Bayes Business School | February 5th 2025

Beyond Chatbots: Financial Innovation and Data Analysis with Agentic LLMs

Nicole Koenigstein

About me

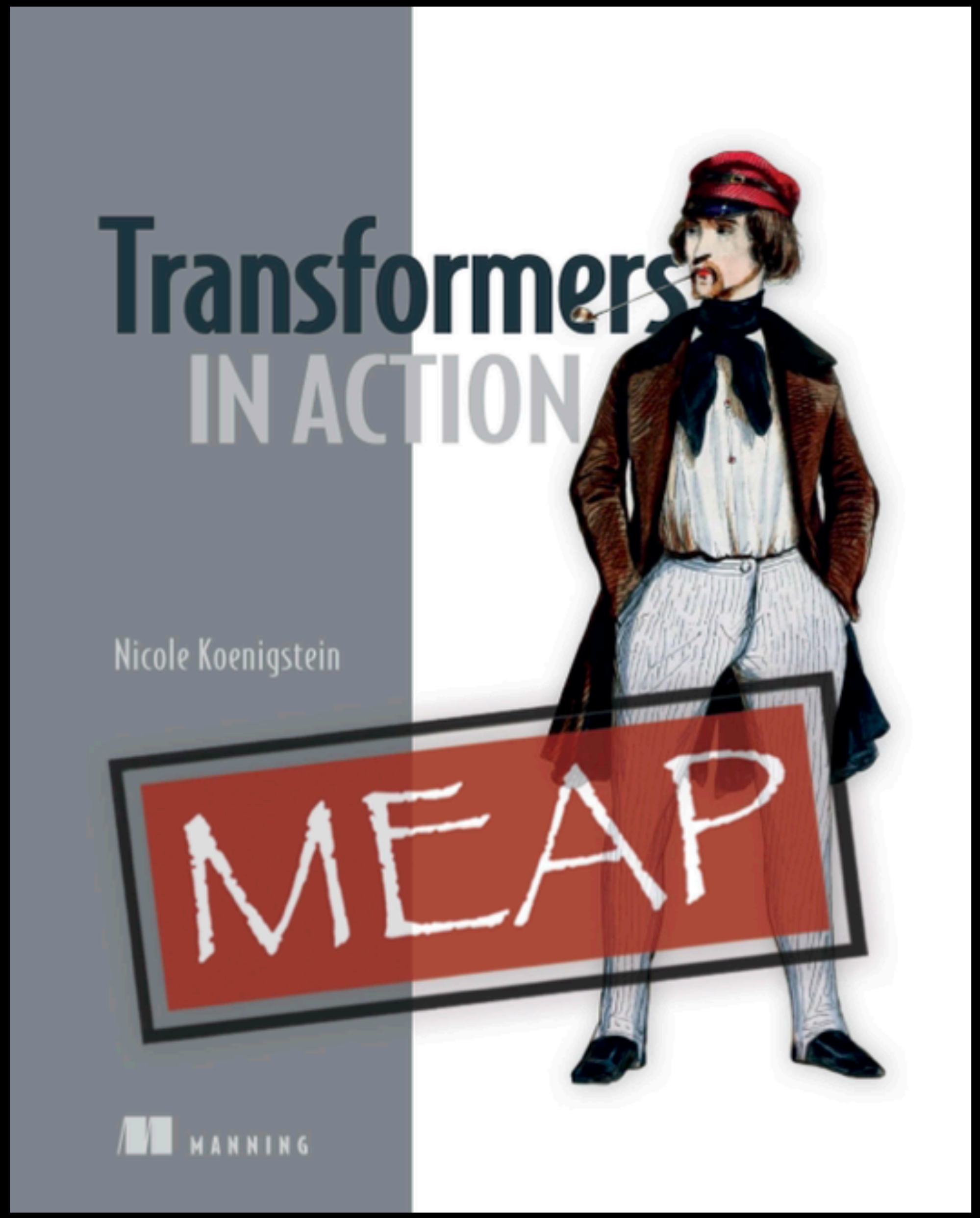
Digital Finance

Smart Data Analytics, Investment Innovation, and Financial Technology

Springer

Volume 5, Issue 3-4

December 2023


Transformers IN ACTION

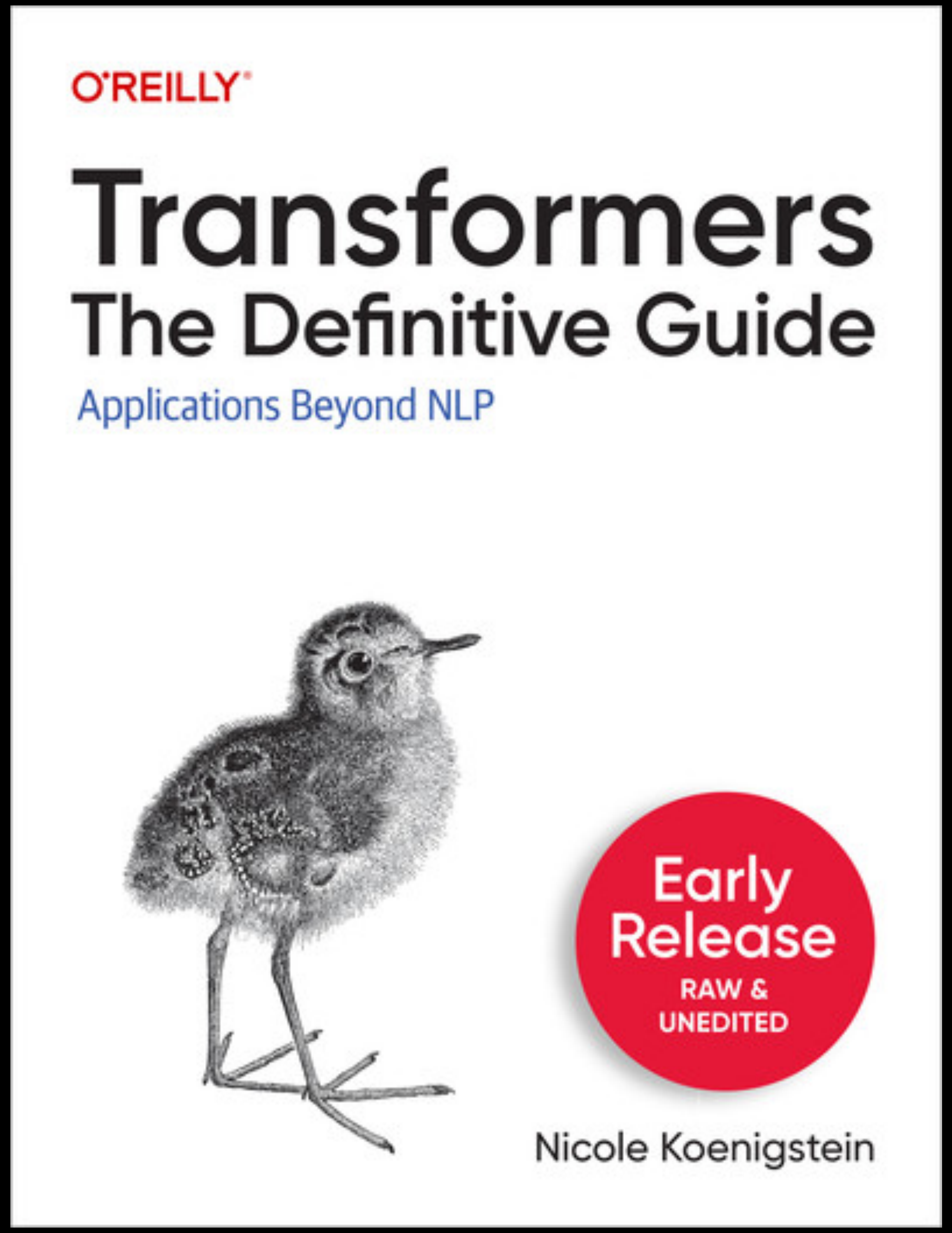
Nicole Koenigstein

MEAP

MANNING

All chapters online, Print Spring 2025

https://www.manning.com/books/transformers-in-action


O'REILLY

Transformers The Definitive Guide

Applications Beyond NLP

Early Release RAW & UNEDITED

Nicole Koenigstein

Online available March 2025, Print Summer 2025

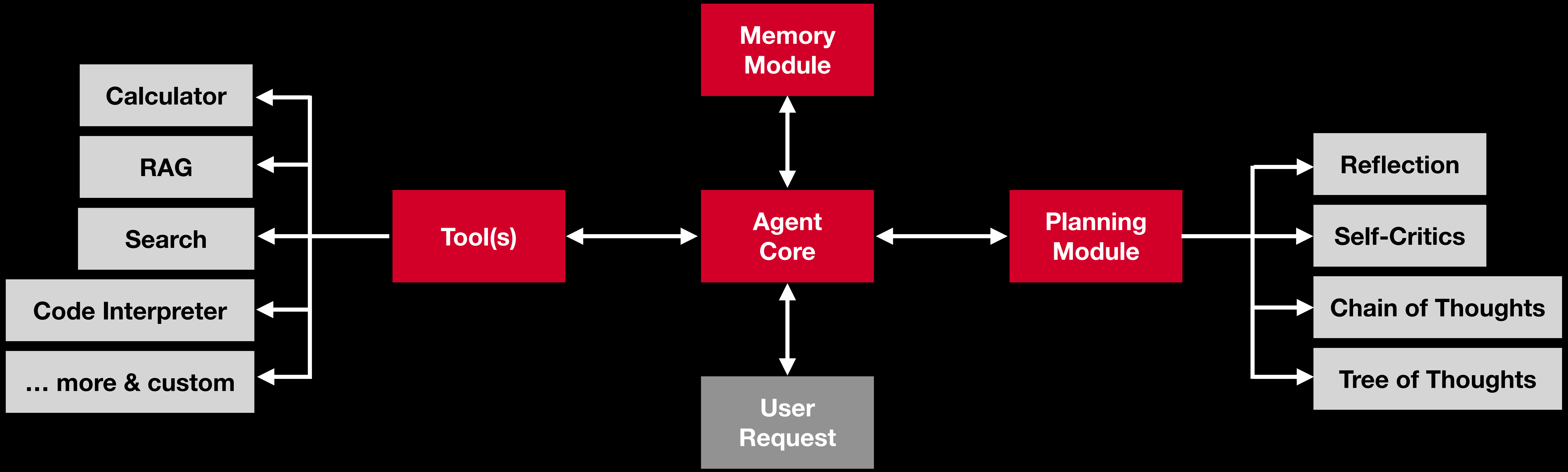
Connect with me

GitHub Repo

LLM Agents

LLM Agents

Agents with an LLM as its Core Controller



In a LLM-powered autonomous agent system, an LLM functions as the agent's brain, complemented by several key components. An agent, in this context, is something that acts or has the capacity to act. Through its components, actively engaging in processing information, making predictions, and providing explanations.

