University of Ljubljana, Faculty of Computer and Information Science

LLM-based agents

Prof Dr Marko Robnik-Šikonja

Natural Language Processing, Edition 2026

Contents

- Agents
- LLM-based agents
- Use case: agents for science

- Many recent papers, see also https://agents4science.github.io
- Some slides and graphics by Ian Foster

Agents

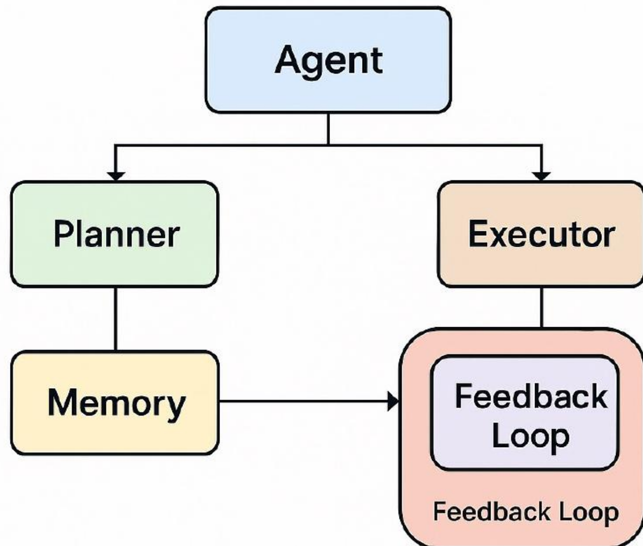
- What is an “agent”?
- In computer software, a “software agent” is [Wikipedia] “**a computer program that acts for a user or another program in a relationship of agency**”
- In AI, an “intelligent agent” is [also Wikipedia] “**an entity that perceives its environment, takes actions autonomously to achieve goals, and may improve its performance through machine learning or by acquiring knowledge**”
  – An AI component, sensors, actuators, memory, etc.

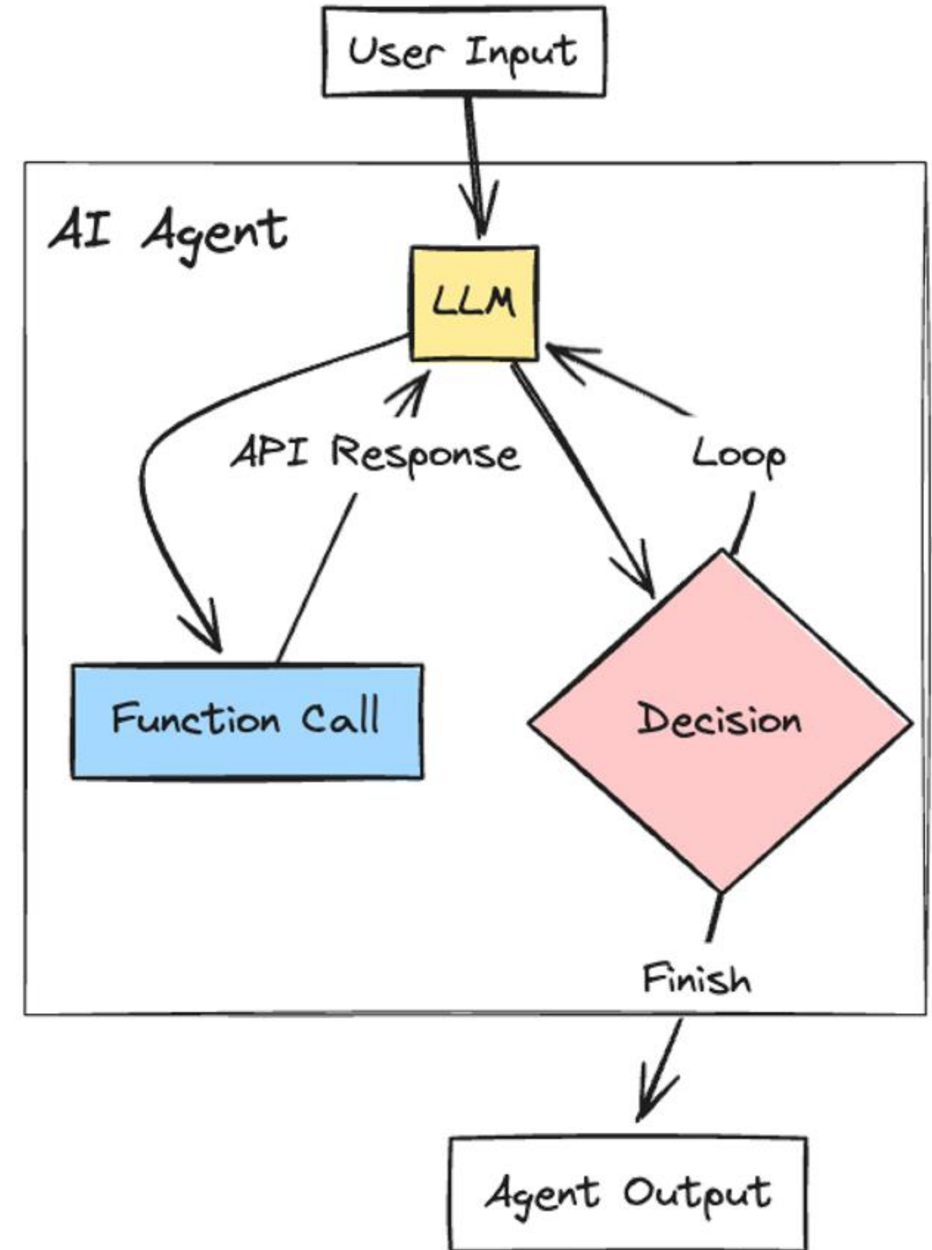
Agent loop

- The “sense-plan-act-learn” loop:
- Initialise state and goals
- Repeat until termination:
  - **Sense**: Gather observations from the environment
  - **Plan**: Evaluate goals and state, plan/select next action
  - **Act**: Execute chosen action on the environment
  - **Learn**: Update internal state, memory, or model based on outcomes

