UNIVERSITY OF
OXFORD

Agentic Workflows: Design and Implementation | May 21st 2025

Multi-Agent-Workflow for Investment Analysis

Nicole Koenigstein

Content

- Intro to LLM Agents and Agent Autonomy
- Intro to LangGraph
- Human in the Loop (HITL)
- Advanced LLM Agents
- Test-time compute
- Tree-search

Code Repository

SCAN ME

LLM Agents & Autonomy

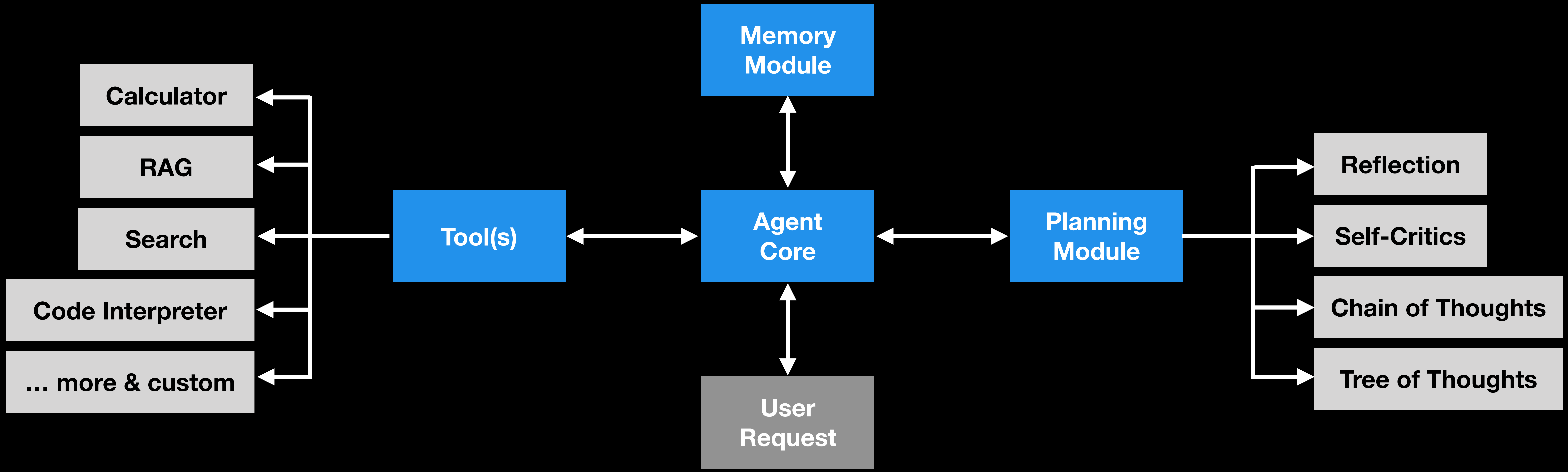
Traditional Workflows vs. Agentic Systems

	Traditional Workflow	Agentic System
Flow	Predefined sequence	Dynamic decision loop
Flexibility	Limited	Context-aware, tool-using
Goal Handling	Linear execution	Iterative, self-correcting

Traditional workflows are static: data → logic → output. Agentic systems are dynamic: they decide what to do, how to do it, and when to adapt in real time.

LLM Agents

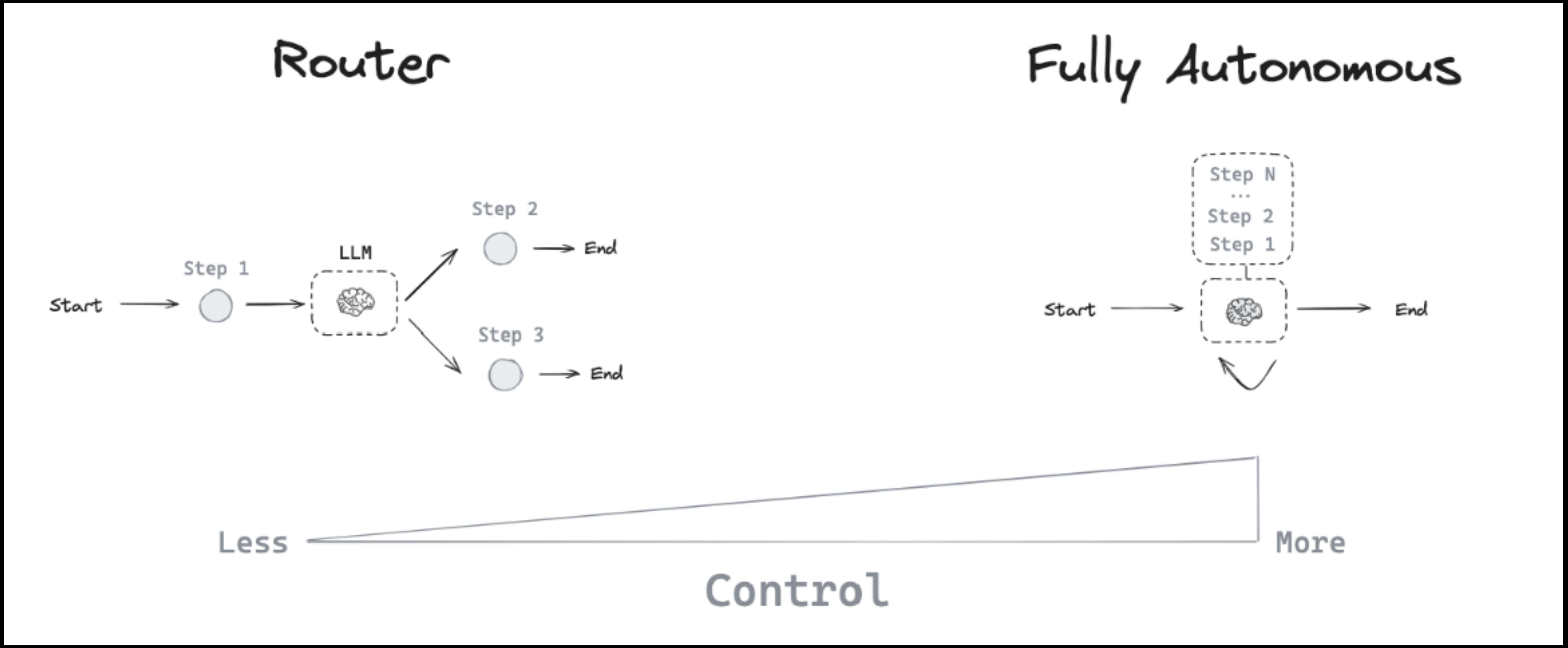
Agents with an LLM as its Core Controller



In a LLM-powered 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

How do we achieve autonomy?



A router allows an LLM to select a single step from a specified set of options. This is an agent architecture that exhibits a relatively limited level of control because the LLM usually focuses on making a single decision and produces a specific output from a limited set of pre-defined options. Routers typically employ a few different concepts to achieve this.

Image-Source: LangGraph

How Can an LLM Decide?

