AI Agents for Science

Lecture 3, October 3: Systems for Agents

Instructor: Ian Foster
TA: Alok Kamatar

Crescat scientia; vita excolatur

CMSC 35370 -- https://agents4science.github.io
https://canvas.uchicago.edu/courses/67079

Curriculum

1) Why AI agents for science?

AI agents and the sense-plan-act-learn loop. Scientific Discovery Platforms (SDPs): AI-native systems that connect reasoning models with scientific resources.

2) Frontiers of Language Models

Surveys frontier reasoning models: general-purpose LLMs (GPT, Claude), domain-specific foundation models (materials, bio, weather), and hybrids. Covers techniques for eliciting better reasoning: prompting, chain-of-thought, retrieval-augmented generation (RAG), fine-tuning, and tool-augmented reasoning.

3) Systems for Agents

Discusses architectures and frameworks for building multi-agent systems, with emphasis on inter-agent communication, orchestration, and lifecycle management.

4) Retrieval Augmented Generation (RAG) and Vector Databases

Covers how to augment reasoning models with external knowledge bases, vector search, and hybrid retrieval methods.

From single-agent control → multi-agent orchestration → system-level runtime

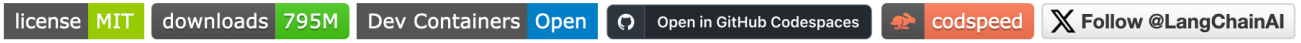
- LangGraph
  - Reframes agents as explicit graphs with state, giving you determinism and controllability for orchestration.
- AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation
  - Shows the conversation-centric paradigm for multi-agent apps (agents as conversants; tool use via messages).
- AIOS: LLM Agent Operating System
  - Treats agents as first-class OS-managed workloads with scheduling, memory/context, storage, and access control—a lifecycle and runtime view at system level; reports speedups from centralized resource management.

Langchain: A tool for building (simple) agents

"Agents combine **language models** with **tools** to create systems that can **reason about tasks**, decide **which tools to use**, and **iteratively work towards solutions**."

LangChain

The platform for reliable agents.



LangChain is a framework for building LLM-powered applications. It helps you chain together interoperable components and third-party integrations to simplify AI application development — all while future-proofing decisions as the underlying technology evolves.

```
pip install -U langchain
```

https://github.com/langchain-ai/langchain

LangChain supports two major paradigms (1/2)

1) Chains – deterministic, linear workflows

- A Chain is a fixed pipeline of LLM calls, prompt templates, retrievers, or other components
- The developer defines *the order* and *the control flow*
- The model just fills in text or returns a result: It doesn't decide what happens next

```
from langchain import LLMChain
chain = LLMChain(llm=gpt4, prompt=prompt)
chain.run("Summarize this document.")
```

- The system runs a **single pass**: input → prompt → LLM → output
- There is **no looping or tool selection**

LangChain supports two major paradigms (2/2)

2) Agents – dynamic, ReAct-style loops

- An Agent wraps one or more tools; the model decides what to do next
- The LLM emits reasoning steps like:

```
Thought: I need to look this up.
Action: Search
Action Input: "CEO of Tesla"
```

- LangChain parses these, executes the matching Tool.func(), injects the "Observation" back into the prompt, and calls the model again
- This forms a **Reason → Act → Observe → Repeat** loop until model outputs Final Answer

Workflows vs. agents

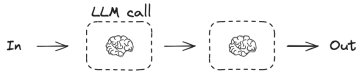
https://langchain-ai.github.io/langgraph/tutorials/workflows/
https://www.anthropic.com/engineering/building-effective-agents

Workflows: Systems where LLMs and tools are orchestrated through predefined code paths.

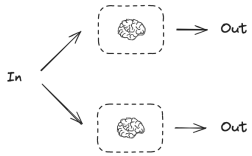
Agents: Systems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