Agents hype

- Current (post 2025) excitement around agents is due to LLMs
- Many similar agent architectures





Example: Chemistry agent CACTUS

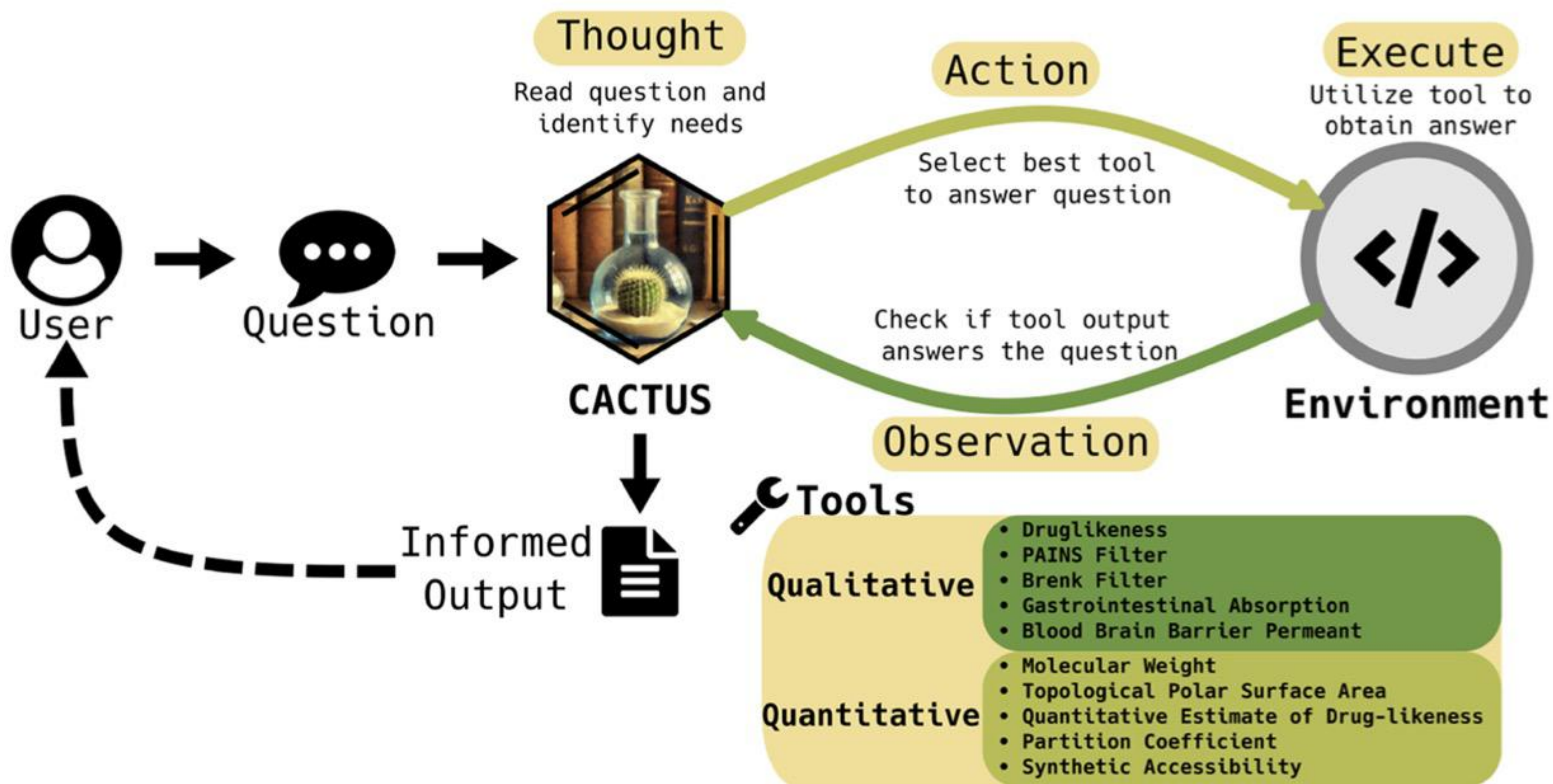


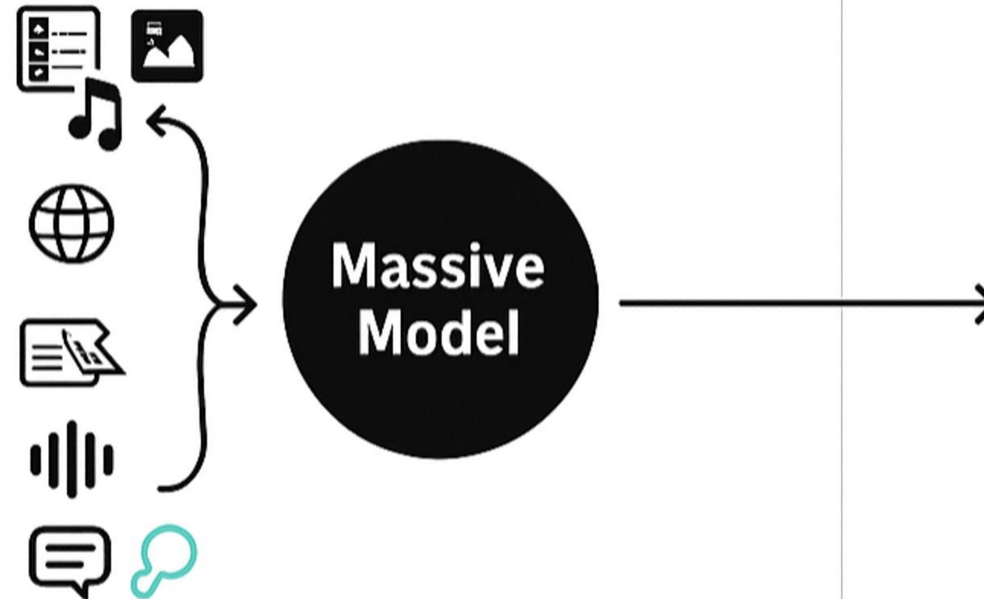
Figure 1. General workflow of the CACTUS agent that details how the LLM interprets input to arrive at the correct tool to use to obtain an answer. Starting from the user input, CACTUS follows a standard "Chain-of-thought" reasoning method with a Planning, Action, Execution, and Observation phase to obtain an informed output.

From ML to agents

Traditional ML

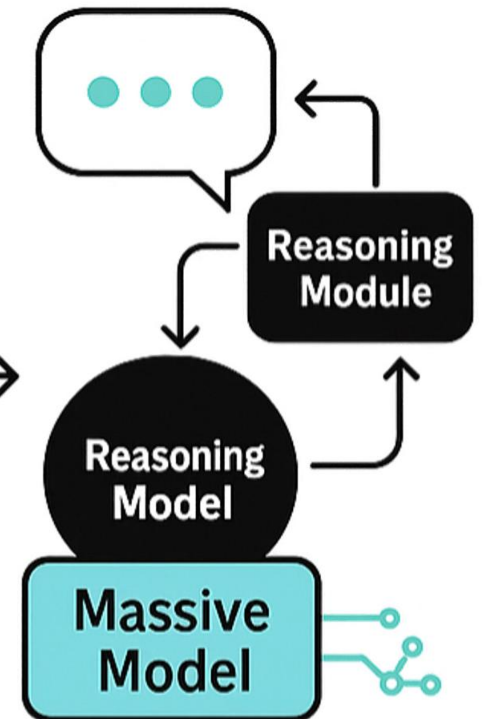
- Individual siloed models
- Require task-specific training
- Lots of human supervised training

Foundation models or LLMs



- Massive multi-modal model
- Adapted with minimal training
- Pre-trained unsupervised learning

Reasoning models



- Deliberative multi-step logic
- Self-consistency checks
- Slower but more accurate inference

LLM-based agents

- AI agents are autonomous systems that can **reason about tasks and act to achieve goals by leveraging external tools and resources**.
- Modern AI agents are typically powered by large language models (LLMs) **connected to external tools or APIs**.
- They can perform **reasoning**, **invoke** specialized models, and **adapt** based on feedback.
- Agents differ from conventional "models" in important regards:
  - They are interactive and adaptive
  - Rather than returning fixed outputs, they can take multi-step actions, integrate context, and support iterative human–AI collaboration.
  - Users can interact with them through human language, substantially reducing usage barriers

How many agents?

- In **principle**, a single reasoning model (like a single human) can apply a variety of different reasoning strategies, have specialized knowledge on different topics, improve expertise over time, etc.
- In **practice**, it is common to create multiple agents (like a team of people), e.g., for:
  - Modularity (specialization, reuse, maintainability)
  - Different roles (e.g., idea generator, idea critic, program generator, …)
  - Parallelism (run multiple copies of an agent to explore different ideas)
  - Expanded capacity (e.g., larger LLM context)

Examples of agents roles

- **Computing agent** utilizes computational models as tools
- **Decision agent** makes decisions in response to given conditions
