- **Parallelization:** Multiple LLM calls execute concurrently, with outputs aggregated programmatically into a single result. Two distinct variants:
 - **Sectioning:** the task is decomposed into independent subtasks, each assigned to a dedicated LLM call running in parallel (reducing end-to-end latency); the aggregator assembles the complementary parts.
 - **Voting:** the same task is submitted to multiple LLM calls independently, and the aggregator act as arbitrator to select a single output (typically by majority vote or confidence scoring), to increase result reliability.



- Parallelization is suitable when subtasks are mutually independent and can be executed concurrently for latency reduction, or when a single-pass result is insufficient and multiple independent attempts are needed to achieve higher confidence.

Agentic Design Patterns: Orchestrator – Workers

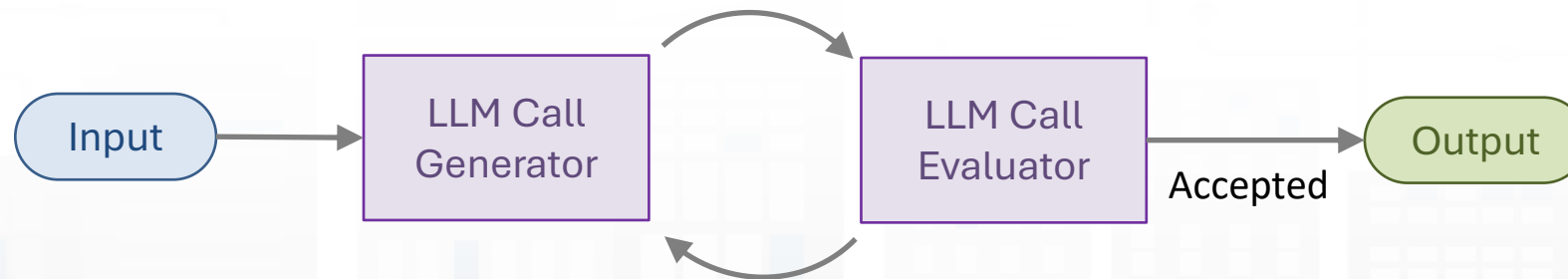
- **Orchestrator – Workers:** A central LLM (the orchestrator) dynamically decomposes the input task, delegates subtasks to specialized worker LLMs, and synthesizes their results into a final output. Unlike Parallelization, the subtask structure is not predefined — it is determined at runtime by the orchestrator based on the specific input.



- Orchestrator – Workers is suitable for complex, open-ended tasks where the number and nature of subtasks depend on the input itself. For instance, a coding task where the set of files to modify and the type of changes required in each file are unknown until the task is analyzed.