- An LLM can route between two potential paths
- An LLM can decide which of many tools to call
- An LLM can decide whether the generated answer is sufficient or more work is needed

Autonomy: What is possibly Right Now?

We can simulate aspects of autonomy using LangGraph or similar frameworks for:

- Modular, orchestrated workflows with LLM-driven state transitions.
- Conditional routing (e.g. “If reflection fails, go to retry agent”).
- Some limited adaptability by letting the LLM “choose” paths.
- Selecting tools or agents.

Autonomy: What is possibly Right Now?

We can simulate aspects of autonomy using LangGraph or similar frameworks for:

- An agent can revise its own prompt or toolset.
- Can evaluate previous outputs and adjust behavior.
- Some even write and run code to generate their next steps (e.g. "write new plan and inject it").

Autonomy: What We Cannot Do Yet

- Rewire their control graph in a grounded way.
- Generate and validate new LangGraph topologies dynamically and robustly.
- Self-assess tool effectiveness and prune or retrain them continuously.
- Construct and adapt hierarchical goals that persist across sessions with long-term memory.
- Guarantee alignment and coherence when changing strategies mid-execution.

We’re in the phase of constrained emergent autonomy.

Agents can simulate flexibility within frameworks such as LangGraph using reflection, planning, and structured outputs. However, we do not yet have systems that can freely redefine their own architecture or reason about their goals in a robust, general way.

Autonomy: So What Can Be Done?

- Use LangGraph with structured reflection to let agents write new nodes or inject new subgraphs (think: a planning agent generates JSON configs that describe new paths).
- Create a tool like edit_graph or add_tool via validated schema calls, coupled with memory tracking for rollback safety.
- Implement a supervisor that enforces coherence checks on any self-generated graph changes.

State Machines in LLM Agents (LangGraph Context)

State Machine = A system where:

- The agent exists in a defined state
- Based on input + logic, it transitions to the next state
- Each state has an associated function or behavior

In LangGraph:

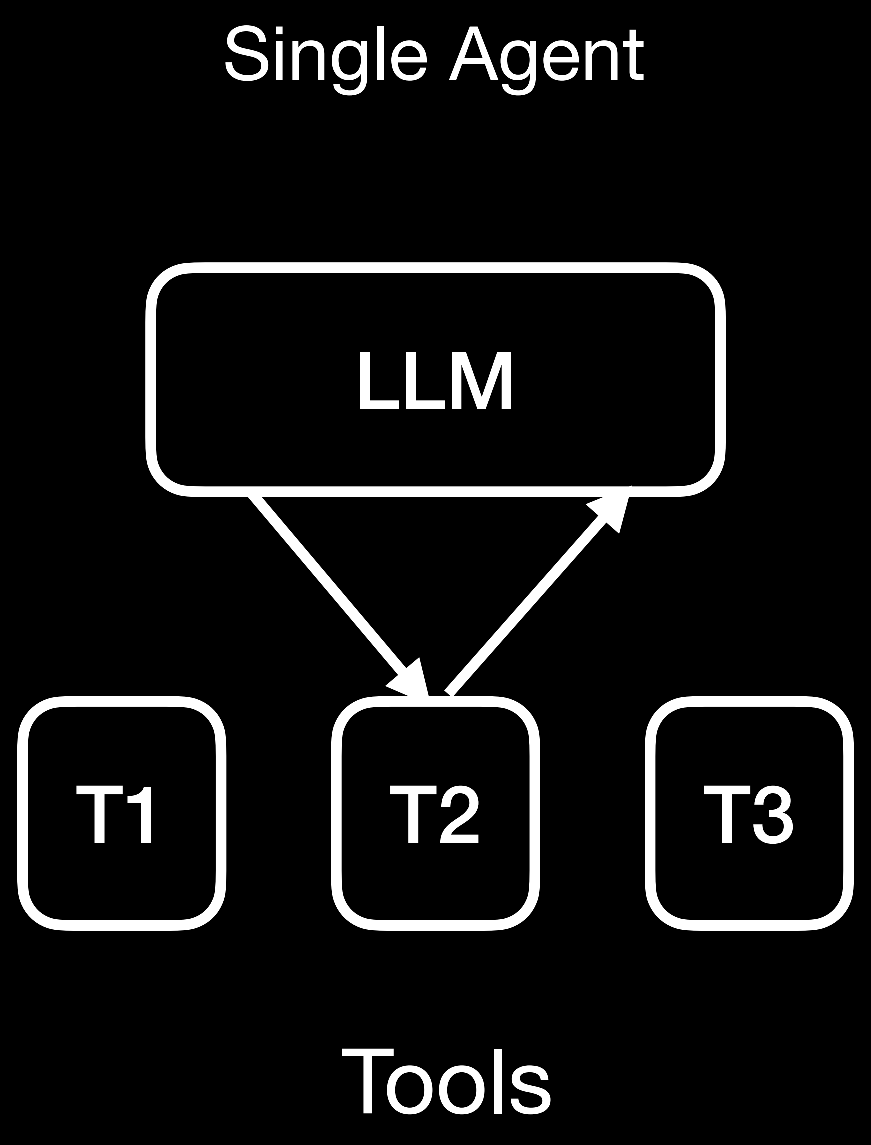
- Nodes = actions or agents
- Edges = transitions, conditional on LLM output
- State object = carries memory, inputs, and metadata