- **Database agent** retrieves relevant information from databases
- **Reasoning agent** capable of direct reasoning and reasoning with feedback
- **Expert agent** provides professional consultation based on reliable sources, such as domain expertise, feedback from human experts, and results of specific tools
- **Hypothesis agent** capable of reflective learning and reasoning to generate hypotheses
- **Planner agent** devises plans for future actions
- **In silico/vitro agent** uses tools in silico or in vitro environment

Self-improvement and feedback loops

- Turn the foundation model onto itself:
- **Self-consistency**: Generate multiple CoT responses and vote on final answers.
- **Reflexion & Reflection**: Ask the model to critique and revise its own output.
- **STaR (Self-Taught Reasoner)**: Use its own outputs to fine-tune itself.
- These methods moved models from **single-shot output** to **deliberative computation**, aligning with Kahneman's System 2"-like behavior.

What reasoning model GPT o5 likely does

- **Parallel Reasoning at Test-Time:** Run multiple internal reasoning paths in parallel, then aggregate or vote among them before deciding on a final answer. (A pattern seen in many top-tier reasoning systems.)
- **Adaptive Reasoning Effort:** Dynamically scale the amount of internal compute (thinking steps) based on the complexity of the prompt — either automatically via the system, or via a parameter like reasoning_effort.
- **Routing Layer:** The model might route between lighter vs heavier reasoning modes. For easier queries, it may use a simpler path; for complex mathematics, it may escalate to deeper reasoning steps. This is consistent with the GPT-5 architecture. The model likely can call tools, perform code execution, search, or retrieve external knowledge as part of its reasoning pipeline.
- **Longer Context / Memory Handling:** To sustain reasoning over long chains or multi-step problems, it may maintain internal memory or context beyond just the prompt tokens.
- **Higher Accuracy / Reduced Hallucination via Verification:** It may include internal verification steps: after generating a candidate answer, re-evaluate or cross-check with alternative reasoning paths to ensure consistency.
- **User-Controlled Verbosity / Effort Trade-Offs:** Provide knobs (parameters) so the user can dial down reasoning effort for speed, or dial it up for precision.