Agentic Design Patterns: Evaluator – Optimizer

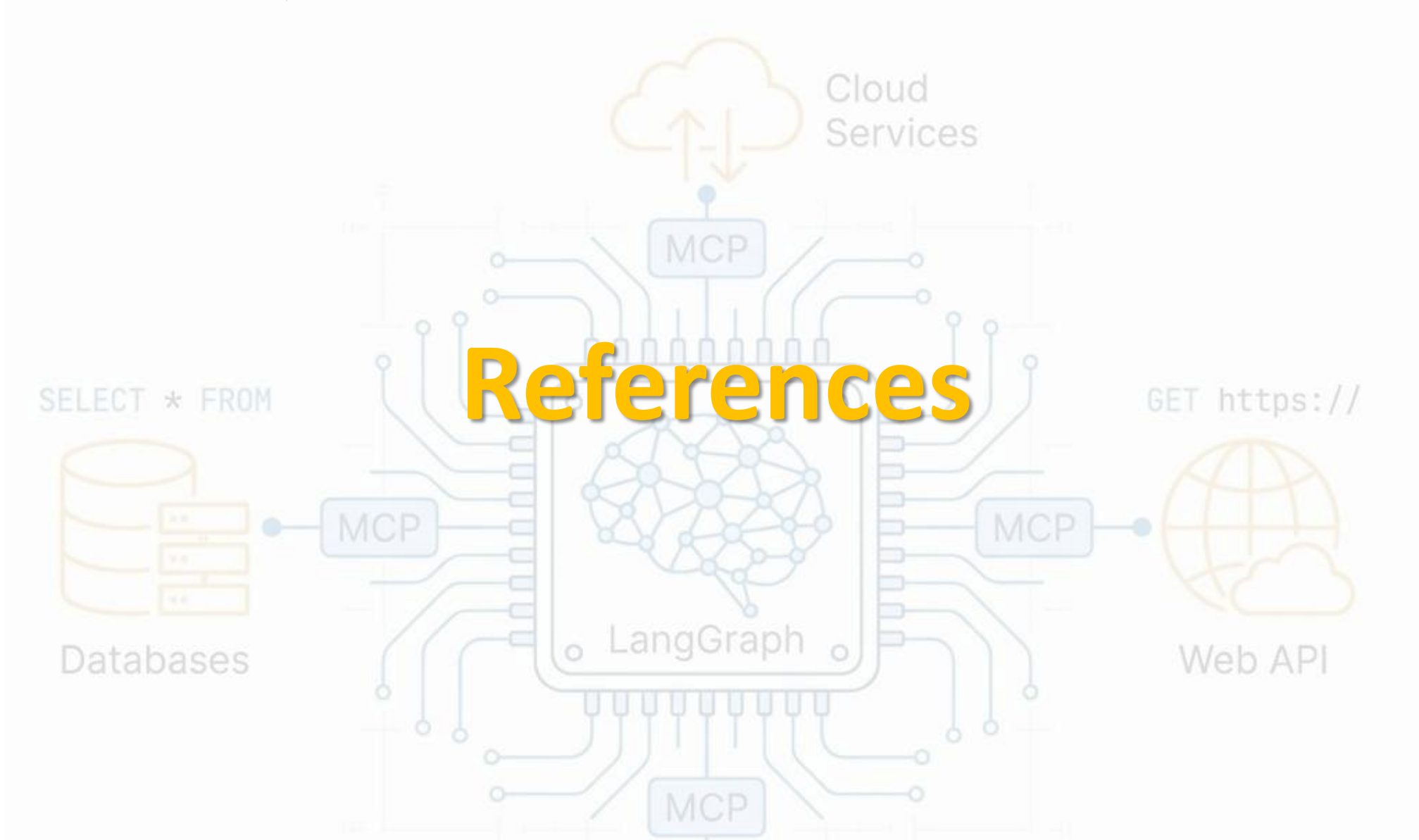
- **Evaluator – Optimizer:** The LLM Call Generator produces a candidate response, the LLM Call Evaluator assesses it against defined criteria and returns a structured feedback. The generator then incorporates the feedback to produce a refined output. The loop repeats until the evaluator determines the output meets the required quality threshold or a maximum number of iterations is reached.



- Evaluator – Optimizer is suitable when there are clear evaluation criteria and an iterative refinement produces measurable quality improvements over successive passes.