LLM Agents Examples

- Single Agents
- ReAct Agents
- Multi-Agents
- Reflection Agents
- Multi-modal Agents

Single Step vs. ReAct Agent

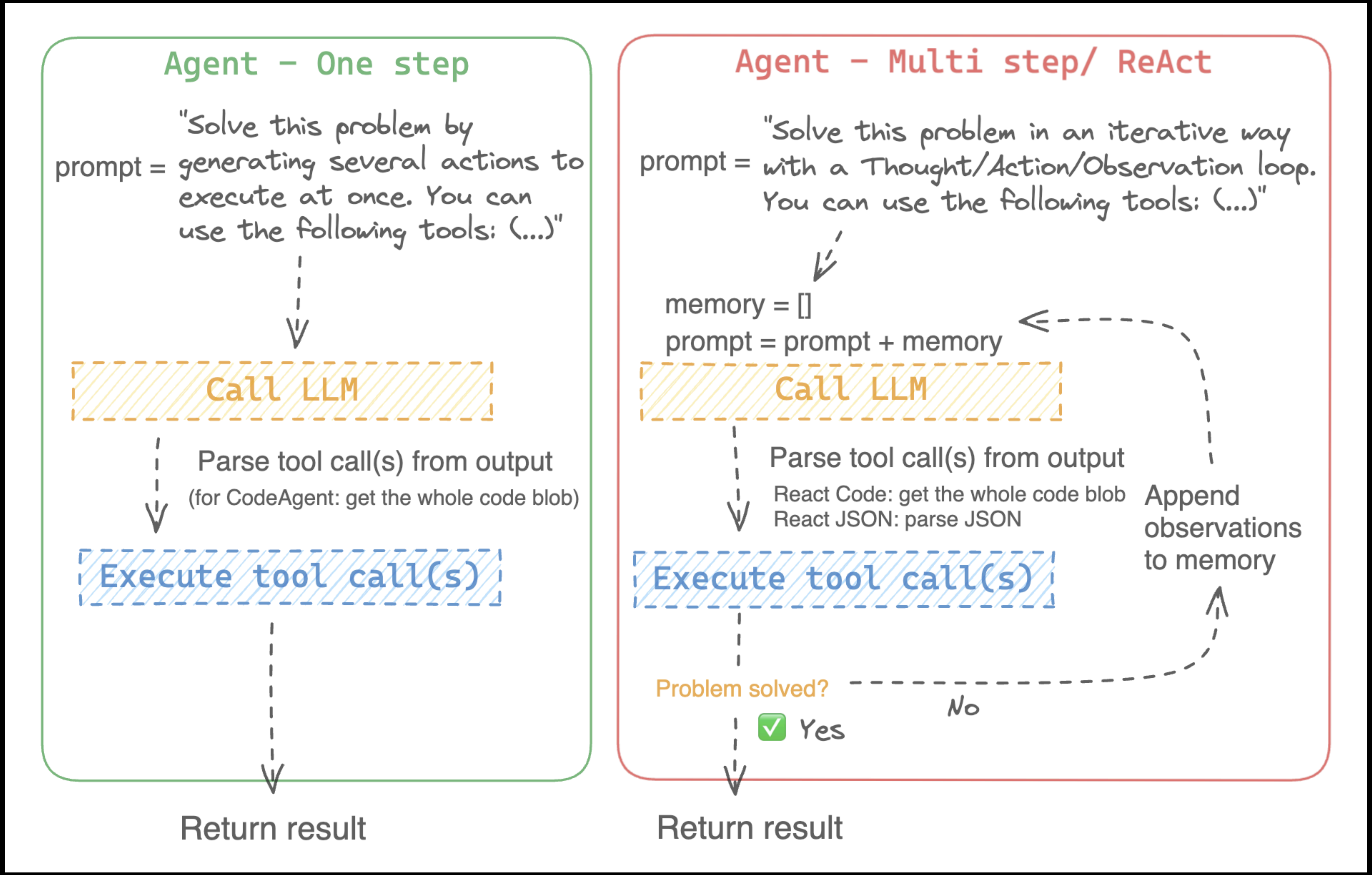

Agent - One step
prompt = "Solve this problem by generating several actions to execute at once. You can use the following tools: (...)"
Call LLM
Parse tool call(s) from output
(for CodeAgent: get the whole code blob)
Execute tool call(s)
Return result
Agent - Multi step/ ReAct
prompt = "Solve this problem in an iterative way with a Thought/Action/Observation loop. You can use the following tools: (...)"
memory = []
prompt = prompt + memory
Call LLM
Parse tool call(s) from output
React Code: get the whole code blob
React JSON: parse JSON
Execute tool call(s)
Append observations to memory
Problem solved?
No
✅ Yes
Return result

Image-Source: LangChain

ReAct: Reasoning Iteratively Over Context

Mapping contexts c_t to actions a_t with action space as $\hat{A} = A \cup L$.