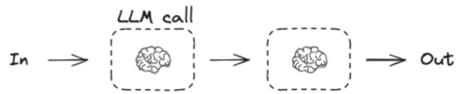
Agent architectures

- **Workflows:** Systems where LLMs and tools are orchestrated through predefined code paths.
- **Agents: S**ystems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

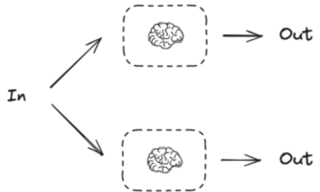
Workflows and agents

Workflows

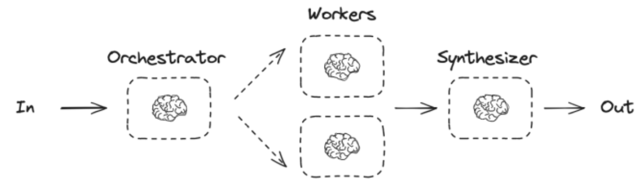
Prompt Chaining



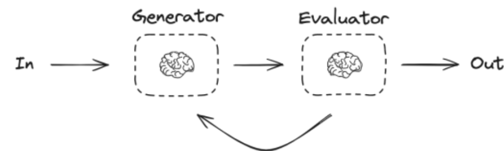
Parallelization



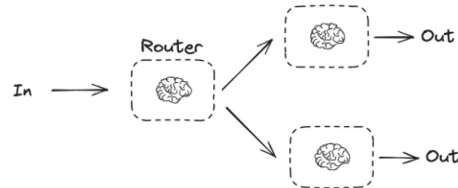
Orchestrator-Worker



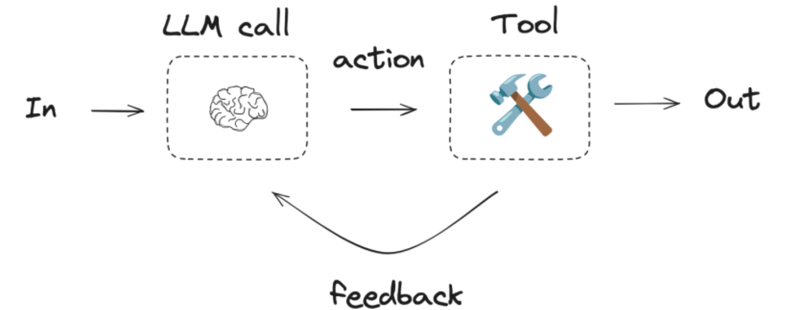
Evaluator-optimizer



Routing



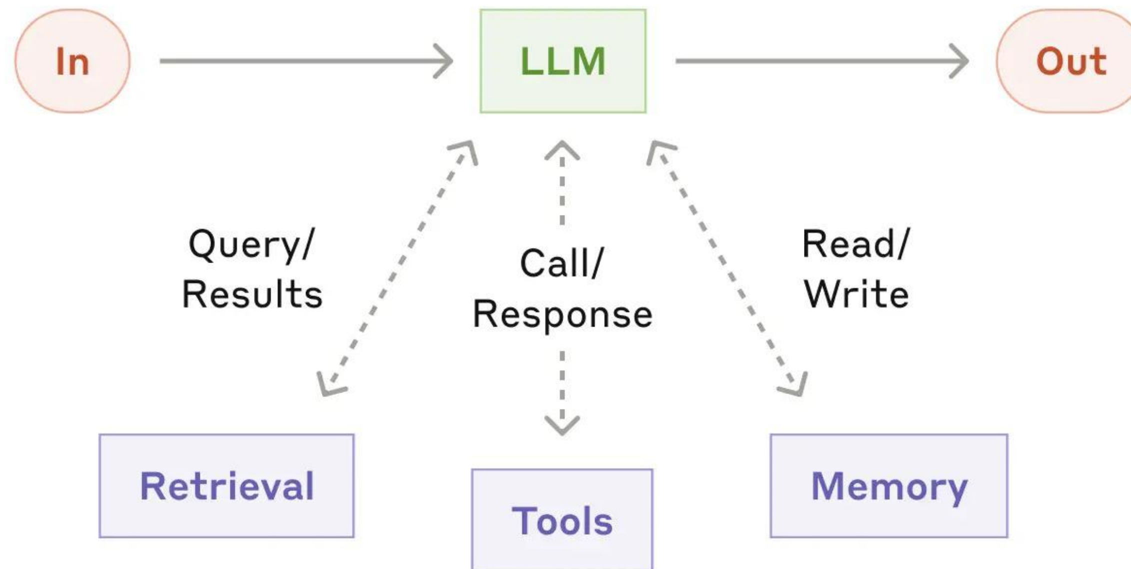
Agent



LLM is embedded in predefined code paths	LLM directs control flow through predefined code paths	LLM directs its own actions based on environmental feedback