Workflows

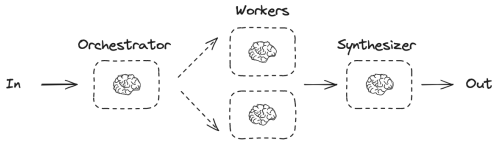
Prompt Chaining



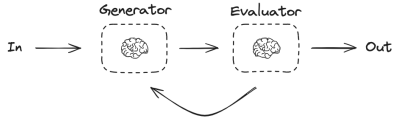
Parallelization



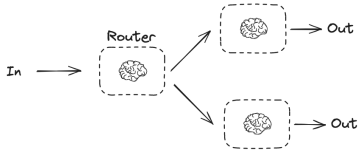
Orchestrator-Worker



Evaluator-optimizer



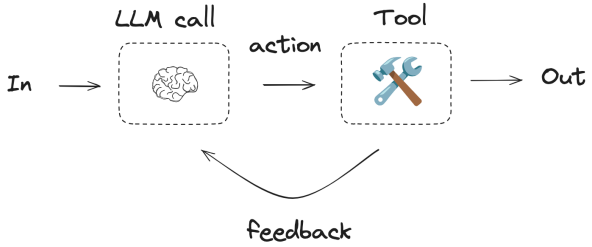
Routing



LLM is embedded in predefined code paths

LLM directs control flow through predefined code paths

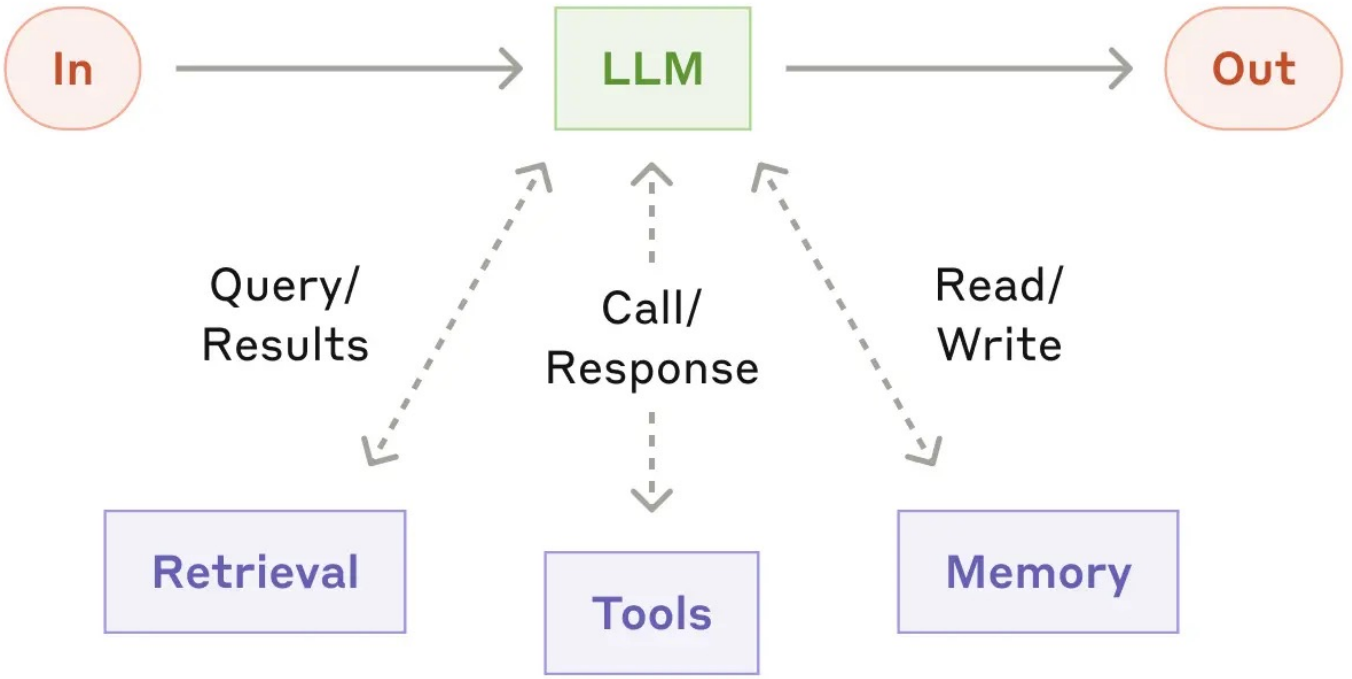
Agent



LLM directs its own actions based on environmental feedback

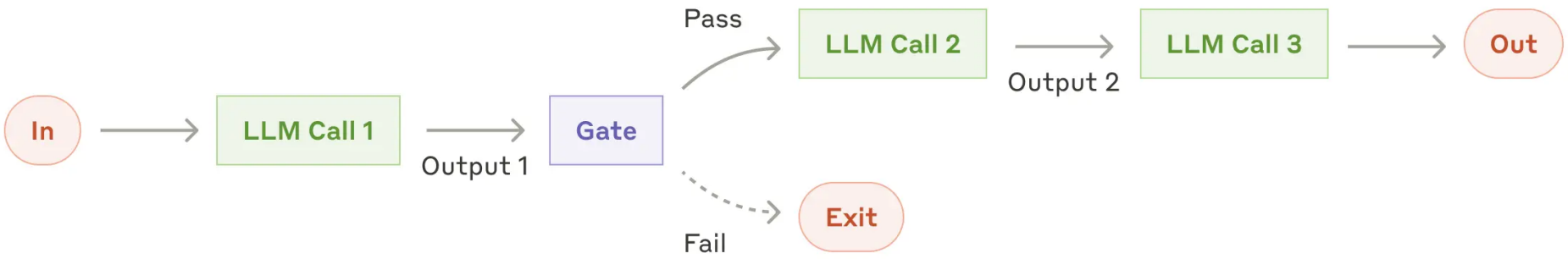
Workflows

Building block: The augmented LLM



Workflows

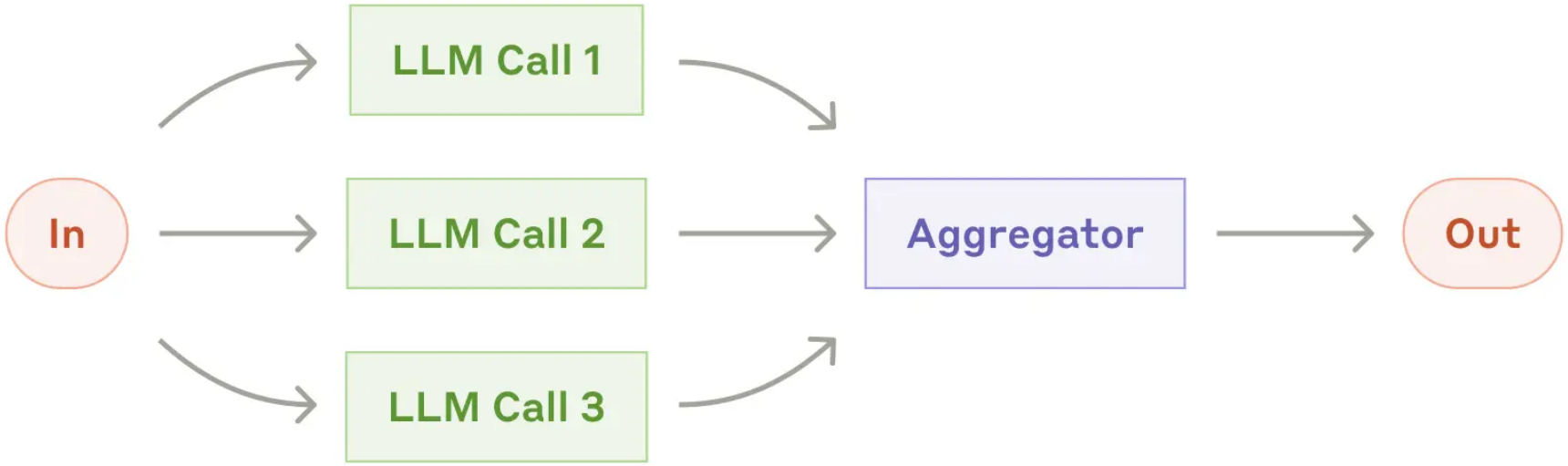
Prompt chaining



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

Workflows

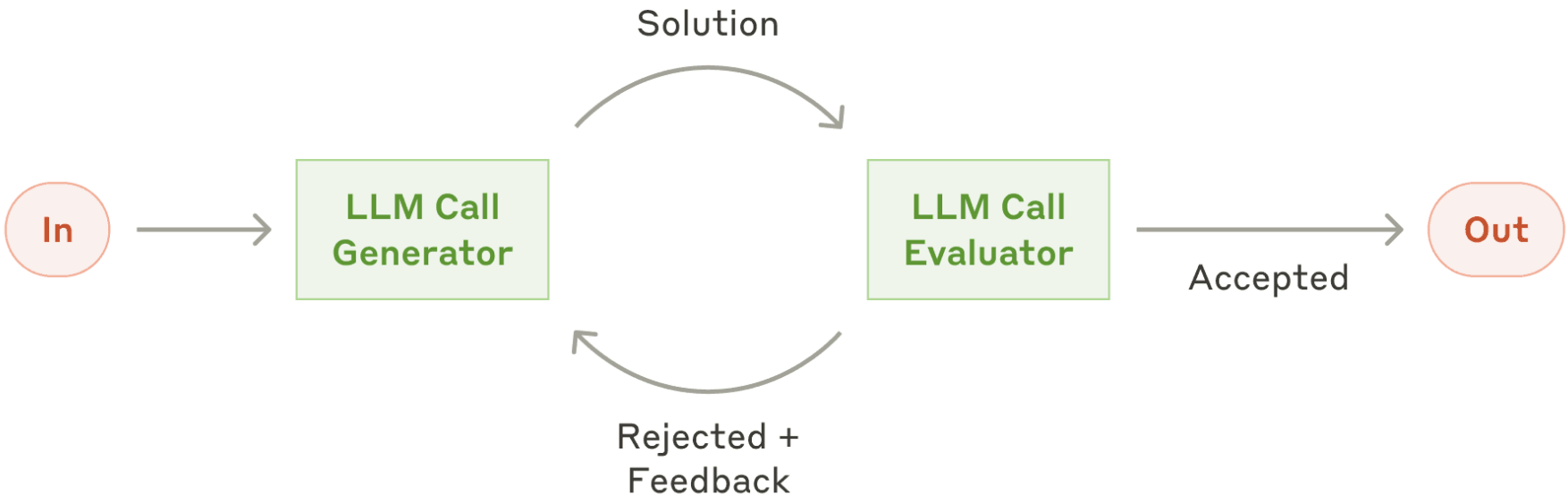
Routing



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.