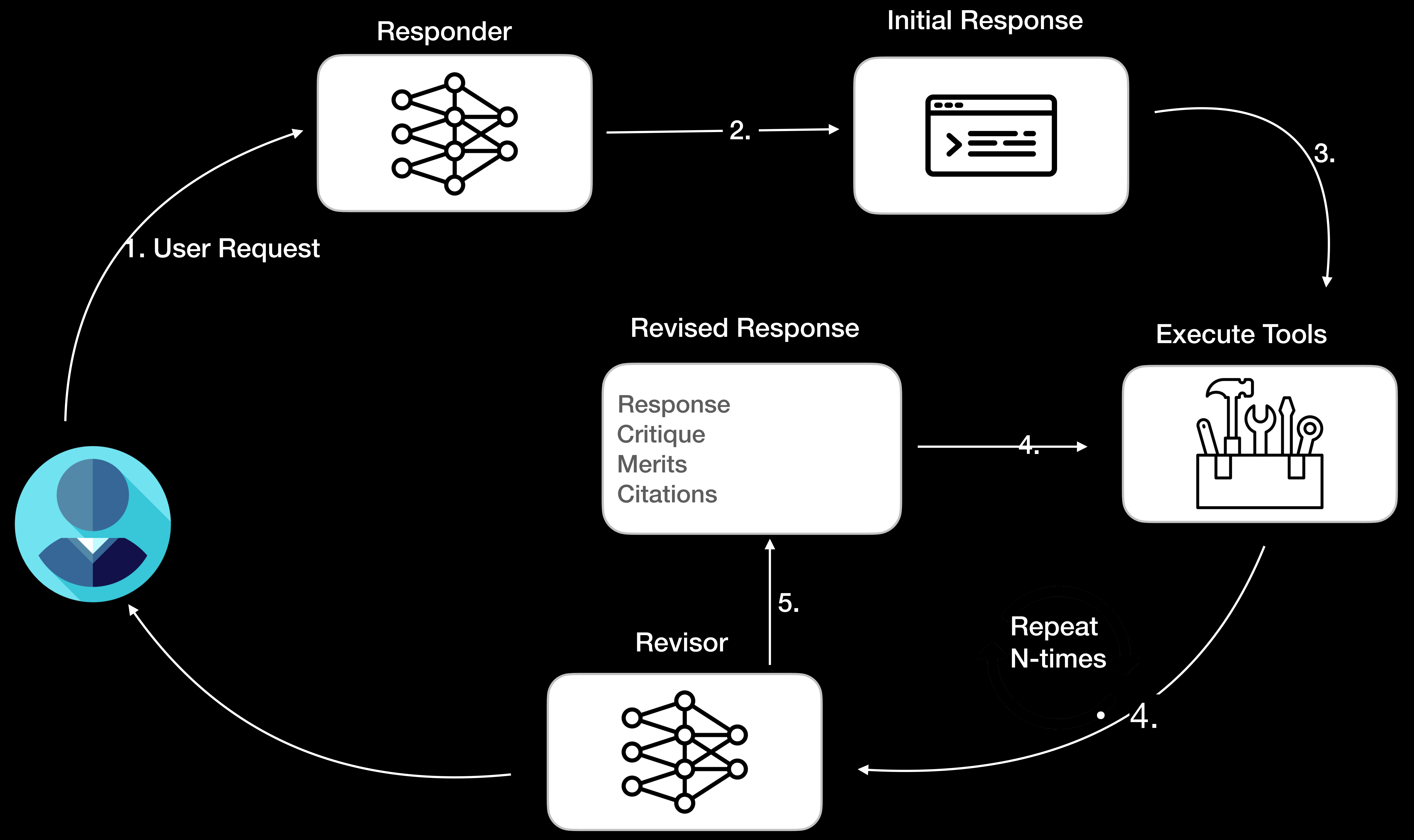
Policy $\pi(a_t \,|\, c_t)$ where $c_t = (o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t)$

Thoughts ($\hat{a}_t$) are integrated for contextual updates as $c_{t+1} = (c_t, \hat{a}_t)$

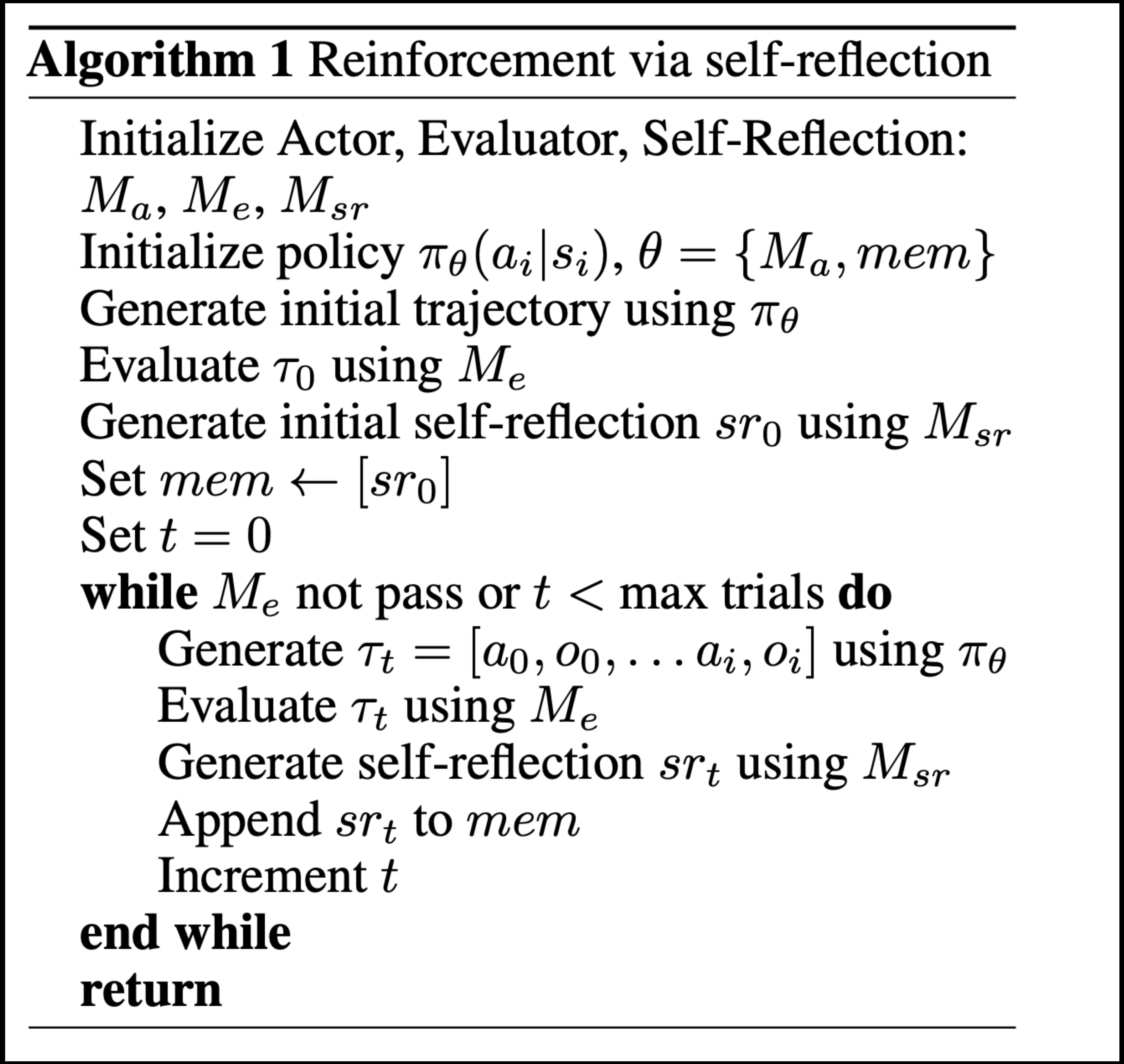
Actions (A): Task-specific steps impacting the environment.
Reasoning Traces (L**)**: Language-based reasoning, or "thoughts," updating