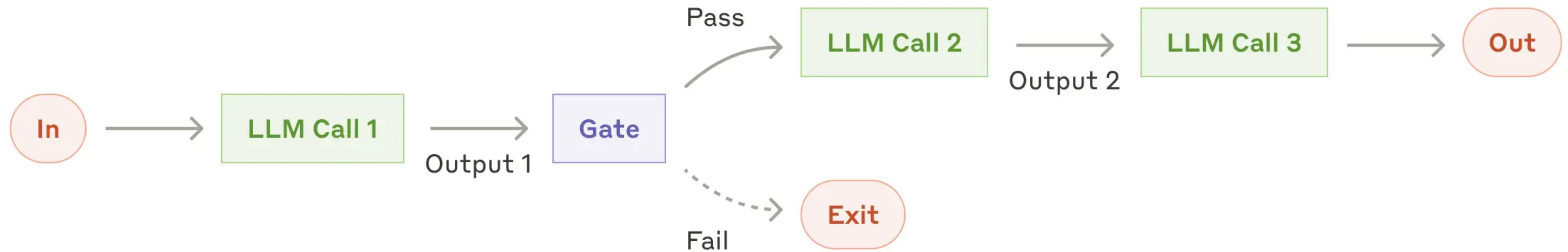
Workflows

- Building block: The augmented LLM



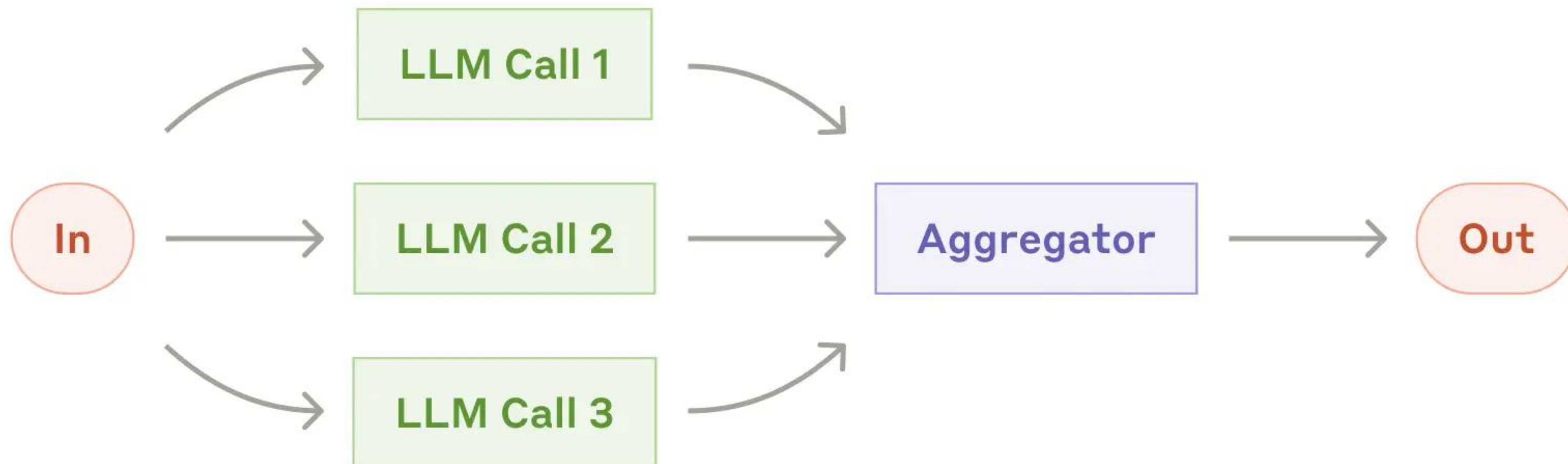
Prompt chaining workflows

- Decompose a task into a sequence of steps, where each LLM call processes the output of the previous one.
- You can add programmatic checks ("gate” in the diagram) on any intermediate steps to ensure that the process is still on track.



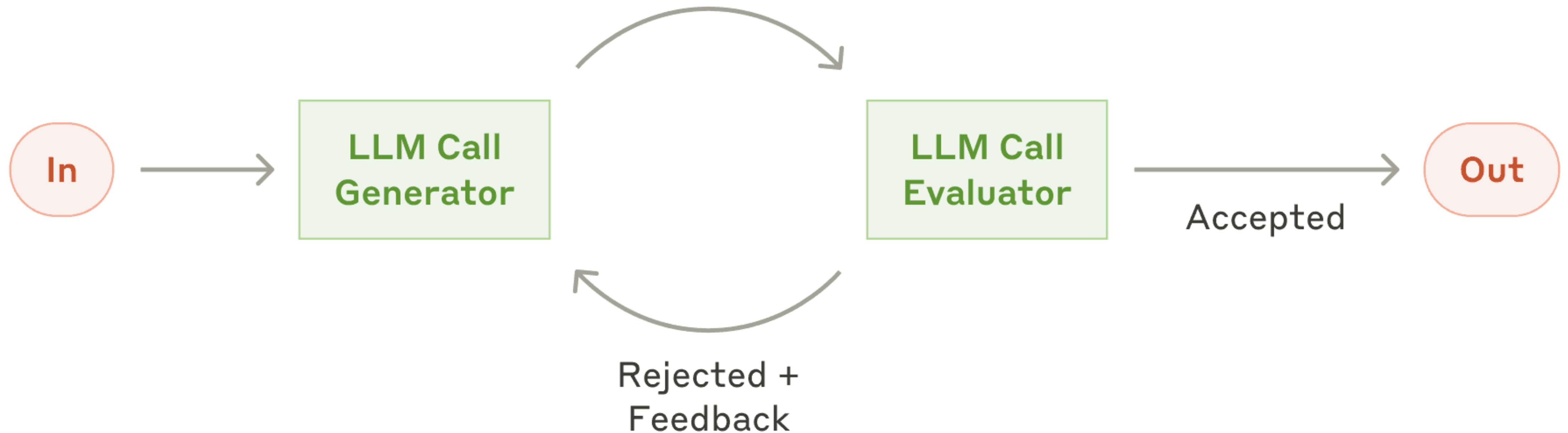
Routing workflows

- Routing classifies an input and directs it to a specialized followup task. Allows for separation of concerns, and building more specialized prompts. Otherwise, optimizing for one kind of input can hurt performance on others.