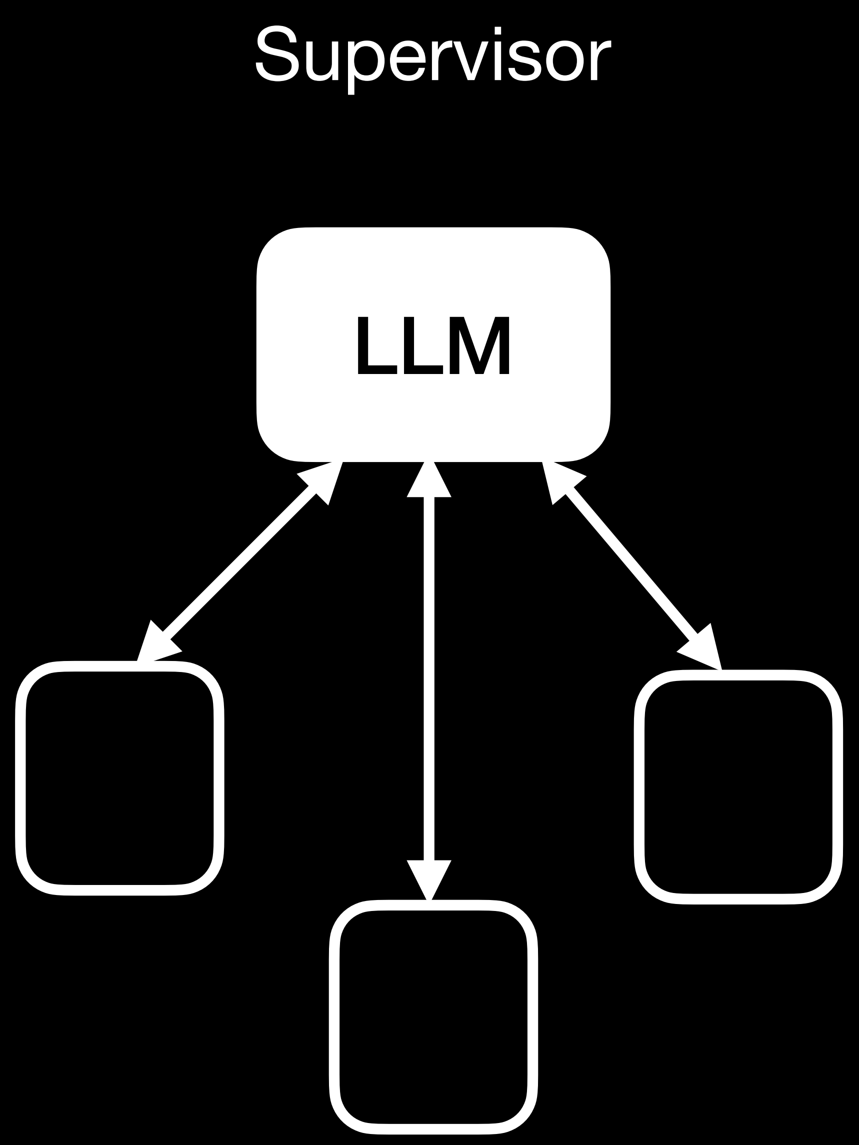
Supervisor-based agent architectures use state machines to coordinate tool use, reflection, and planning within bounded autonomous flows.

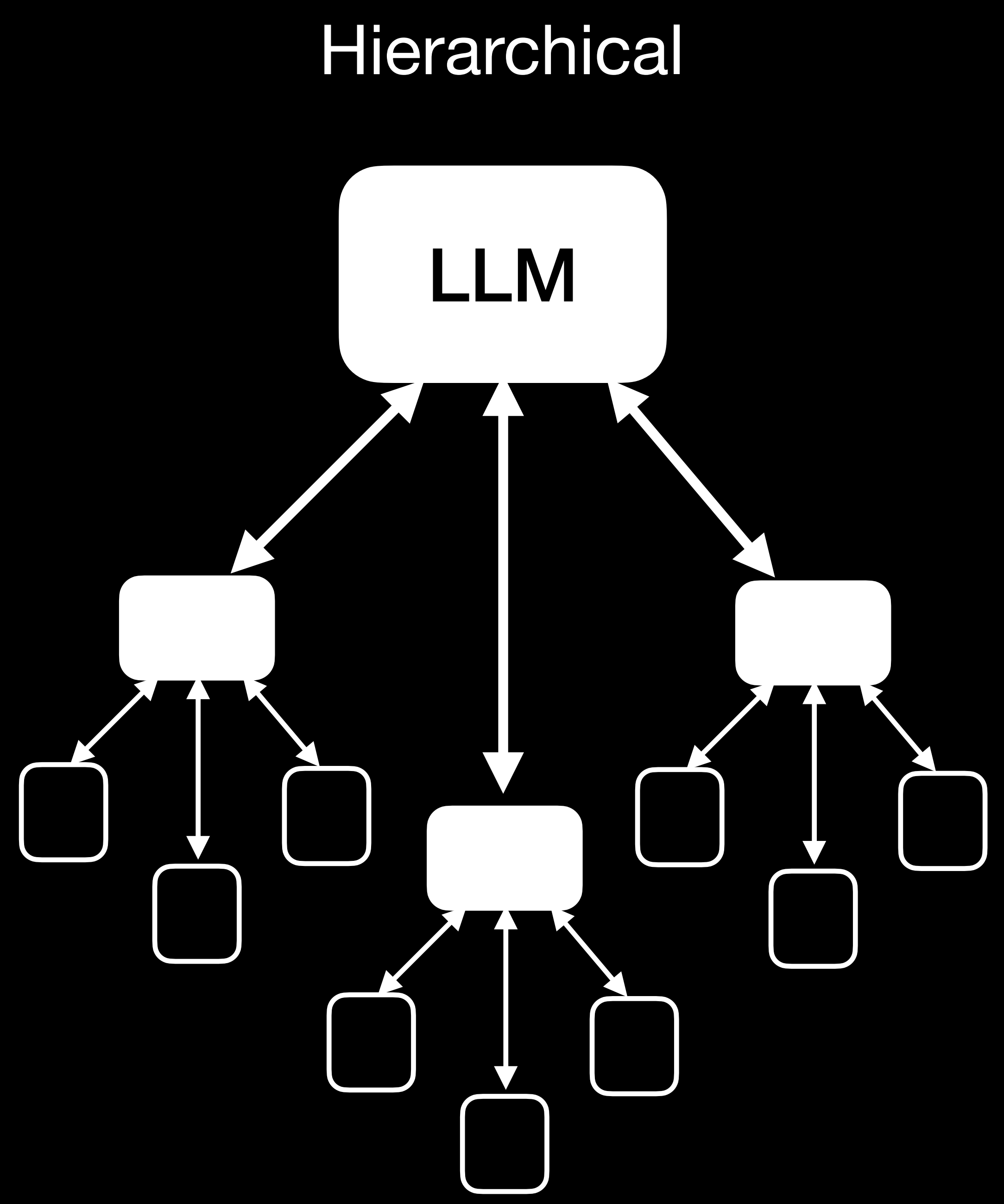
This is why this rather falls under the category of an agentic systems, not fully autonomous agents. The autonomy is orchestrated, not self-evolving.