Paper: ReACT: Snyergizing Reasoning and Acting in Language Models; https://arxiv.org/abs/2210.03629

Reflection Agent



Reflexion: Reinforcement via Verbal Reflection



Algorithm 1 Reinforcement via self-reflection

```
Initialize Actor, Evaluator, Self-Reflection:
$M_a, M_e, M_{sr}$
Initialize policy $\pi_\theta(a_i|s_i), \theta = \{M_a, mem\}$
Generate initial trajectory using $\pi_\theta$
Evaluate $\tau_0$ using $M_e$
Generate initial self-reflection $sr_0$ using $M_{sr}$
Set $mem \leftarrow [sr_0]$
Set $t = 0$
while $M_e$ not pass or $t <$ max trials do
    Generate $\tau_t = [a_0, o_0, \dots a_i, o_i]$ using $\pi_\theta$
    Evaluate $\tau_t$ using $M_e$
    Generate self-reflection $sr_t$ using $M_{sr}$
    Append $sr_t$ to $mem$
    Increment $t$
end while
return
```

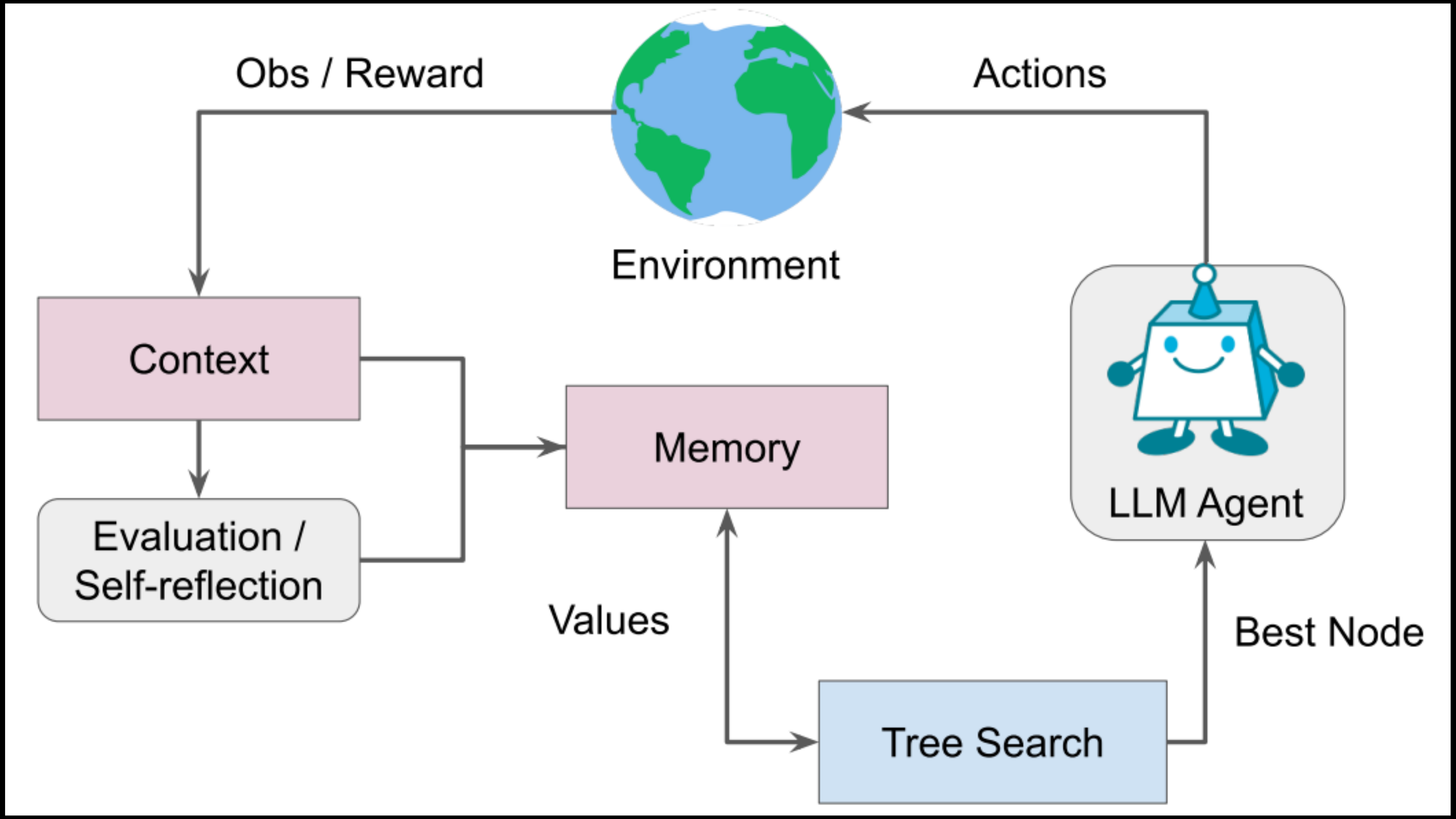
- **Actor (M_a)**: Generates actions based on the environment's state using a language model, guided by both current observations and memory of past reflections.

- **Evaluator (M_e)**: Assesses the Actor's actions with task-specific criteria (e.g., success/failure or heuristic checks) and produces a reward score for feedback.

- **Self-Reflection Model (M_{sr})**: Analyzes actions and produces verbal feedback stored in memory, providing context for future decisions and helping the Actor improve.

Paper: Reflexion: Language Agents with Verbal Reinforcement Learning; https://arxiv.org/abs/2303.11366

Language Agent Tree Search (LATS)



Overview of Architecture.

Paper: Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models; https://arxiv.org/pdf/2310.04406