Routing workflow: Evaluator-optimizer

- One LLM call generates a response while another provides evaluation and feedback in a loop



Recall the agent loop

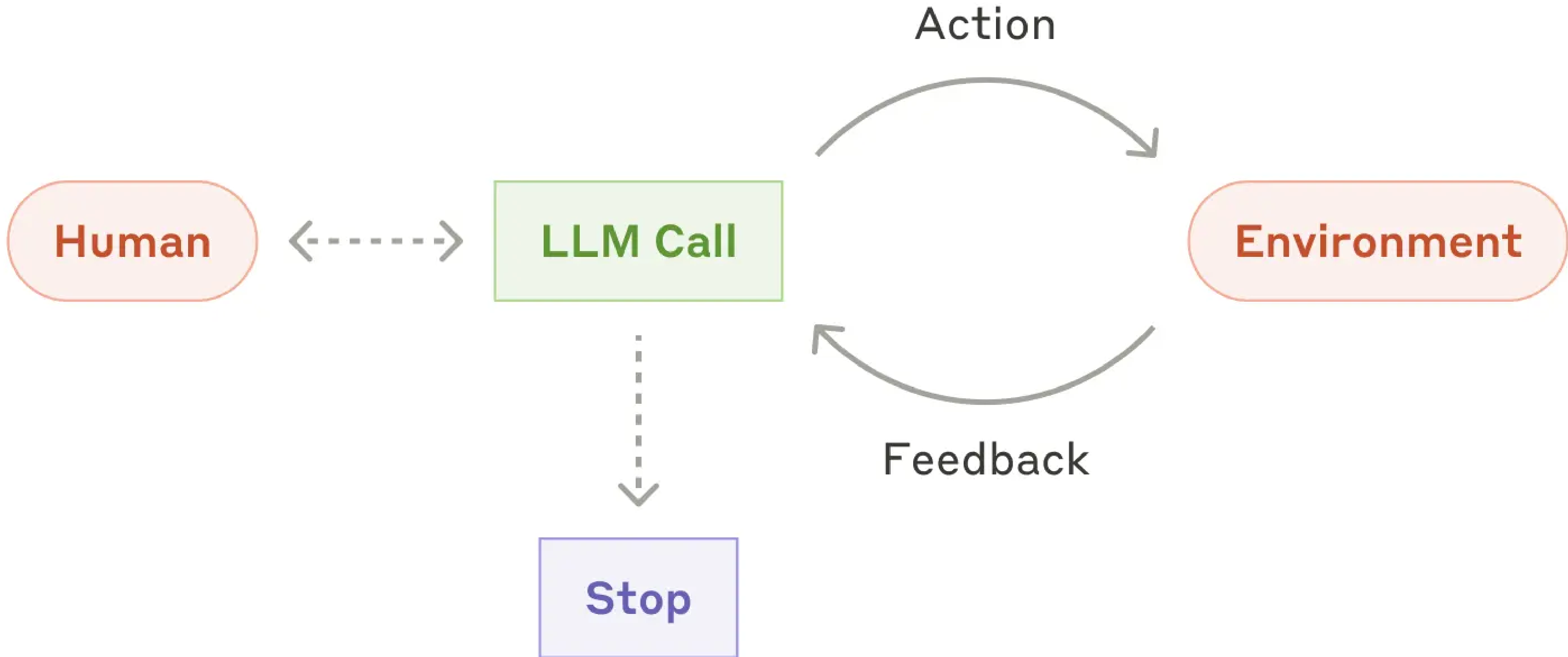
- The “sense-plan-act-learn” loop:
- Initialise state and goals
- Repeat until termination:
  - **Sense**: Gather observations from the environment
  - **Plan**: Evaluate goals and state, plan/select next action
  - **Act**: Execute chosen action on the environment
  - **Learn**: Update internal state, memory, or model based on outcomes

- **In sequence:** call tool, invoke LLM, test for termination; repeat

LLM agents

- Agents begin their work with either a command from, or interactive discussion with, the human user.
- Once the task is clear, agents plan and operate independently, potentially returning to the human for further information or judgement.
- During execution, agents gain “ground truth” from the environment at each step (such as tool call results or code execution) to assess progress.
- Agents can pause for human feedback at checkpoints or when encountering blockers.

LLM agents

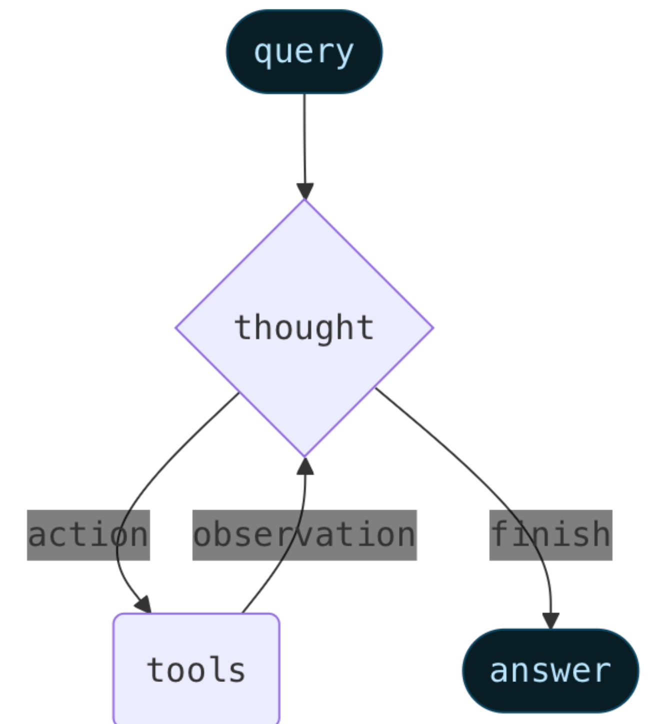


Example: ReAct loop with LangChain create-agent

- ReAct frames an agent's behavior as an interleaving of thought -> action -> observation

steps, where the model writes out its reasoning, picks a tool, sees the tool's result, and then repeats.