Workflows: Evaluator-optimizer

Routing



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

Built-in chains: e.g., “summarize chain”

```
from langchain import OpenAI
from langchain.chains.summarize import load_summarize_chain
from langchain.docstore.document import Document

llm = OpenAI(temperature=0)
docs = [Document(page_content="LangChain helps build modular LLM applications.")]

chain = load_summarize_chain(llm, chain_type="map_reduce")
summary = chain.run(docs)
print(summary)
```

```python
def load_summarize_chain(
    llm: BaseLanguageModel,
    chain_type: Optional[str] = None,
    **kwargs: Any,
) -> BaseCombineDocumentsChain:
    """Load summarizing chains."""
    # If chain_type not provided, choose default
    chain_type = chain_type or detect_default_type(llm)
    if chain_type == "stuff":
        return StuffSummarizationChain(llm=llm, **kwargs)
    elif chain_type == "map_reduce":
        map_chain = load_summarize_chain(llm, chain_type="stuff", **kwargs)
        combine_chain = load_summarize_chain(llm, chain_type="stuff", **kwargs)
        return MapReduceSummarizationChain(llm=llm,
                                           map_chain=map_chain,
                                           combine_chain=combine_chain,
                                           **kwargs)
    elif chain_type == "refine":
        # iterate refine style
        return RefineSummarizationChain(llm=llm, **kwargs)
    else:
        raise ValueError(f"Unknown chain_type: {chain_type}")
```

Recall structure of a basic agent

- Initialize state and goals
- Repeat until termination:
  - **Sense**: Gather observations from environment
  - **Plan**: Evaluate goals and state, plan/select next action
  - **Act**: Execute chosen action on environment
  - **Learn**: Update internal state, memory, or model based on outcomes

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

Agents

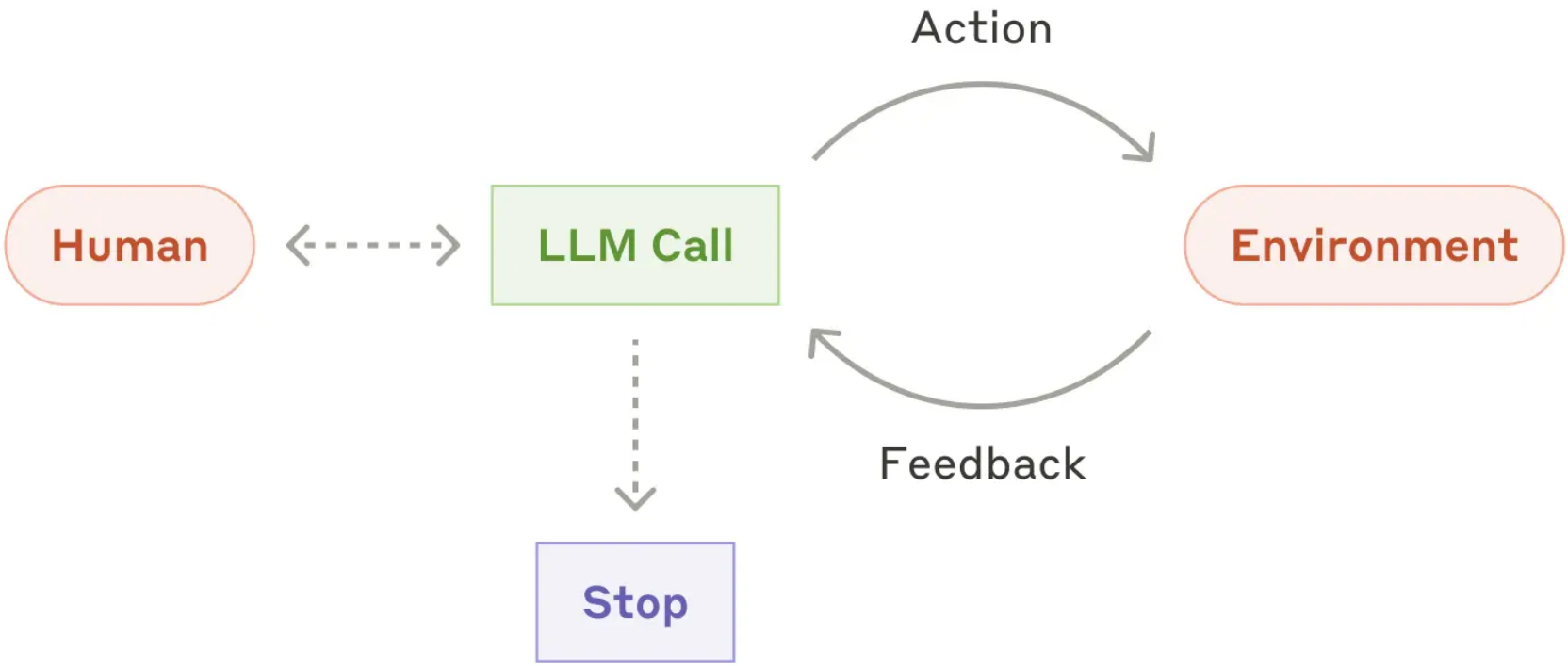
Agents begin their work with either a command from, or interactive discussion with, the human user.

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.

Agents



ReAct loop with LangChain create-agent

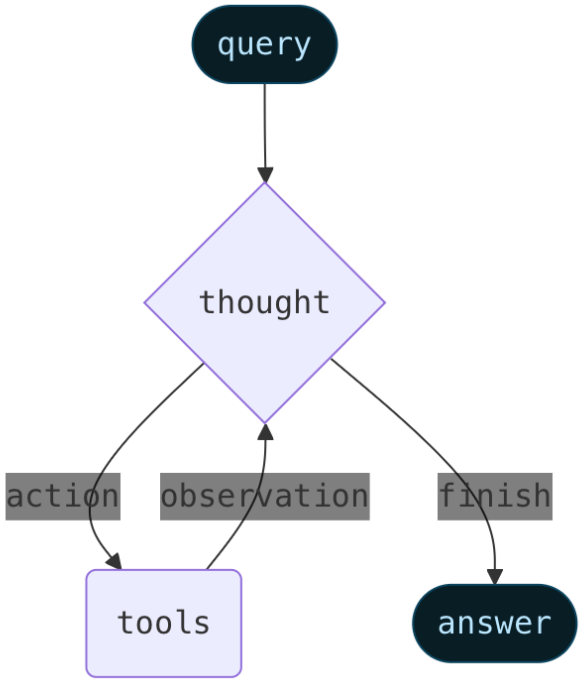
ReAct frames an agent's behavior as an interleaving of