Paper on higher Autonomous Agent category: https://arxiv.org/pdf/2305.16291

Example Multi-Agent Architectures

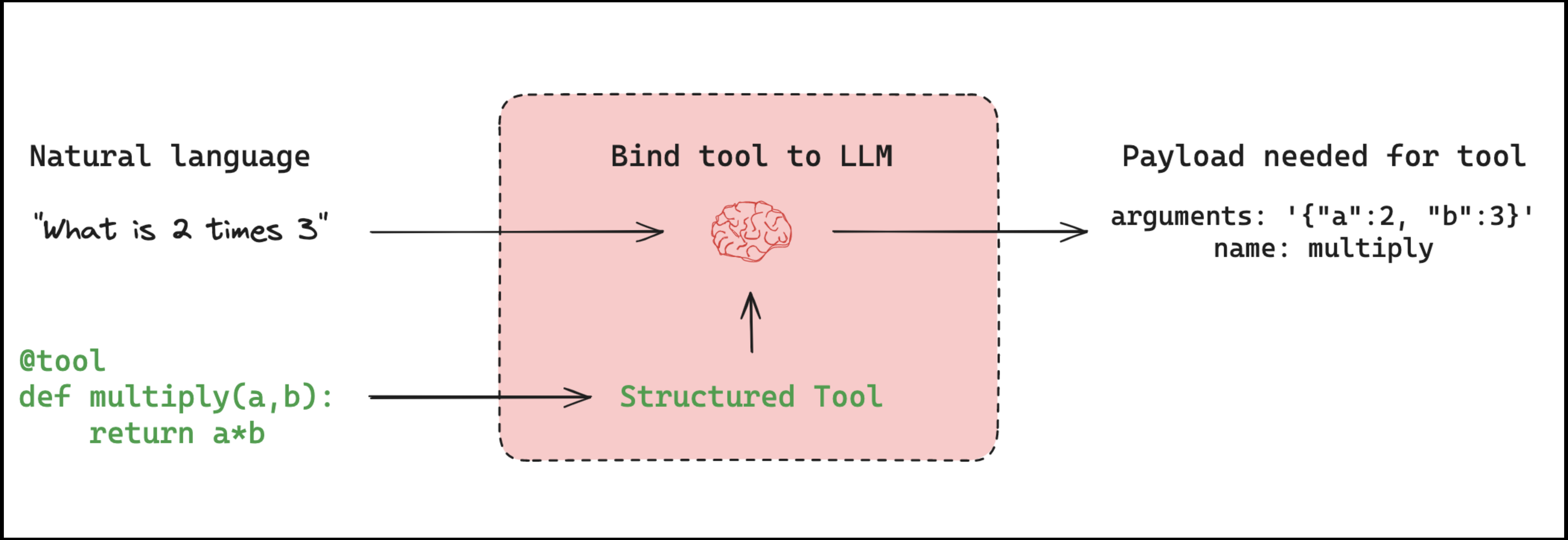






In a supervisor architecture, each agent communicates with a single supervisor agent, which makes decisions on which agent should be called next. In contrast, a hierarchical architecture features a multi-agent system with a supervisor of supervisors. This generalizes the supervisor architecture and allows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

Tool Calling with LangGraph



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

Image-Source: LangChain

Primary Benefits of Using Multi-agent Systems

- **Modularity**: Separate agents make it easier to develop, test, and maintain agentic systems.
- **Specialization**: You can create expert agents focused on specific domains, which helps with the overall system performance.
- **Control**: You can explicitly control how agents communicate (as opposed to relying on function calling).

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) -> facilitates multi-actor (or multi-agent) communication. 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.

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.

Command (Edgeless graphs) 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"
    )
```

Command, when returned from a node specifies not only the update to the state (as usual in LangGraph) but also which node to go to next. This allows nodes to more directly control which nodes are executed after-the-fact.

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"}
    )
```

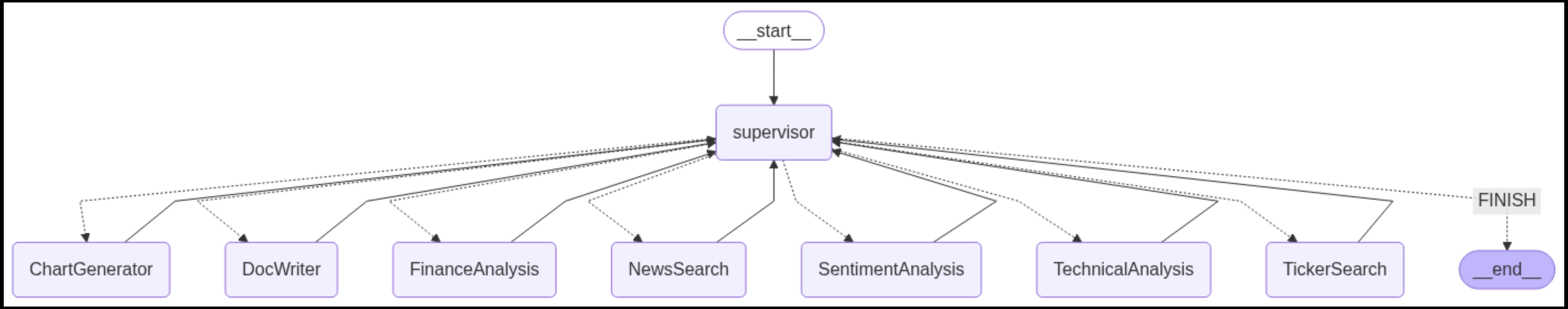
Hands-on:
Code Walkthrough

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.
- Expert Technical Analyst Agent conducts a technical analysis of a given asset.
- Fundamental Analyst Agent conducts a fundamental analysis of a given asset.
- News Sentiment Agent for sentiment analysis.
- 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.

Multi-Agent System for Investment Analysis



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.

Human in the Loop (HITL) with LangGraph

There are typically three different actions that you can do with a human-in-the-loop workflow:

- Approve or Reject: Pause the graph before a critical step, such as an API call, to review and approve the action. If the action is rejected, you can prevent the graph from executing the step, and potentially take an alternative action. This pattern often involve routing the graph based on the human's input.

- Edit Graph State: Pause the graph to review and edit the graph state. This is useful for correcting mistakes or updating the state with additional information. This pattern often involves updating the state with the human's input.

- Get Input: Explicitly request human input at a particular step in the graph. This is useful for collecting additional information or context to inform the agent's decision-making process or for supporting multi-turn conversations.