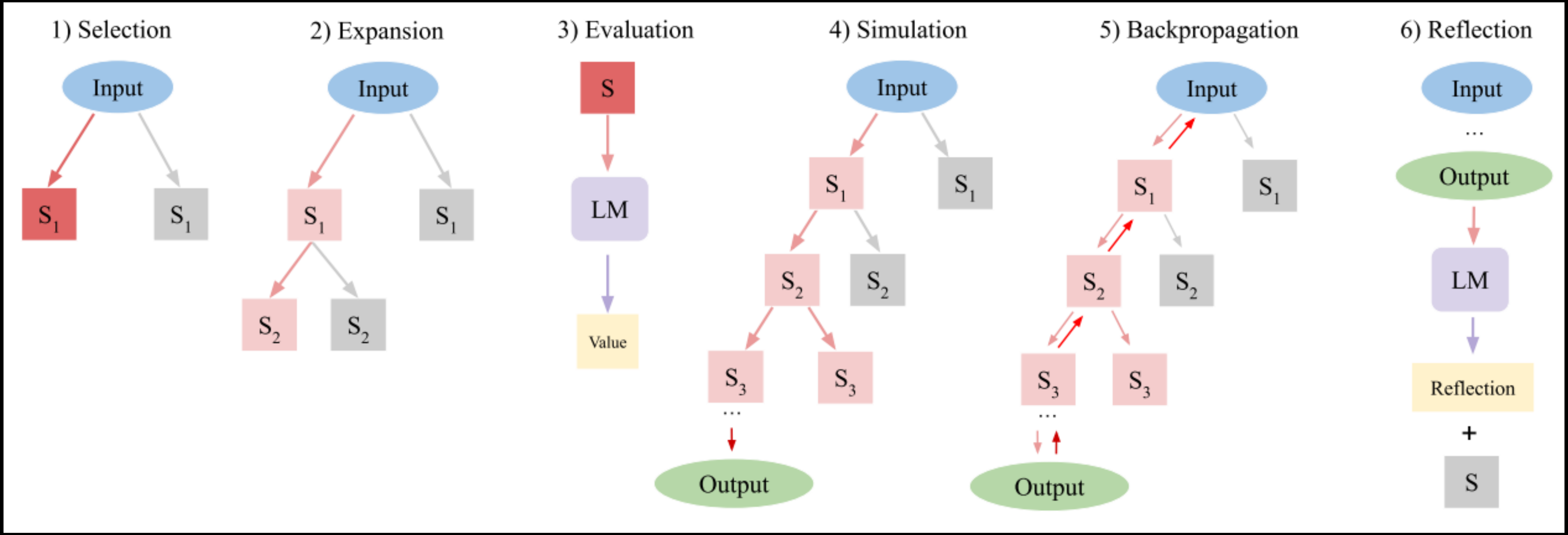
Language Agent Tree Search

First Framework Incorporating Reasoning, Acting, and Planning to Enhance LM Performance. LATS integrates Monte Carlo Tree Search (MCTS) with LMs to unify reasoning, acting, and planning.

Key Features:

- LM-based value functions (replaces traditional RL-trained evaluators, leveraging in-context learning)
- Self-reflections for exploration
- External feedback for adaptive problem-solving

Language Agent Tree Search



Overview of the six operations in LATS: A node is selected, expanded, evaluated, and simulated until a terminal node is reached, after which the value is backpropagated. If the trajectory fails, a reflection is generated for future trials. This process repeats until the budget is exhausted or the task succeeds.

Monte Carlo Tree Search

- **Selection**: Start from the root node and traverse the tree to select a node based on the Upper Confidence Bound applied to Trees (UCT).
- **Expansion**: If the selected node is not terminal, expand by generating child nodes from possible actions.
- **Simulation**: Run a random simulation from the new node to estimate a value.

Monte Carlo Tree Search

- **Backpropagation**: Propagate the simulation results back up the tree, updating node values along the way.

- **Reflection**: When a trajectory fails, use feedback or self-reflection to adjust future decisions, refining the search by avoiding repeated mistakes and improving strategy.

Monte Carlo Tree Search: Explore-exploit Tradeoff

Explore-exploit tradeoff is the tradeoff between gathering new information (exploration) and using that information to improve performance (exploitation).

Upper Confidence bounds Applied to Trees (UCT)

The value for expansion is selected by the next iteration. The UCT of a child state s is calculated as follows:

$$\text{UCT}(s) = V(s) + c\sqrt{\frac{\ln N(p)}{N(s)}},$$

where:

- $V(s)$: Value estimate of node s
- $N(s)$: Visit count of node s
- $N(p)$: Visit count of the parent node
- c: Exploration weight

Backpropagation in LATS

The return r is used for updating every $V(s)$ along the path with the formula

$$V(s) = \frac{V_{old}(s)(N(s) - 1) + r}{N(s)},$$

where $V_{old}(s)$ is the old value function.

Intro to LangGraph

LangGraph

Nodes Do the Work. Edges Tell What to Do Next.

- **State**: A shared data structure that represents the current snapshot of an application. It can be any Python type, but is typically a TypedDict or Pydantic BaseModel.
- **Nodes**: Functions that encode the logic of the agents. They receive the current state as input, perform some computation or side-effect, and return an updated state.
- **Edges**: Functions that determine which node to execute next based on the current state. They can be conditional branches or fixed transitions.
- **Command**: Combines control flow (edges) and state updates (nodes). For example, a node can update state and decide the next node to visit. LangGraph enables this by returning a Command object from node functions.

Command in LangGraph

```python
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    return Command(
        # state update
        update={"foo": "bar"},
        # control flow
        goto="my_other_node"
    )
```

LangGraph

- **Handoffs**: In multi-agent architectures, agents are represented as graph nodes. Each node executes steps and decides whether to terminate or route to another agent, including itself (e.g., loops). A common pattern is handoffs, where one agent transfers control to another, specifying:
  - **Destination**: Target agent (node name).
  - **Payload**: Information passed (e.g. state update).

Hand-offs in LangGraph

```
def agent(state) -> Command[Literal["agent", "another_agent"]]:
    # the condition for routing/halting can be anything, e.g. LLM tool call /
structured output, etc.
    goto = get_next_agent(...)  # 'agent' / 'another_agent'
    return Command(
        # Specify which agent to call next
        goto=goto,
        # Update the graph state
        update={"my_state_key": "my_state_value"}
    )
```