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)



The ReAct agent

In a ReAct agent (AgentType.ZERO_SHOT_REACT_DESCRIPTION):

- The LLM generates Thought → Action → Action Input → Observation blocks
- Each Tool is advertised via its description
- The model *chooses* an action name (e.g., "Summarizer") and passes arguments (Action Input) as JSON-like text
- LangChain executes the corresponding Python function (func) and inserts the result into the next prompt as an “Observation”

Example: Summarize Document via ReAct Loop

```python
from langchain import OpenAI, LLMMathChain
from langchain.agents import Tool, initialize_agent, AgentType
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
from langchain.chains.summarize import load_summarize_chain

# ------------------------------------------------------------
# 1️⃣ PREPARE LLM AND DOCUMENT
# ------------------------------------------------------------
llm = OpenAI(temperature=0)

text = """
LangChain provides a standard interface for chains, a set of integrations with other to
and end-to-end chains for common applications such as summarization, question answering
and code analysis. It is modular and composable, enabling developers to build complex
applications from small, reusable components.
"""

# Convert to a Document object
docs = [Document(page_content=text)]

# ------------------------------------------------------------
# 2️⃣ DEFINE A SUMMARIZATION TOOL
# ------------------------------------------------------------
# You can wrap any chain as a "Tool" so the agent can decide to use it.

summarize_chain = load_summarize_chain(llm, chain_type="stuff")

def summarize_tool(input_text):
    """Summarize long text using an LLM chain."""
    document = [Document(page_content=input_text)]
    return summarize_chain.run(document)

summarize_tool_def = Tool(
    name="Summarizer",
    func=summarize_tool,
    description="Summarize a long text document into a concise summary."
)
```

```python
# ------------------------------------------------------------
# 3️⃣ DEFINE OTHER OPTIONAL TOOLS
# ------------------------------------------------------------
calc_chain = LLMMathChain.from_llm(llm)
calc_tool_def = Tool(
    name="Calculator",
    func=calc_chain.run,
    description="Perform basic math operations when needed."
)

tools = [summarize_tool_def, calc_tool_def]

# ------------------------------------------------------------
# 4️⃣ INITIALIZE A REACT AGENT
# ------------------------------------------------------------
agent = initialize_agent(
    tools=tools,
    llm=llm,
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,  # This is the ReAct loop
    verbose=True
)

# ------------------------------------------------------------
# 5️⃣ RUN THE AGENT
# ------------------------------------------------------------
query = "Summarize this document:\n" + text
result = agent.run(query)

print("\nFinal Answer:\n", result)
```

Simplified system prompt:

```
You can use the following tools:

Summarizer: Summarize a long text document into a concise summary.
Calculator: Perform basic math operations when needed.

Use the following format:

Thought: what you are thinking
Action: the tool to use
Action Input: the input to that tool
Observation: the result of the tool
... (repeat as needed)
Thought: I now know the final answer
Final Answer: your answer to the user
```

Step 1: LLM “reasons” in text

When you run

```python
agent.run("Summarize this document: [text]")
```

LangChain sends the question and tool list to the LLM.

The model’s training on the **ReAct pattern** (Reason + Act) guides it to follow that format:

```vbnet
Thought: I should summarize this text.
Action: Summarizer
Action Input: [the text]
```

Step 2: LangChain parser detects the “Action”

The `AgentExecutor` watches the model’s output.

It uses regex-based parsing to extract:

```python
action_name = "Summarizer"
action_input = "[the text]"
```

It then finds the corresponding `Tool` object by name and runs:

```python
tools["Summarizer"].func(action_input)
```

The tool’s result (the summary) is captured as:

```vbnet
Observation: LangChain is a modular framework for building LLM apps.
```

Step 3: Observation returned to LLM

LangChain appends that observation to the prompt and calls the model again:

```vbnet
Observation: LangChain is a modular framework ...
Thought: I now have the final answer.
Final Answer: [summary]
```

When the model writes “Final Answer,” the executor stops the loop.

ReAct loop with LangChain create-agent

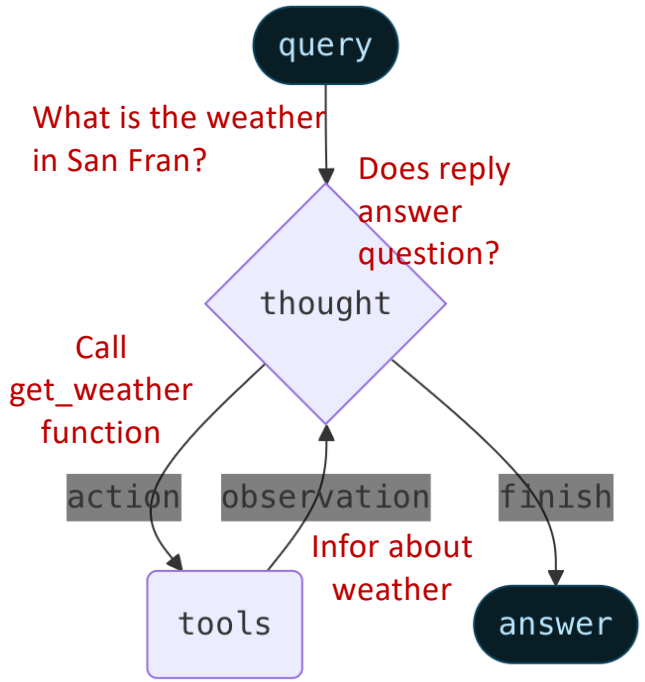
ReAct frames an agent's behavior as an interleaving of

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)



```python
# pip install -qU "langchain[anthropic]" to call the model

from langchain.agents import create_agent

def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

agent = create_agent(
    model="anthropic:claude-3-7-sonnet-latest",
    tools=[get_weather],
    prompt="You are a helpful assistant",
)

# Run the agent
agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather in sf"}]}
)
```

A simple agent that can answer questions

The agent has the following components:

- A language model (Claude 3.7 Sonnet)
- A simple tool (weather function)
- A basic prompt
- The ability to invoke it with messages

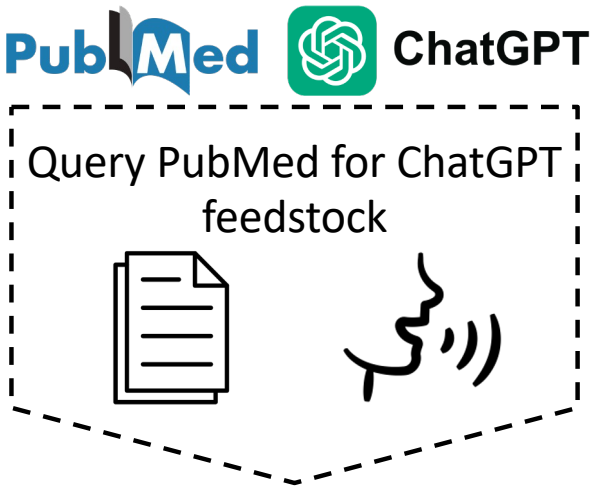
https://docs.langchain.com/oss/python/langchain/overview

```
from langchain.agents import create_agent
from langchain_core.messages import AIMessage

def extract_final_answer(run_output: dict) -> str:
    messages = run_output.get("messages", [])
    # Walk backward through messages to find the last AIMessage with content
    for msg in reversed(messages):
        if isinstance(msg, AIMessage):
            if msg.content and msg.content.strip():
                return msg.content.strip()
    return ""

def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

agent = create_agent(
    model="openai:o4-mini",
    tools=[get_weather],
    prompt="You are a helpful assistant",
)

r = agent.invoke({"messages": [{"role": "user", "content": "what is the weather in sf today"}]})

print(extract_final_answer(r))
```

“It’s sunny in San Francisco today! The skies are clear, and you can expect mild temperatures—around 68°F (20°C). Enjoy the beautiful weather!”