- LangChain **create_agent()** can be used to implement ReAct loops:
  - It builds a graph-based agent runtime using LangGraph, where a graph consists of **nodes** (steps) and **edges** (connections) that define how your agent processes information.
  - The agent moves through this graph, executing nodes like the **model** node (which calls the model) & **tools** node (which executes tools)



Multi-agent systems

Why multiple agents

- Partition expertise
- Run in parallel
- Contain risk
- Govern data

Why multiple agents?

- Specialization improves quality: Narrow, role-tuned agents outperform generalists on their slice
- Parallelism increases throughput: Independent agents work concurrently on sub-tasks
- Heterogeneous tools & environments: Different agents own different toolchains (HPC, LIMS, robots, DBs)
- Robustness & fault isolation: One agent can fail, restart, or be replaced without taking down the system
- Self-checking & consensus: Cross-agent critique, redundancy, and voting reduce error and bias
- Long-running workflow control: Persistent agents watch queues, schedule jobs, and resume after interruptions
- Governance, security, provenance: Data stays within the minimal-privilege agent; actions are auditable by role
- Scalability & cost control: Orchestrators spawn cheap/ephemeral workers; premium models only where needed

Challenges of multi-agent systems

- Creating agents
- Monitoring agents
- Inter-agent communication
- Monitoring agents/agent lifecycle

Perils of multiple agents

Risk Area	Description	Mitigation
Coordination Chaos	Agents loop, contradict, or amplify each other's errors.	Limit turn count, add arbiter/watchdog agent.
Loss of Accountability	Hard to trace which agent caused errors or bias.	Log every message & decision; require provenance.
Emergent Behavior	Unpredictable dynamics—agents invent new "protocols."	Constrain roles and allowable message types.
Security & Safety	Tool calls, code exec, or API use may cause harm.	Sandbox actions, validate outputs, human approval.
Human Factors	Over-trust, confusion, or reduced oversight.	Keep humans in the loop; clear UI for agent reasoning.
Resource/Cost Blow-up	Many agents → exponential token and time costs.	Prune redundant roles; use shared memory or caching.

How can agent systems help?

- Scheduling & Fairness: Coordinated access to LLM cores, memory, and tools (AIOS kernel)
- Isolation & Governance: Sandboxed agents with explicit privileges; traceable syscalls
- Reproducibility: Logged context snapshots and deterministic replay improve (scientific) auditability
- Safety & Oversight: Policy enforcement and user-intervention checkpoints before critical actions
- Scalability: From a handful of agents to thousands sharing compute, without chaos
- Example: AutoGen, see https://arxiv.org/abs/2308.08155
- Anthropic advices: https://www.anthropic.com/engineering/multi-agent-research-system

AutoGen

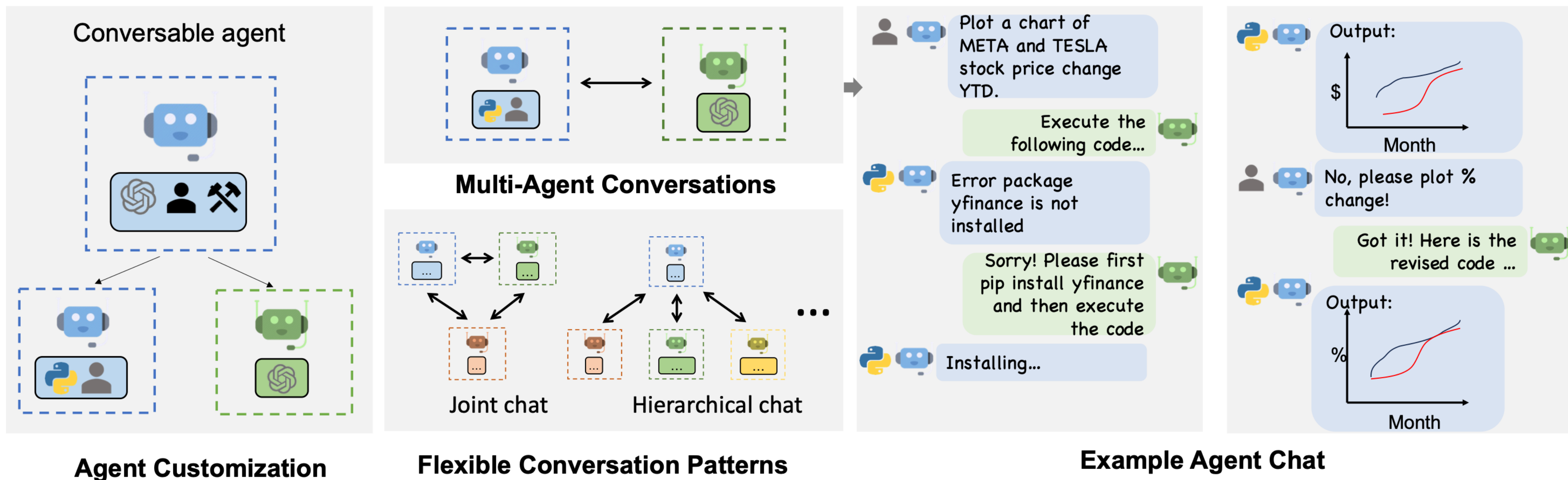


Figure 1: AutoGen enables diverse LLM-based applications using multi-agent conversations. (Left) AutoGen agents are conversable, customizable, and can be based on LLMs, tools, humans, or even a combination of them. (Top-middle) Agents can converse to solve tasks. (Right) They can form a chat, potentially with humans in the loop. (Bottom-middle) The framework supports flexible conversation patterns.