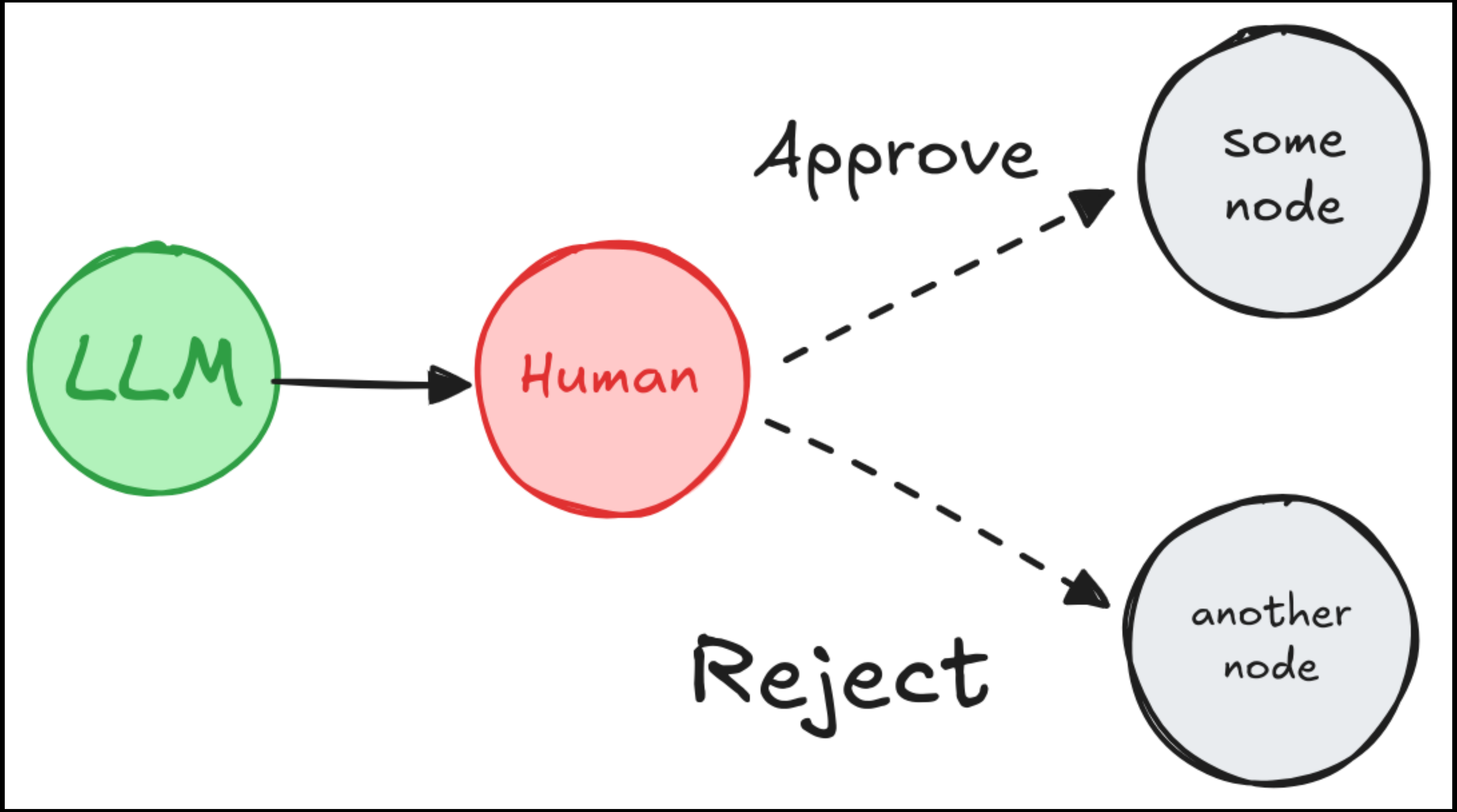
Human In The Loop

```python
@tool
def human_assistance(query: str) -> str:

    """Request assistance from a human."""
    human_response = interrupt({"query": query})

    return human_response["data"]
```

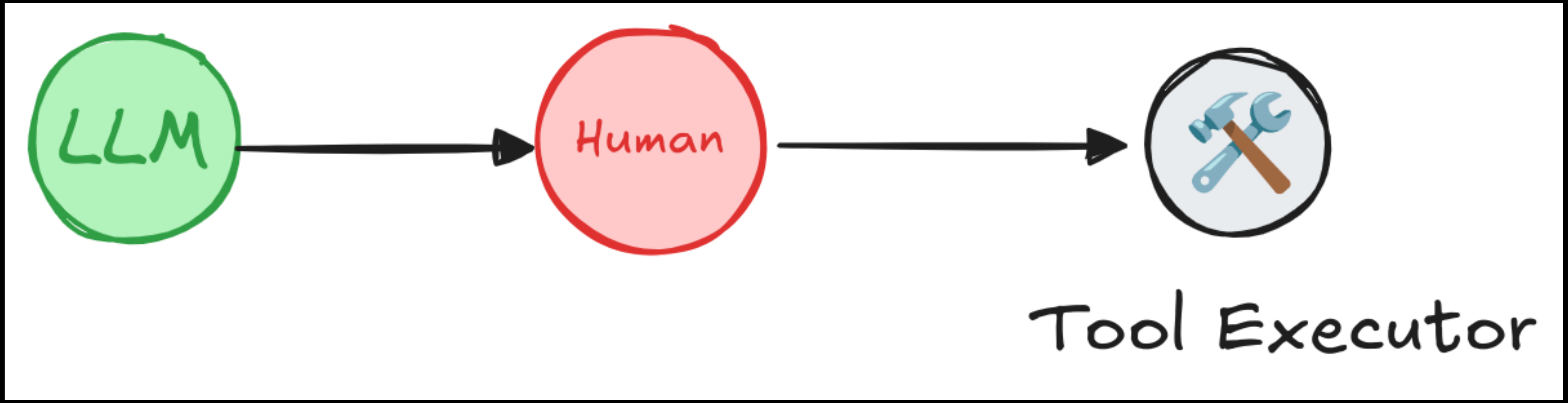
HITL: Approve or Reject



Depending on the human's approval or rejection, the graph can proceed with the action or take an alternative path.

Image-Source: LangChain

HITL: Review Tool Calls



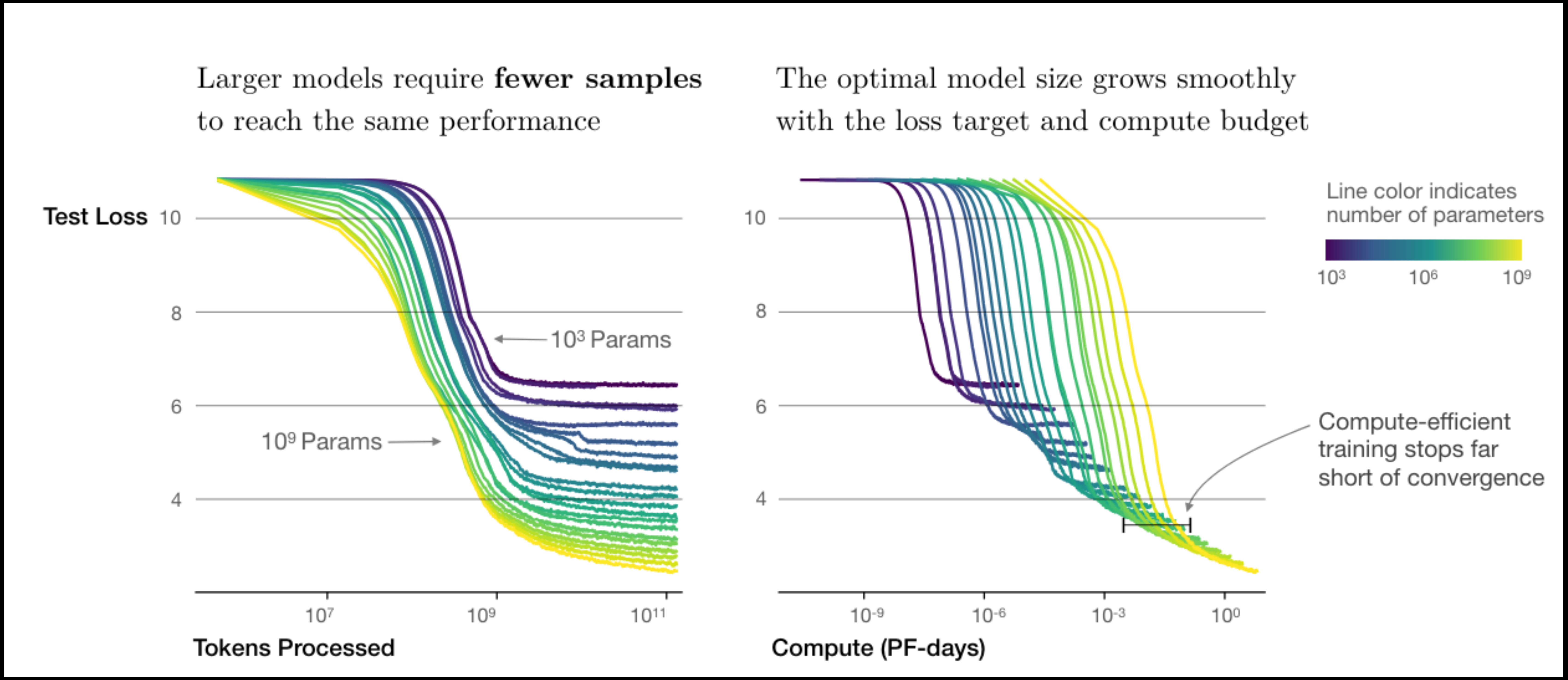
A human can review and edit the output from the LLM before proceeding. This is particularly critical in applications where the tool calls requested by the LLM may be sensitive or require human oversight.