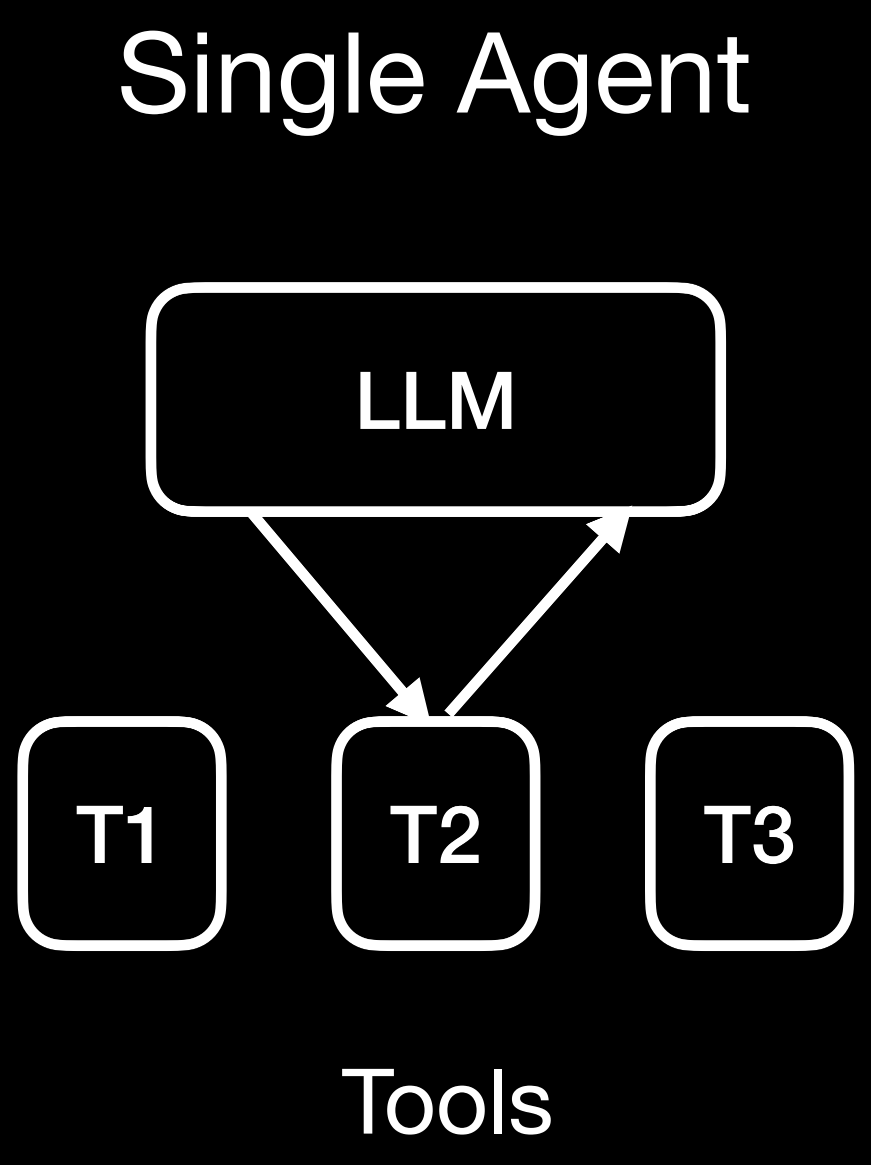
LangGraph

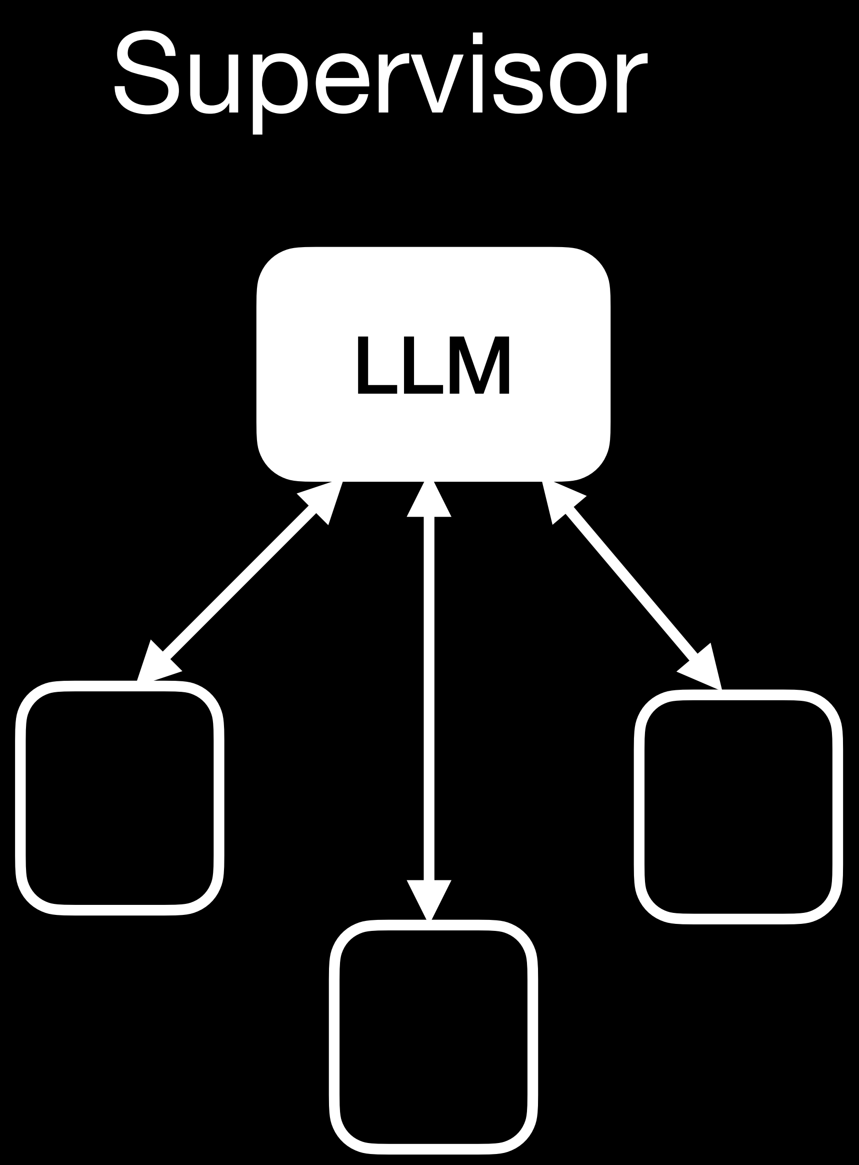
Multi-agent Workflow

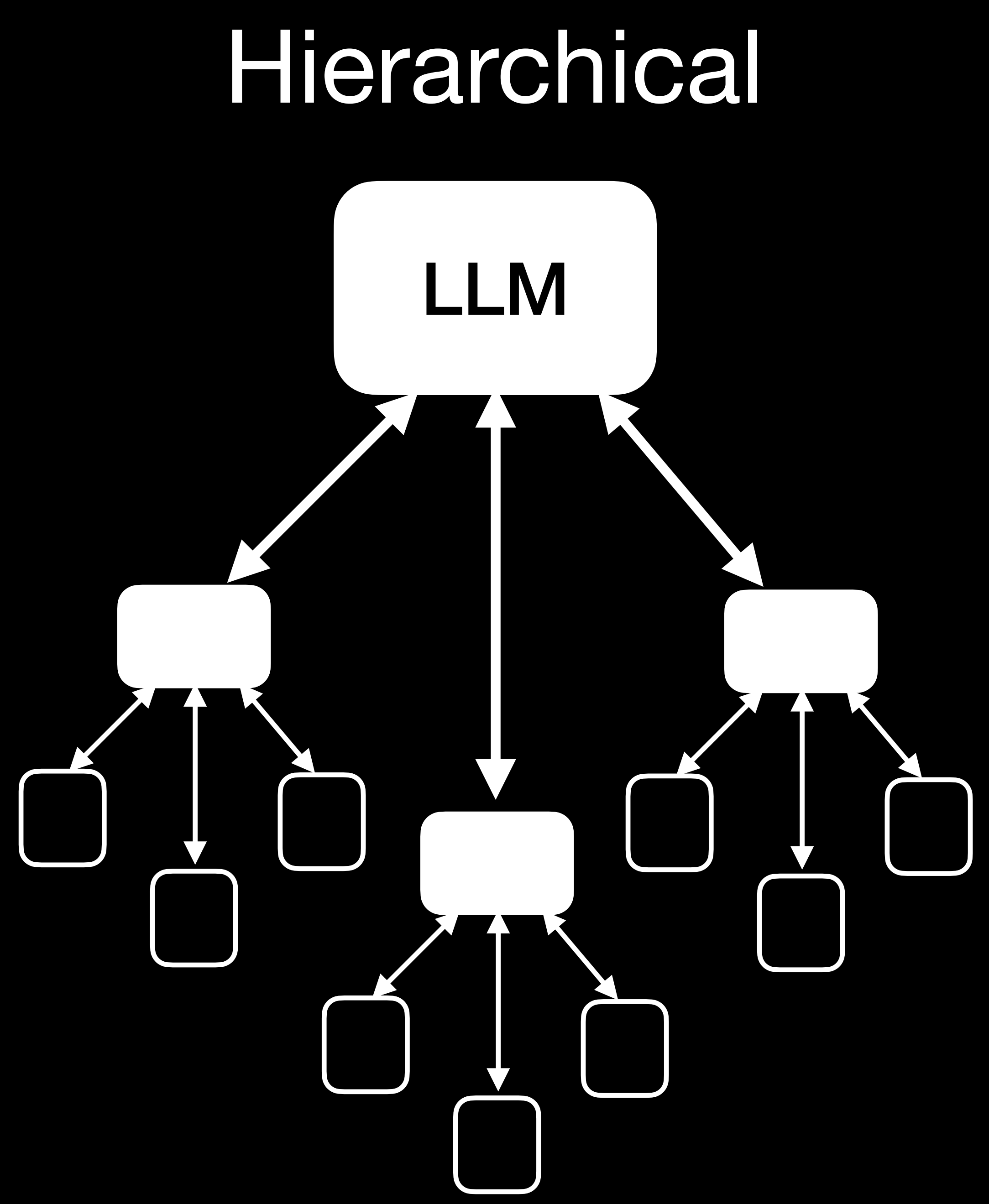
- **Explicit Control Flow (Normal Edges)**: LangGraph allows you to define the application's control flow explicitly via graph edges, ensuring a deterministic sequence where the next agent is known in advance.