AIOS: LLM Agent Operating System

- Proposes a comprehensive system-level architecture for serving LLM- based agents
- introduces a **kernel-based architecture** that treats LLMs and their associated tools as *first-class operating-system resources*.

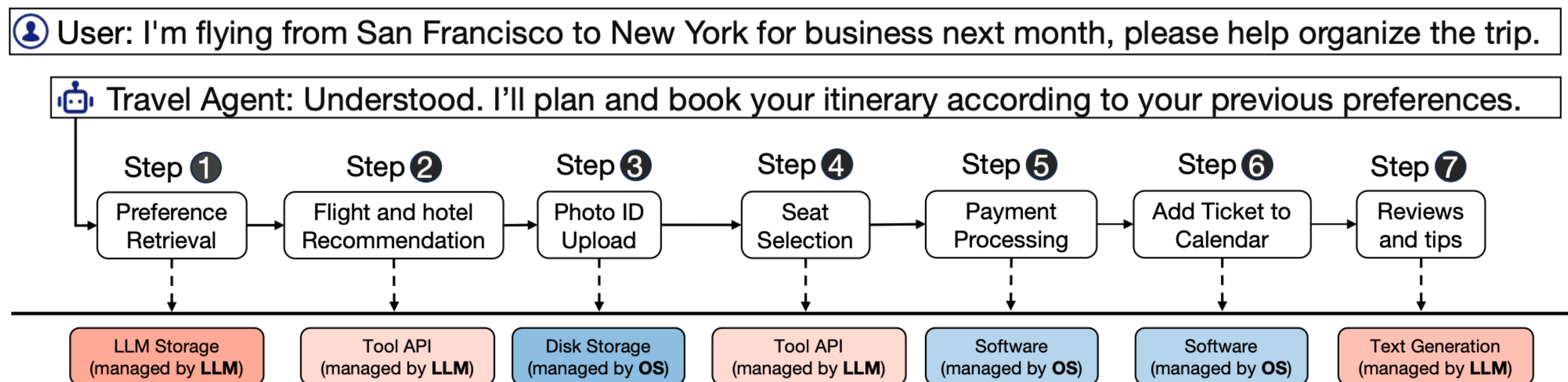


Figure 1: A motivating example of how an agent (i.e., travel agent) requires both LLM-related and Non-LLM-related (i.e., OS) services to complete a task, where color in red represents services related to LLM and color in blue represents services not related to LLM.

Three agent systems compared & contrasted

Axis	**AutoGen**	**LangGraph**	**AIOS**
Abstraction	Conversation-programmed agents	Graph/state machine	OS-style runtime (kernel + SDK)
Primary concern	Inter-agent **dialog + tools**	**Deterministic orchestration**, persistence, human-in-loop	**Scheduling**, memory/context, storage, **access control**
State	Conversation history + tool results	Explicit shared **state object**	Kernel-managed **contexts/ memory**
When to use	Fast prototyping of multi-agent patterns	Production flows needing reliability & debuggability	Multi-tenant, long-running, resource-intensive deployments

Tool Calling

- Methods for invoking external tools from reasoning models.
- Informal standard is model context protocol (MCP)
- Motivation
  - Pure text-based reasoning has limitations: e.g., computation, precision, data freshness
  - Thus, we want *tool-augmented reasoning*: models calling external APIs, databases, or simulations
  - Much like a scientist using an instrument or database

Getting LLMs to calculate

- Example: What is the pH of a 0.01 M solution of hydrochloric acid?
- meta-llama/Meta-Llama-3-8B-Instruct: **pH 1.0**
- meta-llama/Meta-Llama-3-70B-Instruct: **pH 2.0**
- Can be calculated: **pH = –log10(0.01) = 2**
- tool_call: calculator.log10(x=0.01)

Bridging models and tools

- LLM expresses itself in language: e.g., “please run a program to compute X” or “compute the negative log of 0.01”
- Meanwhile:
  - A “tool” might be an executable, Python function, workflow, …
  - It might expect input, and produce output, in a variety of formats
  - It might run on your PC, on an HPC system, in the cloud
  - It might require specialised permissions to use
  - The request may refer (implicitly or explicitly) to past interactions
  - Available tools may change over time or depend on user identity
  - Etc.

Model Context Protocol

- A **protocol** designed to let **models** exchange and extend their **context** in a standardized way
- **Models** (LLMs, reasoning systems, multi-agent components)
  - Need to interact with external resources (tools, files, APIs, simulations)
  - MCP gives these models a defined communication channel to do so
- **Context** = everything the model can condition on while reasoning
  - external knowledge or data pulled from tools
  - task parameters and metadata
  - persistent state shared across calls or agents
  - The results of prior computations
- **Protocol**: A structured, interoperable message format that allows models, clients, and tool servers to exchange context:
  - how to describe tools (schemas, parameters)
  - how to invoke them (requests/responses)
  - how to carry metadata (context_id, auth_token, timestamp)
  - how to share or update context objects across sessions

MDP