Image-Source: LangChain

Hands-on:
Code Walkthrough HITL

Scaling Laws

Scaling Laws



The scaling law states that model performance improves in a predictable power-law fashion as you increase the amount of data, the number of parameters, and the compute used. Meaning, the sheer volume of data helps the model to generalize better and can perform well on a wide range of tasks without task-specific fine-tuning.

Paper: Scaling Laws for Neural Language Models, https://arxiv.org/pdf/2001.08361

Advanced LLM Agents

ReAct: Reasoning Iteratively Over Context

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

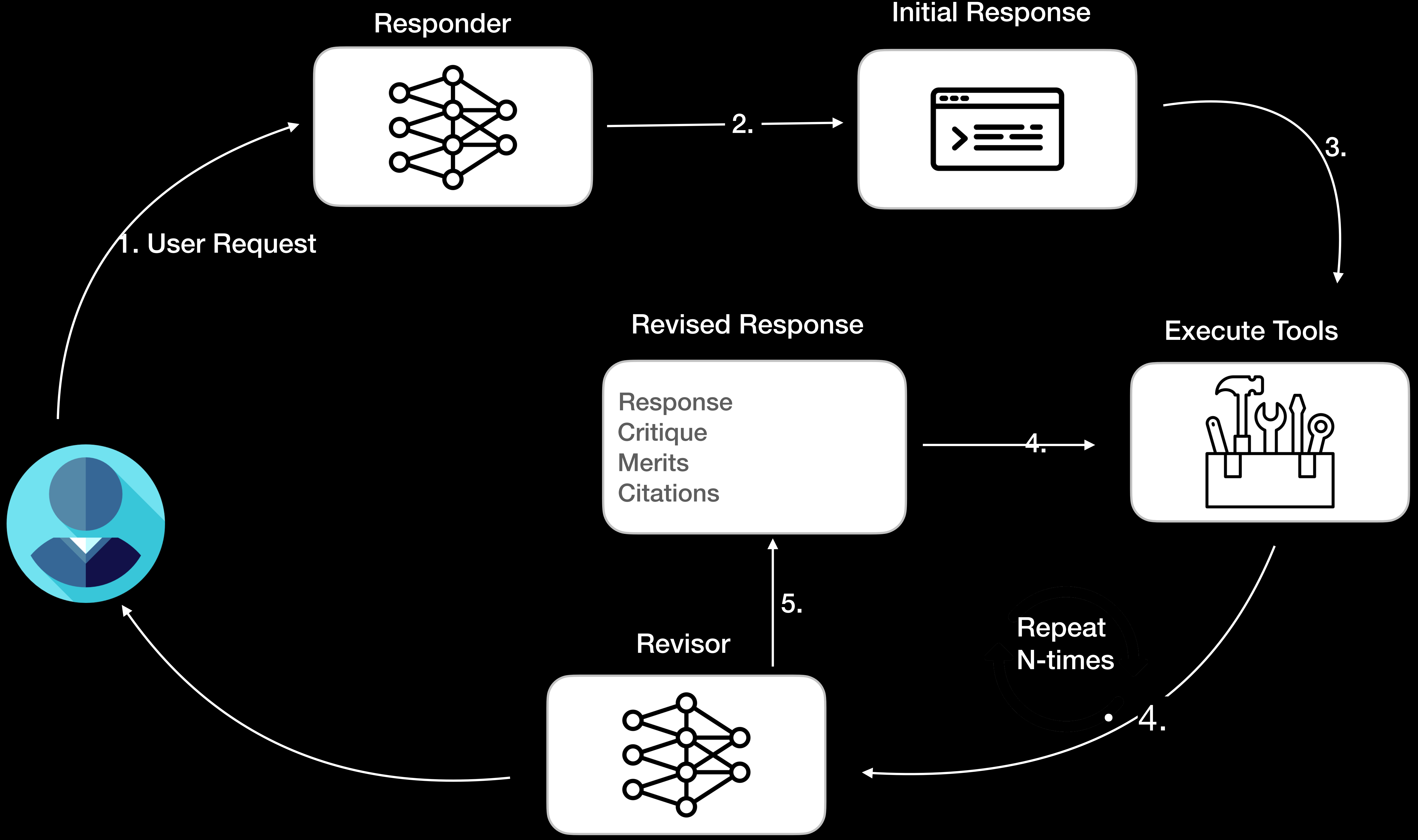
Policy $\pi(a_t \mid c_t)$ where $c_t = (o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t)$