“Today in Chicago, it’s sunny with clear skies and pleasant conditions. Enjoy your day outdoors!”

Recall “Peptide Expert” example



Repeat until answer is satisfactory:

- Retrieve abstracts **A** from PubMed that reference specified **peptide**
- Use ChatGPT to build hypotheses:

  “Given **A**, on which organism is {**peptide**} acting?”

I.e.: query PubMed, and then call ChatGPT

Arvind Ramanathan, Priyanka Setty, et al.

ReAct loop with LangChain create-agent

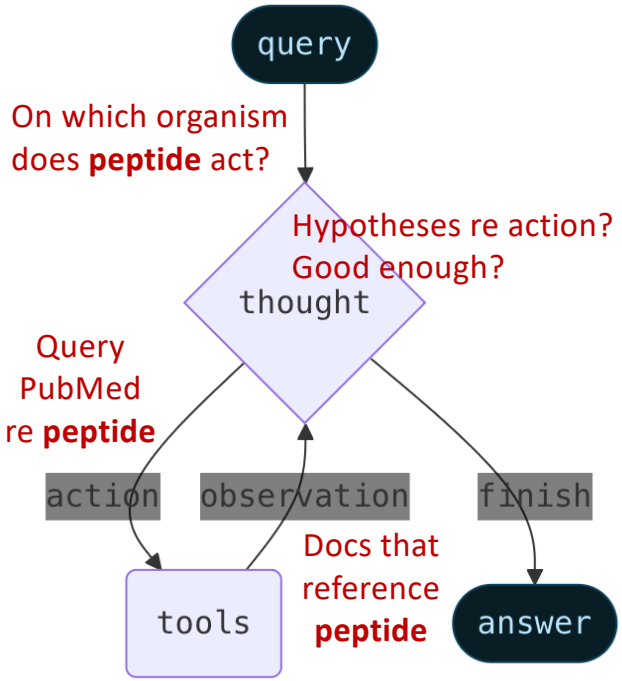
ReAct frames an agent's behavior as an interleaving of

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)



Build an agent with langchain

Next, build a practical weather forecasting agent that demonstrates key production concepts:

1. **Detailed system prompts** for better agent behavior
2. **Create tools** that integrate with external data
3. **Model configuration** for consistent responses
4. **Structured output** for predictable results
5. **Conversational memory** for chat-like interactions
6. **Create and run the agent** create a fully functional agent

https://docs.langchain.com/oss/python/langchain/quickstart

1. **Detailed system prompts**
2. Create tools
3. Model configuration
4. Structured output
5. Conversational memory
6. Create and run the agent

The system prompt defines your agent's role and behavior. Keep it specific and actionable:

```
system_prompt = """You are an expert weather forecaster, who speaks in puns.

You have access to two tools:

- get_weather_for_location: use this to get the weather for a specific location
- get_user_location: use this to get the user's location

If a user asks you for the weather, make sure you know the location. If you can tell from the question
that they mean wherever they are, use the get_user_location tool to find their location."""
```

1. Detailed system prompts
2. **Create tools**
3. Model configuration
4. Structured output
5. Conversational memory
6. Create and run the agent

Tools let a model interact with external systems by calling functions you define.

Tools can depend on **runtime context** and also interact with agent memory

Notice below how the get_user_location tool uses **runtime context**

```python
from dataclasses import dataclass
from langgraph.runtime import get_runtime

def get_weather_for_location(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

@dataclass
class Context:
    """Custom runtime context schema."""
    user_id: str

def get_user_location() -> str:
    """Retrieve user information based on user ID."""
    runtime = get_runtime(Context)
    user_id = runtime.context.user_id
    return "Florida" if user_id == "1" else "SF"
```

1. Detailed system prompts
2. Create tools
3. **Model configuration**
4. Structured output
5. Conversational memory
6. Create and run the agent

```python
from langchain.chat_models import init_chat_model

model = init_chat_model(
    "anthropic:claude-3-7-sonnet-latest",
    temperature=0
)
```

1. Detailed system prompts
2. Create tools
3. Model configuration
4. **Structured output**
5. Conversational memory
6. Create and run the agent

Optionally, define a structured response format if you need the agent responses to match a specific schema.

```python
from dataclasses import dataclass
from typing import Optional

# We use a dataclass here, but Pydantic models are
also supported.
@dataclass
class ResponseFormat:
    """Response schema for the agent."""
    # A punny response (always required)
    punny_response: str
    # Any interesting information about the weather
if available
    weather_conditions: Optional[str] = None
```

1. Detailed system prompts
2. Create tools
3. Model configuration
4. Structured output
5. **Conversational memory**
6. Create and run the agent

Add **memory** to your agent to maintain state across interactions. This allows the agent to remember previous conversations and context.

```
from langgraph.checkpoint.memory import InMemorySaver

checkpointer = InMemorySaver()
```

In production, use a persistent checkpointer that saves to a database. See **add and manage memory** for more details.

1. Detailed system prompts
2. Create tools
3. Model configuration
4. Structured output
5. Conversational memory
6. **Create and run the agent**

```python
agent = create_agent(
    model=model,
    prompt=system_prompt,
    tools=[get_user_location, get_weather_for_location],
    context_schema=Context,
    response_format=ResponseFormat,
    checkpointer=checkpointer
)

# `thread_id` is a unique identifier for a given conversation.
config = {"configurable": {"thread_id": "1"}}

response = agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather outside?"}]},
    config=config,
    context=Context(user_id="1")
)

print(response['structured_response'])
# ResponseFormat(
#   punny_response="Florida is still having a 'sun-derful' day! The sunshine is playing
'ray-dio' hits all day long! I'd say it's the perfect weather for some 'solar-bration'!
If you were hoping for rain, I'm afraid that idea is all 'washed up' - the forecast
remains 'clear-ly' brilliant!",
#   weather_conditions="It's always sunny in Florida!"
# )
```

```python
# Note that we can continue the conversation using the same `thread_id`.
response = agent.invoke(
    {"messages": [{"role": "user", "content": "thank you!"}]},
    config=config,
    context=Context(user_id="1")
)

print(response['structured_response'])
# ResponseFormat(
#   punny_response="You're 'thund-erfully' welcome! It's always a 'breeze' to help you stay 'current' with the weather. I'm just 'cloud'-ing around waiting to 'shower' you with more forecasts whenever you need them. Have a 'sun-sational' day in the Florida sunshine!",
#   weather_conditions=None
# )
```