- **Dynamic Control Flow (Command)**: LangGraph enables LLMs to decide parts of the control flow using Command. A special case is the supervisor tool-calling architecture, where the supervisor agent's tool-calling LLM determines the order in which tools (agents) are invoked.

Example Multi-Agent Architectures

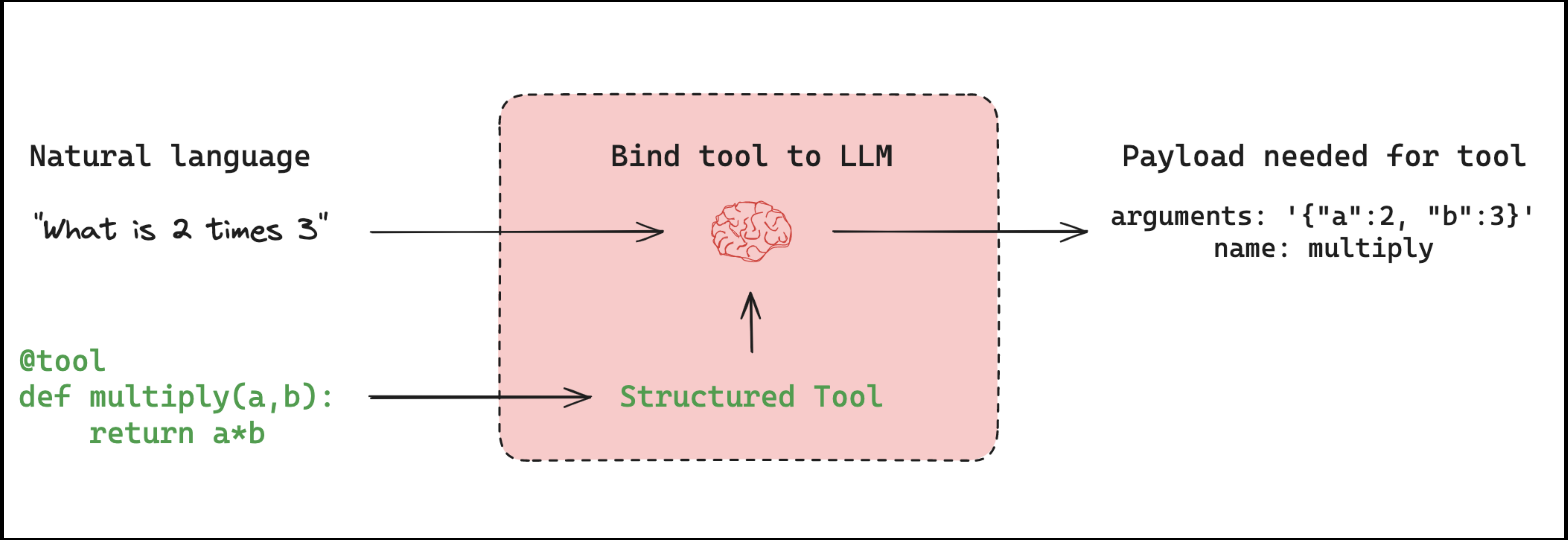






In a supervisor architecture, each agent communicates with a single supervisor agent, which makes decisions on which agent should be called next. In contrast, a hierarchical architecture features a multi-agent system with a supervisor of supervisors. This generalizes the supervisor architecture and allows for more complex control flows.

Tool Calling with LangGraph

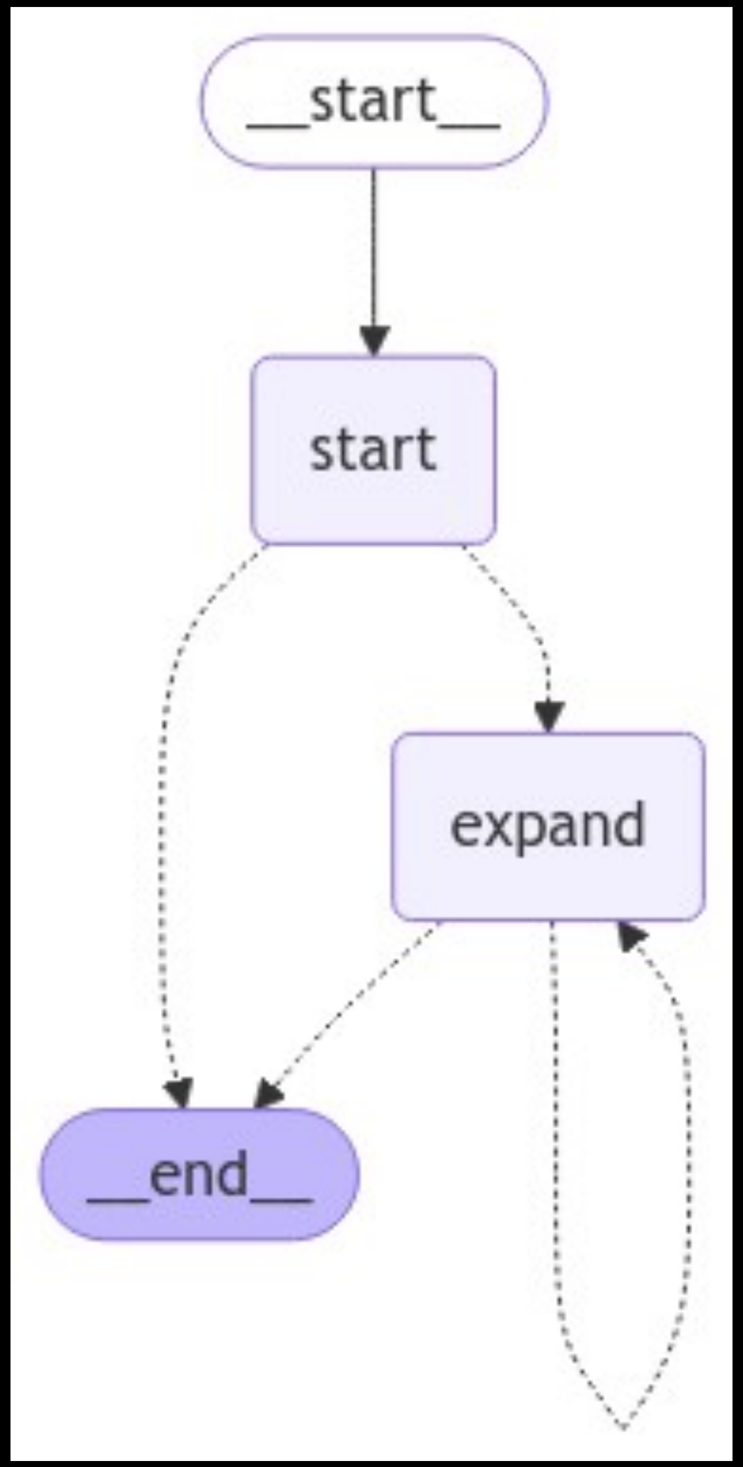


```
@tool
def multiply(a,b):
    return a*b
```