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

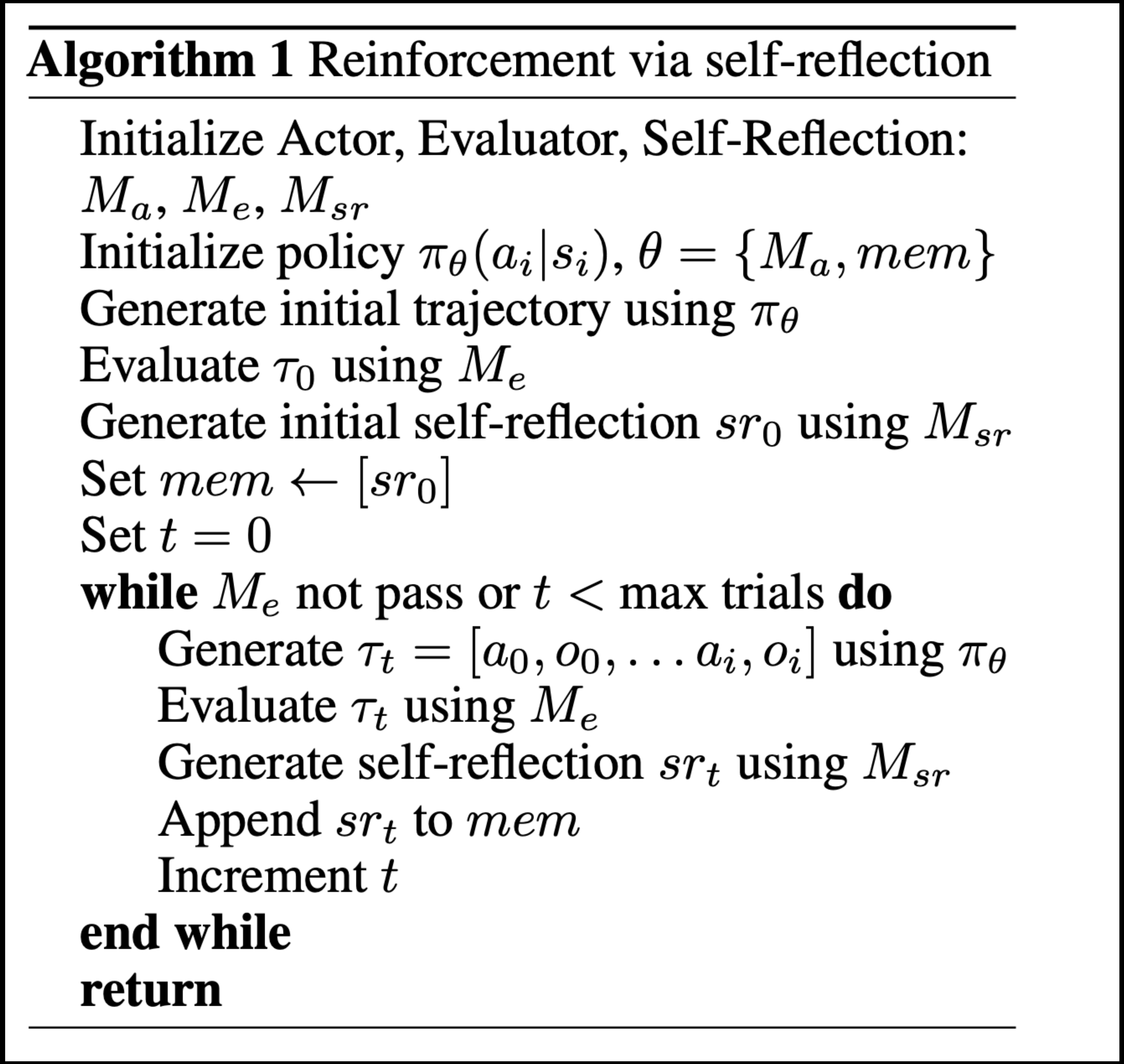
Actions (A): Task-specific steps impacting the environment.
Reasoning Traces (L): Language-based reasoning, or "thoughts," updating

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

Reinforcement learning (RL) x LLMs = ICRL*

*In-context Reinforcement Learning

RL-Basics: Explore-exploit Tradeoff

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

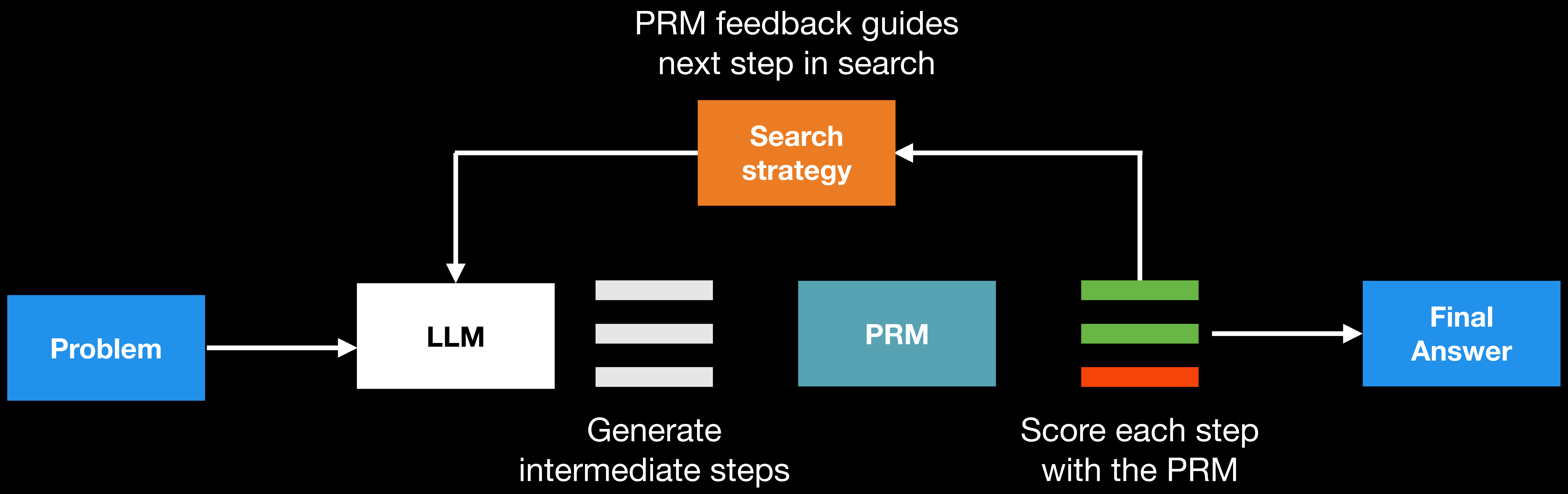
Test-Time Compute in LLMs: Thinking Smarter, Not Bigger

- **The Problem:** Training ever-larger LLMs is becoming incredibly expensive. Test-time compute offers a complementary way to improve performance after training. It's about making the LLM "think longer" during inference, not just making the model bigger.

- **Why it Matters:** Smaller models, given more "thinking time" (compute) at test time, can sometimes outperform much larger models, especially on complex reasoning tasks. This is crucial for deploying LLMs on resource-constrained devices or reducing inference costs.

- **Key Idea:** Instead of just taking the LLM's first answer, we use strategies to explore multiple reasoning paths and/or refine the output.

System Setup for Test-time Compute


PRM feedback guides next step in search
Search strategy
Problem
LLM
PRM
Final Answer
Generate intermediate steps
Score each step with the PRM

1. We begin by feeding a math problem to an LLM, which generates N **partial solutions**, e.g. an intermediate step in a derivation.
2. **Each step is scored by a process reward model (PRM)**, which estimates the probability of each step to eventually reach the correct final answer. The steps and PRM scores are then used by a given search strategy to select which partial solutions should be further explored to generate the next round of intermediate steps.
3. Once the search strategy terminates, the final candidate solutions are ranked by the PRM to produce the final answer.