Summary of the example

Langchain allows our agent to:

- **Perform tasks** in sequence
- **Understand context** and remember conversations
- **Use multiple tools** intelligently
- **Provide structured responses** in a consistent format
- **Handle user-specific information** through context
- **Maintain conversation state** across interactions

LangGraph

"LangGraph is a low-level orchestration framework for building, managing, and deploying long-running, stateful agents:

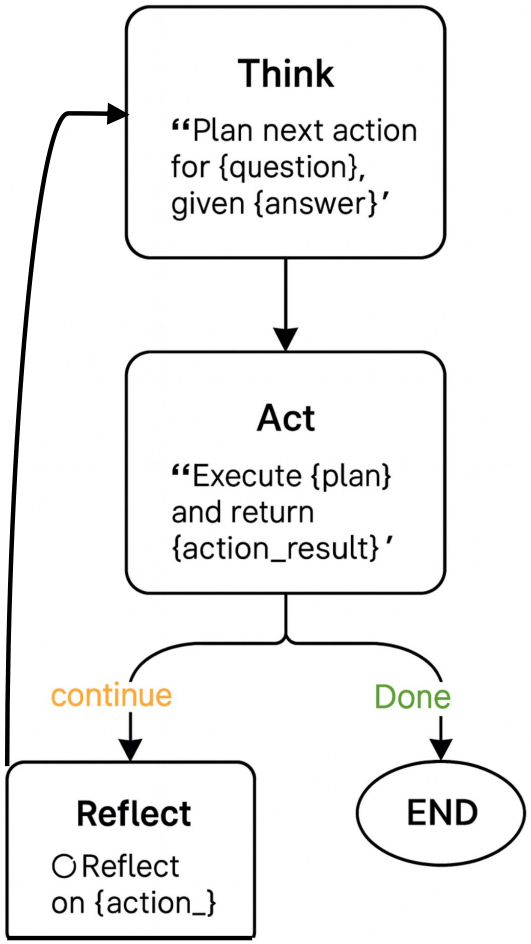
- **Durable execution**: Build agents that persist through failures and can run for extended periods, resuming from where they left off.
- **Human-in-the-loop**: Incorporate human oversight by inspecting and modifying agent state at any point.
- **Comprehensive memory**: Create stateful agents with both short-term working memory for ongoing reasoning and long-term memory across sessions.
- **Debugging with LangSmith**: Gain deep visibility into complex agent behavior with visualization tools that trace execution paths, capture state transitions, and provide detailed runtime metrics.
- **Production-ready deployment**: Deploy sophisticated agent systems confidently with scalable infrastructure designed to handle the unique challenges of stateful, long-running workflows."

https://docs.langchain.com/oss/python/langgraph/overview

ReAct-style “reasoning + acting + reflecting” loop in LangGraph

LangGraph lets you represent each step — “Reason”, “Act”, “Observe”, and “Reflect” — as nodes in a graph, connected by conditional edges that can loop until a termination condition is met.

```python
from langgraph.graph import StateGraph, END
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o")

def think(state):
    plan = llm.invoke(f"Think step: {state['question']} Current answer: {state['answer'
    return {"plan": plan}

def act(state):
    # Execute code or call a tool
    action_result = run_tool(state["plan"])
    return {"action_result": action_result}

def reflect(state):
    reflection = llm.invoke(f"Reflect on action: {state['action_result']}")
    if "complete" in reflection:
        return {"done": True}
    return {"done": False}

workflow = StateGraph()
workflow.add_node("think", think)
workflow.add_node("act", act)
workflow.add_node("reflect", reflect)

workflow.add_edge("think", "act")
workflow.add_edge("act", "reflect")
workflow.add_conditional_edges("reflect", lambda s: END if s["done"] else "think")

graph = workflow.compile(checkpointer="sqlite")
graph.invoke({"question": "Find the largest prime under 100."})
```


Think
“Plan next action for {question}, given {answer}’
Act
“Execute {plan} and return {action_result}’
continue
Done
Reflect
Reflect on {action_}
END

AutoGen for multi-agent LLM frameworks

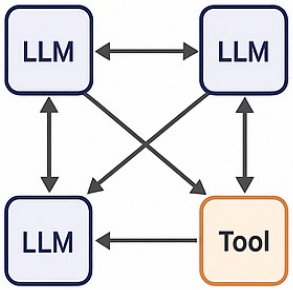
LangChain

Functional Chaining

A modular sequence of LLMs and tools invoked to complete tasks

Design Focus
Tool Integration

LangGraph

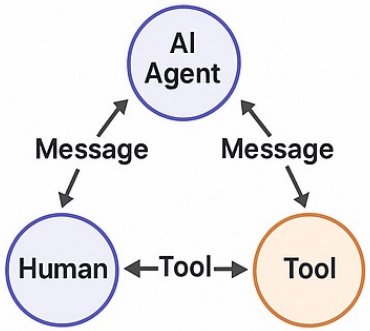


Stateful Graphs

Structured workflows formed from LLMs and tools with cyclical edges

Design Focus
Safety and Control

AutoGen



Conversational Agents

Agents communicate by exchanging messages in a conversation-like manner

Design Focus
Creativity and Collaboration

Multi-agent systems

Why multiple agents

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

Why multiple agents? (in more detail)

- **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 relating to multi-agent systems

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

Perils of multiple agents – Why systems matter

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 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 systems can 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

“AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation”

AutoGen2 allows developers to build LLM applications via multiple agents that can converse with each other to accomplish tasks.

AutoGen agents are customizable, conversable, and can operate in various modes that employ combinations of LLMs, human inputs, and tools.

Developers can also flexibly define agent interaction behaviors.

Both natural language and computer code can be used to program flexible conversation patterns for different applications.