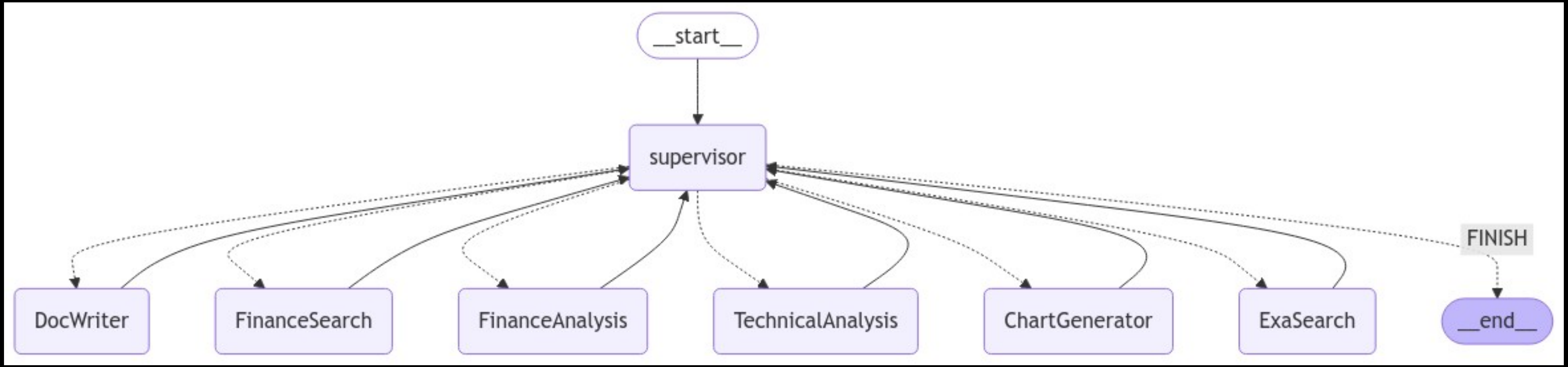
Image-Source: LangChain

Hands-on:
Code Walkthrough

LATS in LangGraph



LATS in LangGraph



Supervisor Architecture for Asset Analysis

Overall Workflow

- The Supervisor Agent assigns tasks to the appropriate agents.
- The Ticker Search Agent looks up the correct symbol for yahoo finance for a given asset.
- The News Search Agent generates first search queries for a given asset and then gathers news data on that asset.
- The Sentiment Agent classifies the news sentiment for a given asset.
- The Expert Technical Analyst Agent conducts a technical analysis of a given asset.
- The Quant Developer Agent writes code to plot any charts requested of a given asset.
- The Chief Investment Strategist synthesizes all analyses to create a definitive investment report on a given asset.

Tools & Frameworks

LangChain

LangGraph

LangSmith

SerpApi Google

Supervisor Architecture for Asset Analysis

Possible Extensions:

- Extending to collecting internal financial information on the asset with RAG.
- A Review and Editing Agent reviews and refines the document for final submission.

Challenges and Limitations

Challenges and Limitations

- Accuracy
- Hallucinations
- Limited Context
- Adapting to Specific Roles
- Prompt Dependence
- Managing Knowledge

Accuracy

LLM agents rely, for instance, on the retrieval augmented generation (RAG) system for information, but even if the accuracy for the retrieval is high, LLMs still generate the next token based on probability.

Hallucinations

LLM hallucinations occur when large language models generate inaccurate or false information that appears plausible but lacks factual basis. These errors arise because the model predicts responses based on patterns in the data it was trained on, rather than verifying the truth of the information.

Limited Context

LLM agents have limited memory and may forget key details from earlier in a conversation or miss important instructions. While vector stores help by offering more information and memory, they still don't fully solve this limitation.

Adapting to Specific Roles

LLM agents must adapt to different roles based on the task, but fine-tuning them for rare roles or aligning with varied human values is a difficult challenge and not all can be solved with prompt engineering.

Prompt Dependence

LLM agents rely on precise prompts, and even minor changes can cause significant errors. Crafting and refining these prompts is a delicate process.

Managing Knowledge

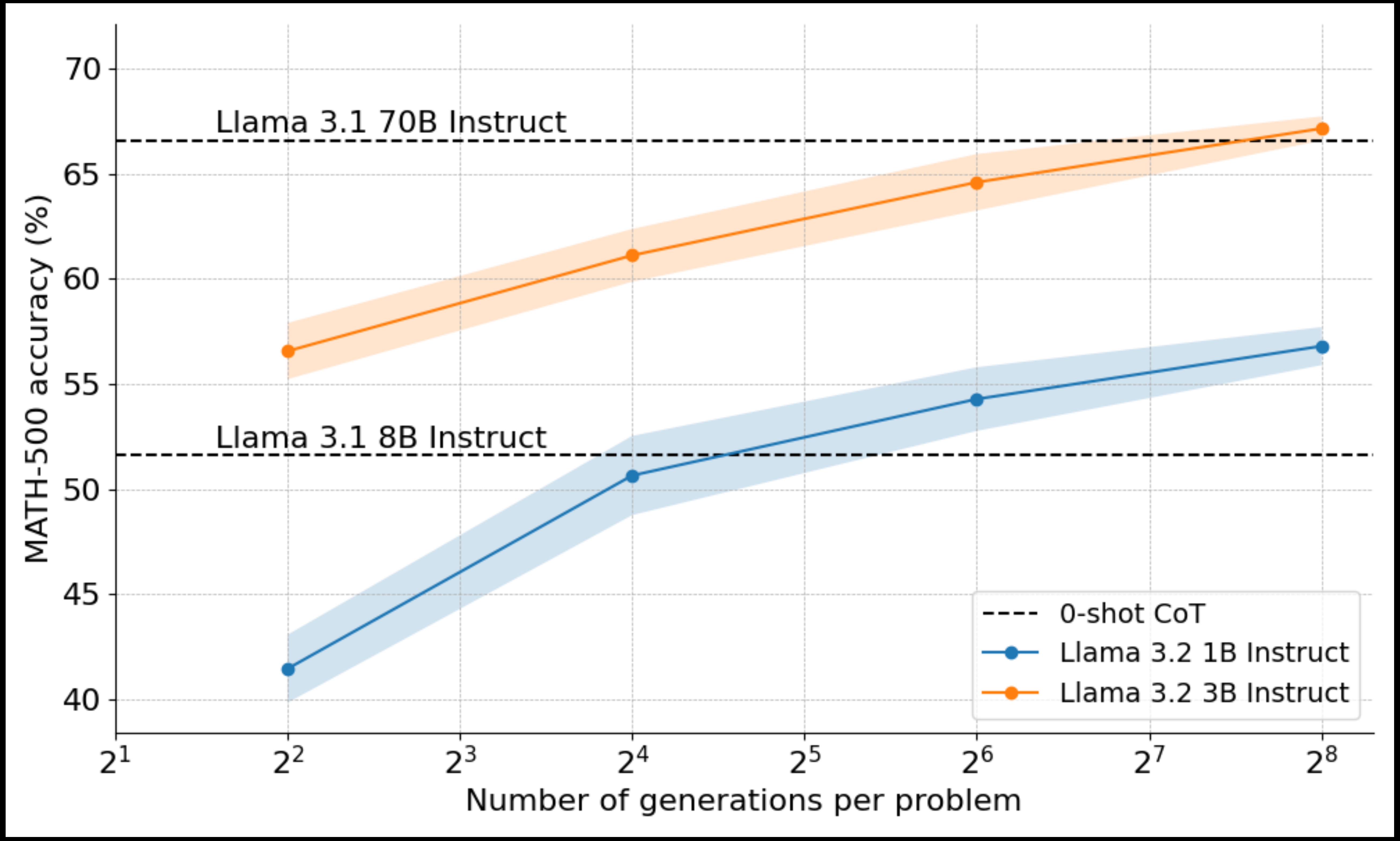
Maintaining an LLM agent's knowledge accuracy and impartiality is challenging. Too much irrelevant data can lead to incorrect conclusions, while outdated facts may result in poor decisions.

Improving Reasoning in Agents and LLMs

Reinforcement Learning and scaling test-time compute can improve the reasoning capabilities in LLMs. Recent papers are:

- RL + Transformer = A General-Purpose Problem Solver
- Critique Fine-Tuning: Learning to Critique is More Effective than Learning to Imitate
- Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters

Improving Reasoning in Agents and LLMs



Tiny 1B and 3B Llama Instruct models outperform their much larger 8B and 70B siblings on the challenging MATH-500 benchmark if you give them enough "time to think".

Image-Source: HuggingFace

Questions