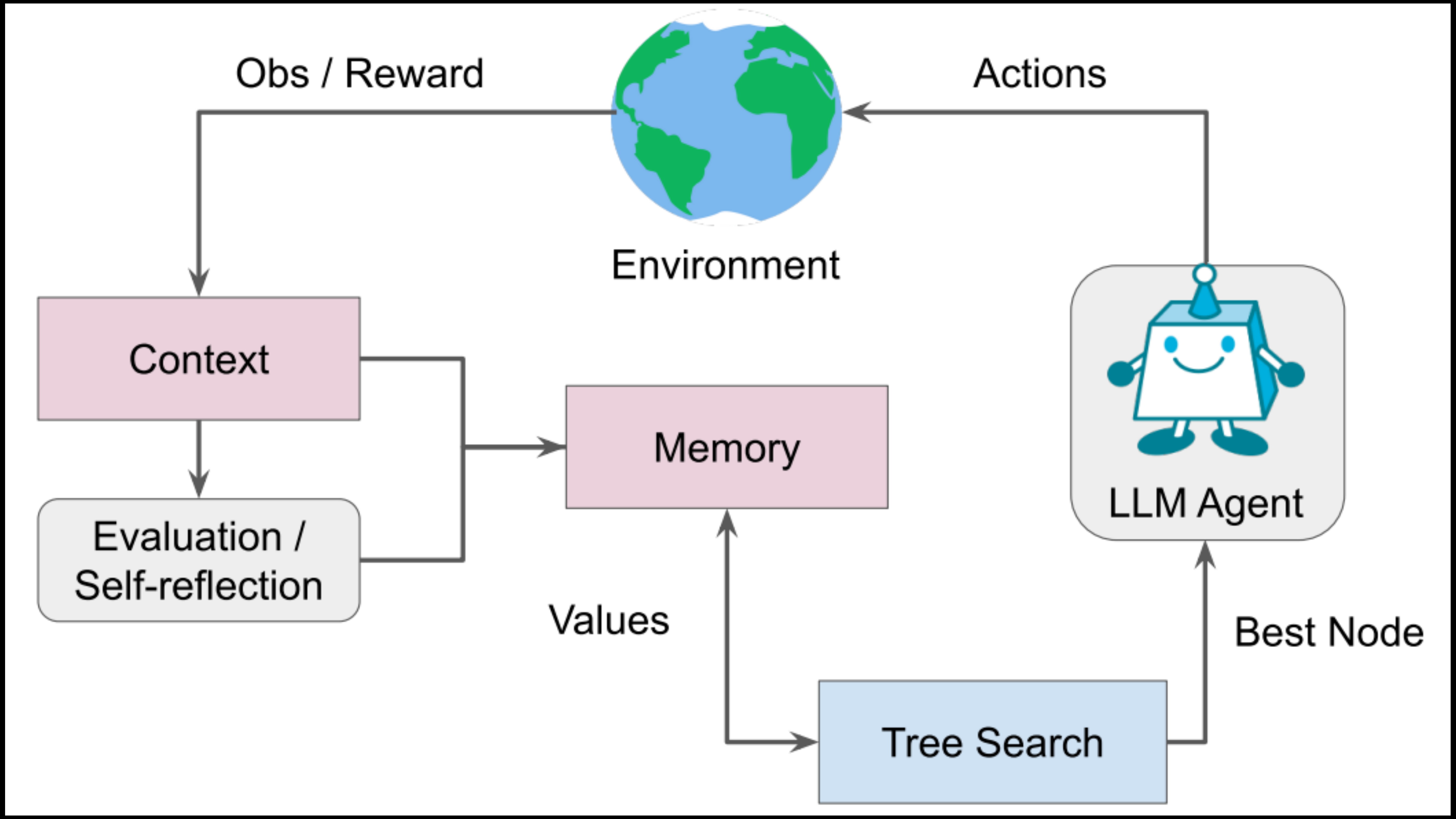
Compute-optimal Scaling

$$\theta^*_{q,a^*(q)}(N) = \mathrm{argmax}_\theta \left(E_{y\sim\mathrm{Target}(\theta,N,q)} \left[1_{y=y^*(q)} \right] \right)$$

where $y^*(q)$ is the ground-truth for question q and $\theta^*_{q,a^*(q)}(N)$ denotes the compute-optimal scaling strategy. Since computing $\theta^*_{q,a^*(q)}(N)$ directly is somewhat tricky, an approximation is used based on the problem difficulty, i.e. allocate test-time compute according to which search strategy achieves best performance for a given difficulty level.

Paper: Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters

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

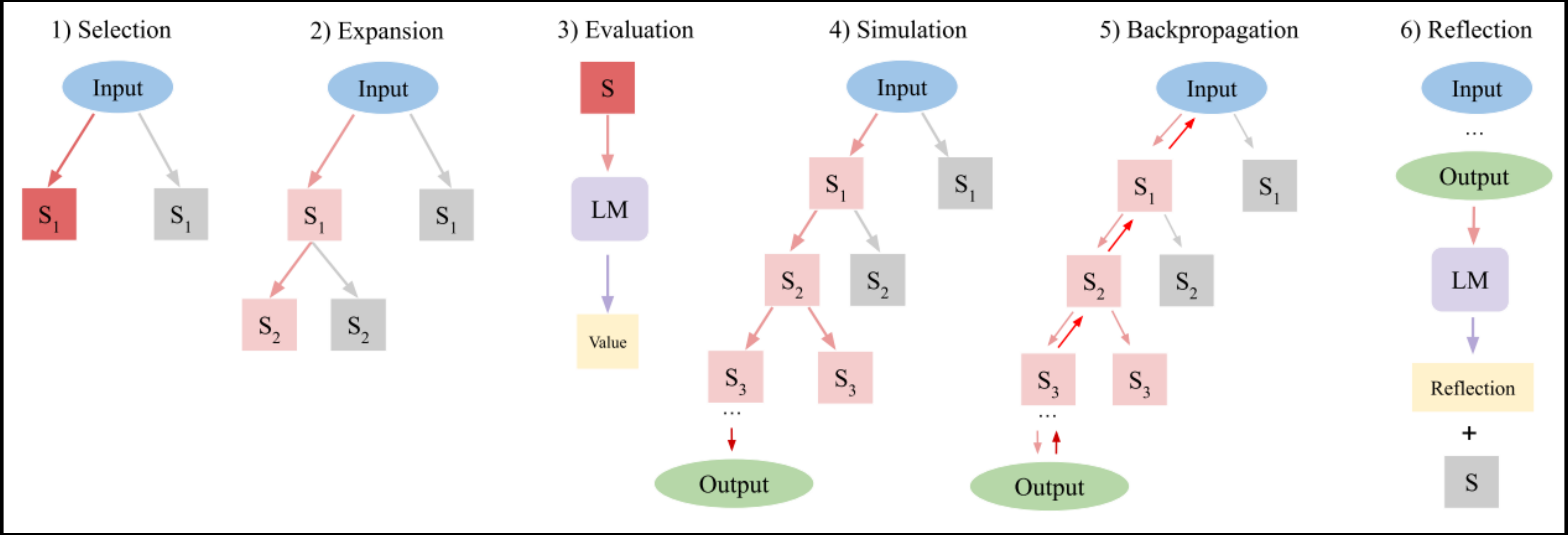
Language Agent Tree Search

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.

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