https://arxiv.org/pdf/2308.08155

See also https://github.com/microsoft/agent-framework

AutoGen

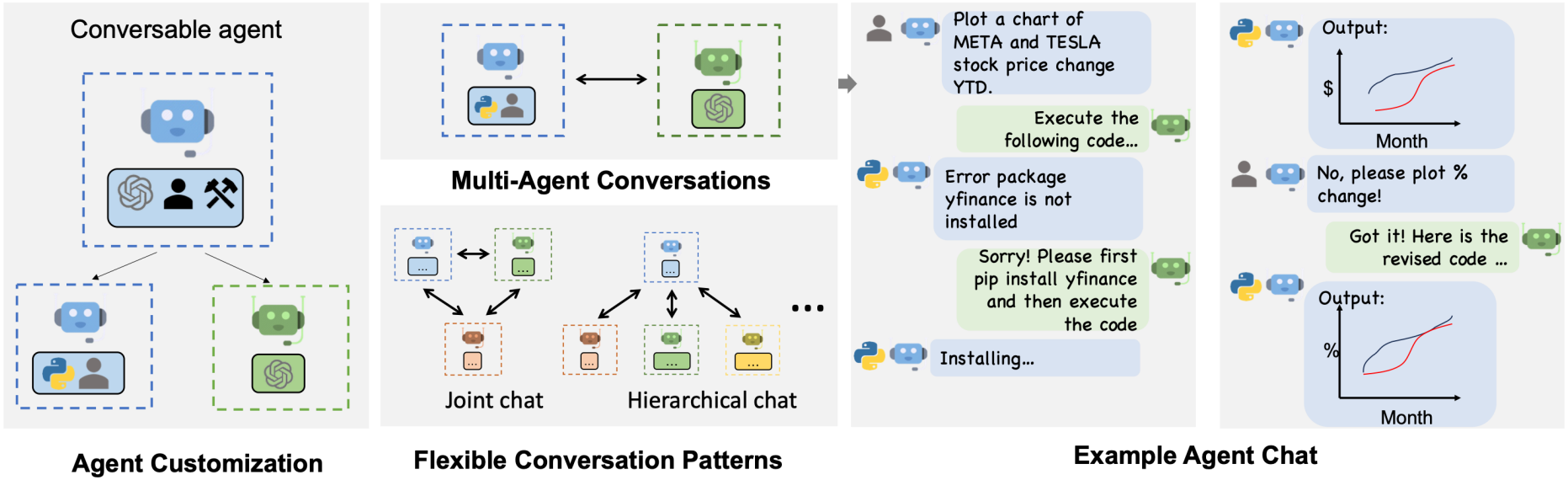


Figure 1: AutoGen enables diverse LLM-based applications using multi-agent conversations. (Left) AutoGen agents are conversable, customizable, and can be based on LLMs, tools, humans, or even a combination of them. (Top-middle) Agents can converse to solve tasks. (Right) They can form a chat, potentially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

https://arxiv.org/pdf/2308.08155

AutoGen: Multi-agent conversation

AutoGen enables **LLM-driven applications** composed of multiple cooperating agents that communicate via **structured conversations**. Each agent can represent a specialized role — such as planner, executor, critic, or user proxy — and interact using natural language or code.

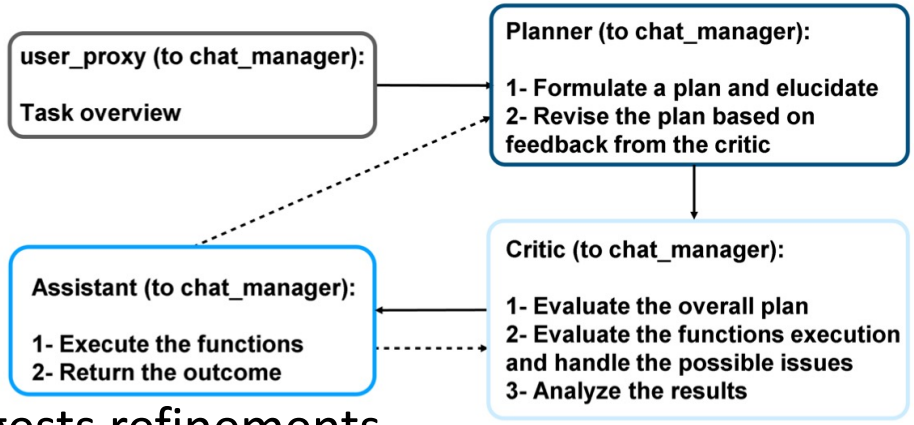
Conversation programming addresses **computation** (actions agents take to compute their response in a conversation) and **control flow** (order and conditions under which individual computations in the conversation are executed or evaluate)

AutoGen — Multi-Agent Conversation Framework

- **Key Features**
  - **Conversational Orchestration:** Tasks are decomposed and coordinated through dialogue rather than fixed pipelines.
  - **Flexible Agent Roles:** Agents can be LLMs, humans, or tool wrappers; each follows its own prompt and behavioral policy.
  - **Hybrid Programming:** Conversations can mix natural language, Python, and tool calls — enabling "conversation-as-code."
  - **Interoperability:** Integrates easily with frameworks like LangChain and supports tools, APIs, or custom function calls.
- **Architecture Overview**
  - **Planner Agent:** Defines goals, breaks down tasks, delegates subtasks.
  - **Executor / Assistant Agents:** Perform the work using tools or reasoning.
  - **Critic Agent:** Evaluates outputs, flags errors, and proposes refinements.
  - **User Proxy:** Represents human intent or feedback.

Example: “ProtAgents: Protein discovery via large language model multi-agent collaborations”

- **Planner**: Determines what steps to take
  - E.g., “design sequence → simulate → evaluate”
- **Assistant**: Can call tools
  - E.g. structure predictors, physics simulators
- **Critic**: Reviews results, spots errors, suggests refinements
- **User proxy**: Receives human instructions



The multi-agent system generates candidate sequences, simulates their mechanical / structural properties, critiques them, and iterates, aiming toward a design goal (e.g., a protein with a particular vibrational frequency or stiffness)

https://arxiv.org/pdf/2402.04268

🧩 Scenario: “Design a peptide inhibitor for a target protein.”

1️⃣ Planner (Coordinator)

> Let’s design a peptide that binds to Protein X’s active site and blocks catalysis.
> **Assistant**, please generate 3 candidate peptide sequences predicted to bind the site.
> **Critic**, once those are ready, evaluate them for stability and binding energy.

2️⃣ Assistant (Generator)

> I used the *AlphaDesign* tool and propose:
>
> 1. “**PepA:** GLY-LEU-SER-TYR-ASP”
> 2. “**PepB:** PHE-LYS-THR-TRP-ARG”
> 3. “**PepC:** VAL-TYR-GLY-ASP-ARG”
>
> Each was optimized for hydrophobic pocket fit. Results are stored in `candidate_peptides.csv`.

3️⃣ Critic (Evaluator)

> Running coarse docking simulations...
>
> - "**PepA**: ΔG = −6.8 kcal/mol (unstable secondary structure)"
> - "**PepB**: ΔG = −8.9 kcal/mol (stable α-helix) ✅"
> - "**PepC**: ΔG = −6.1 kcal/mol (poor binding surface)"
>
> Recommend **PepB** as the best candidate.

4️⃣ Planner

> Excellent. **Assistant**, please refine PepB to improve solubility.
> Add polar residues at termini while preserving core interactions.

5️⃣ Assistant

> Modified sequence: **SER-PHE-LYS-THR-TRP-ARG-ASP**
> Predicted solubility: ↑ 15% per *ProtSol* model.

6️⃣ Critic

> Re-simulated: ΔG = −8.7 kcal/mol; RMSD = 1.4 Å → still stable.
> Solubility confirmed. ✅

7️⃣ User Proxy

> That’s within target parameters. Proceed to wet-lab validation.

AIOS: LLM Agent Operating System

- Proposes a comprehensive system-level architecture for serving large-language-model-based agents
- introduces a **kernel-based architecture** that treats large language models (LLMs) and their associated tools as *first-class operating-system resources*.