Integrated Taxonomy and Design Patterns of LLM-based Agents

Level	Architecture	Workflow	Key Capabilities	Pattern
L1 Simple Tool-Calling Chatbot/Assistant <ul style="list-style-type: none"> Actual 	<ul style="list-style-type: none"> LLM + optional function/tool calling Single-step: Perceive → Decide → Act No planning, no iteration, no memory 	<ul style="list-style-type: none"> Single pass / Fixed workflow 	<ul style="list-style-type: none"> Q&A Chatbot with tools Classification, retrieval, summarization Basic RAG pipelines 	Prompt Chaining / Routing
L2 Reasoning Agent <ul style="list-style-type: none"> Actual 	<ul style="list-style-type: none"> LLM + CoT + interleaved tool use Thought → Action → Observation loop (ReAct) Self-evaluation + retry (Reflexion) 	<ul style="list-style-type: none"> ReAct loop [Yao et al., 2023] Reflexion [Shinn et al., 2023] 	<ul style="list-style-type: none"> Multi-step Q&A reasoning agent, fact verification Code debugging, research tasks Auto-correction from environment feedback 	Evaluator-Optimizer
L3 Goal-Based Planning Agent <ul style="list-style-type: none"> Actual 	<ul style="list-style-type: none"> LLM orchestrator + worker LLMs Plan-and-Execute, Tree/Graph of Thought (GoT) 	<ul style="list-style-type: none"> Plan → Decompose → Dispatch → Sync Parallel tool calls via Direct Acyclic Graph (DAG) - 	<ul style="list-style-type: none"> Complex multi-step workflows Multi-file coding, advanced research Dependency-aware task planning 	Orchestrator-Workers
L4 Multi-Agent Collaborative System <ul style="list-style-type: none"> Actual / Frontier 	<ul style="list-style-type: none"> Multiple specialised agents + central orchestrator Inter-agent communication: A2A Shared memory, distinct roles 	<ul style="list-style-type: none"> Centralised or peer-to-peer coordination Human-in-the-Loop (optional checkpoints) 	<ul style="list-style-type: none"> Cognitive parallelism on heterogeneous tasks Specialization + emergent agent behaviors 	Multi-Agent Collaboration
L5 Self-Improving Lifelong Agent <ul style="list-style-type: none"> Research 	<ul style="list-style-type: none"> Agent that generates tools at runtime Self-improvement at runtime, new skill creation without fine-tuning 	<ul style="list-style-type: none"> Automatic Iterative Prompting Environment feedback → new skill coded & stored 	<ul style="list-style-type: none"> Builds own operational tools at runtime Lifelong learning in open contexts Zero-shot generalization to unseen tasks 	Self-Improving / Self-Tooling



References

- [Anthropic, 2024a]** “Building effective agents”. Online: <https://www.anthropic.com/engineering/building-effective-agents>
- [Anthropic, 2024b]** “Introducing the Model Context Protocol”. Online: <https://www.anthropic.com/news/model-context-protocol>
- [Besta et al., 2024]** Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Gianinazzi, L., Gajda, J., Lehmann, T., Podstawski, M., Niewiadomski, H., Nyczyk, P., & Hoefler, T., “Graph of Thoughts: Solving Elaborate Problems with Large Language Models”, in Proceedings of the AAIL Conference on Artificial Intelligence, 38(16), 17682–17690. arXiv:2308.09687, 2024.
- [Kojima et al., 2022]** Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., & Iwasawa, Y., “Large Language Models are Zero-Shot Reasoners”, in Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS'22). Article No.: 1613, Pages 22199 - 22213. arXiv:2205.11916, 2022.
- [OECD, 2026]** “The Agentic AI Landscape and its Conceptual Foundations”, OECD Artificial Intelligence Papers, No. 5, February 2026. https://www.oecd.org/content/dam/oecd/en/publications/reports/2026/02/the-agentic-ai-landscape-and-its-conceptual-foundations_a9d4b451/396cf758-en.pdf
- [Russell & Norvig, 1995]** Russell, S. J., & Norvig, P. (1995). Artificial Intelligence: A Modern Approach. Prentice Hall
- [Shinn et al., 2023]** Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S., “Reflexion: Language Agents with Verbal Reinforcement Learning”. In Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS 2023). arXiv:2303.11366, 2023.
- [Wei et al., 2022]** Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D., “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”, in Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS'22). Article No.: 1800, Pages 24824 – 24837, 2022. arXiv:2201.11903, 2022.

References

[Wooldridge & Jennings, 1995] Wooldridge, M. and Jennings, N. R., “Intelligent agents: theory and practice”, The Knowledge Engineering Review, Vol. 10:2,115-152, 1995.

[Yao et al., 2023a] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y., & Narasimhan, K., “Tree of Thoughts: Deliberate Problem Solving with Large Language Models”, in Proceedings of the 37th International Conference on Neural Information Processing Systems (NIPS’23). Article No.: 517, Pages 11809 – 11822. arXiv:2305.10601, 2023.

[Yao et al., 2023b] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y., “ReAct: Synergizing reasoning and acting in language models”, in Proceedings of the 11th International Conference on Learning Representations (ICLR 2023). arXiv preprint arXiv:2210.03629v3, 2023.