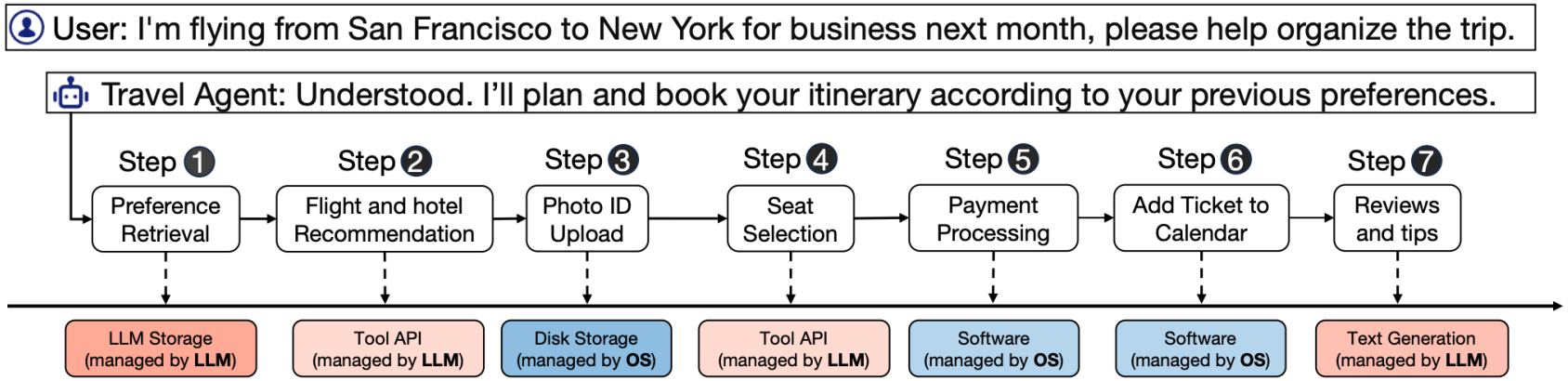


Figure 1: A motivating example of how an agent (i.e., travel agent) requires both LLM-related and Non-LLM-related (i.e., OS) services to complete a task, where color in red represents services related to LLM and color in blue represents services not related to LLM

The AIOS “Agent OS kernel”

- Instead of allowing each agent framework (e.g., LangChain, AutoGen, ReAct) to manage LLM calls, memory, and tools independently, **AIOS provides a central “Agent OS kernel”** responsible for:
  - **Scheduling** and multiplexing access to LLM resources
  - **Memory and storage management** for agent state and context
  - **Tool management** with validation and concurrency control
  - **Access control** and user-confirmation safeguards
  - **An SDK layer** (API interface) allowing agent developers to invoke “AIOS syscalls” instead of managing resources directly
- A three-layer stack:
  - **Application layer:** agent logic built via SDK APIs
  - **Kernel layer:** AIOS kernel managing LLM, memory, tools, access
  - **Hardware layer:** traditional OS and devices (CPU/GPU/memory)
- **LLM cores** are treated like CPU cores; **agent requests** become system calls; and a **scheduler** orchestrates them

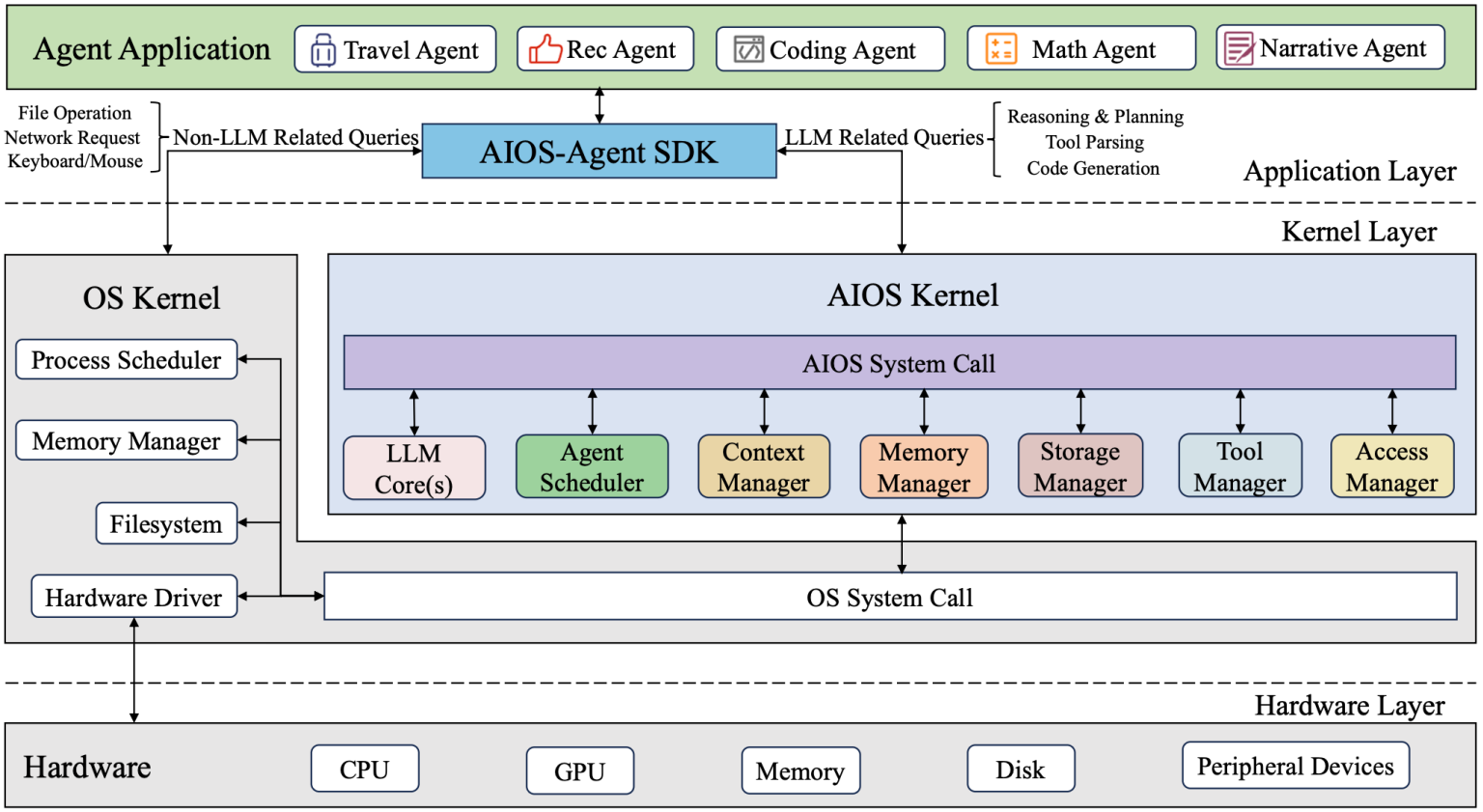


Figure 2: An overview of the AIOS architecture of distinct layers. Application layer facilitates the design and development of agent applications. Kernel layer manages core functionalities and resources to serve agent applications. Hardware layer controls and manages physical computing resources and devices to support kernel layer functionalities.

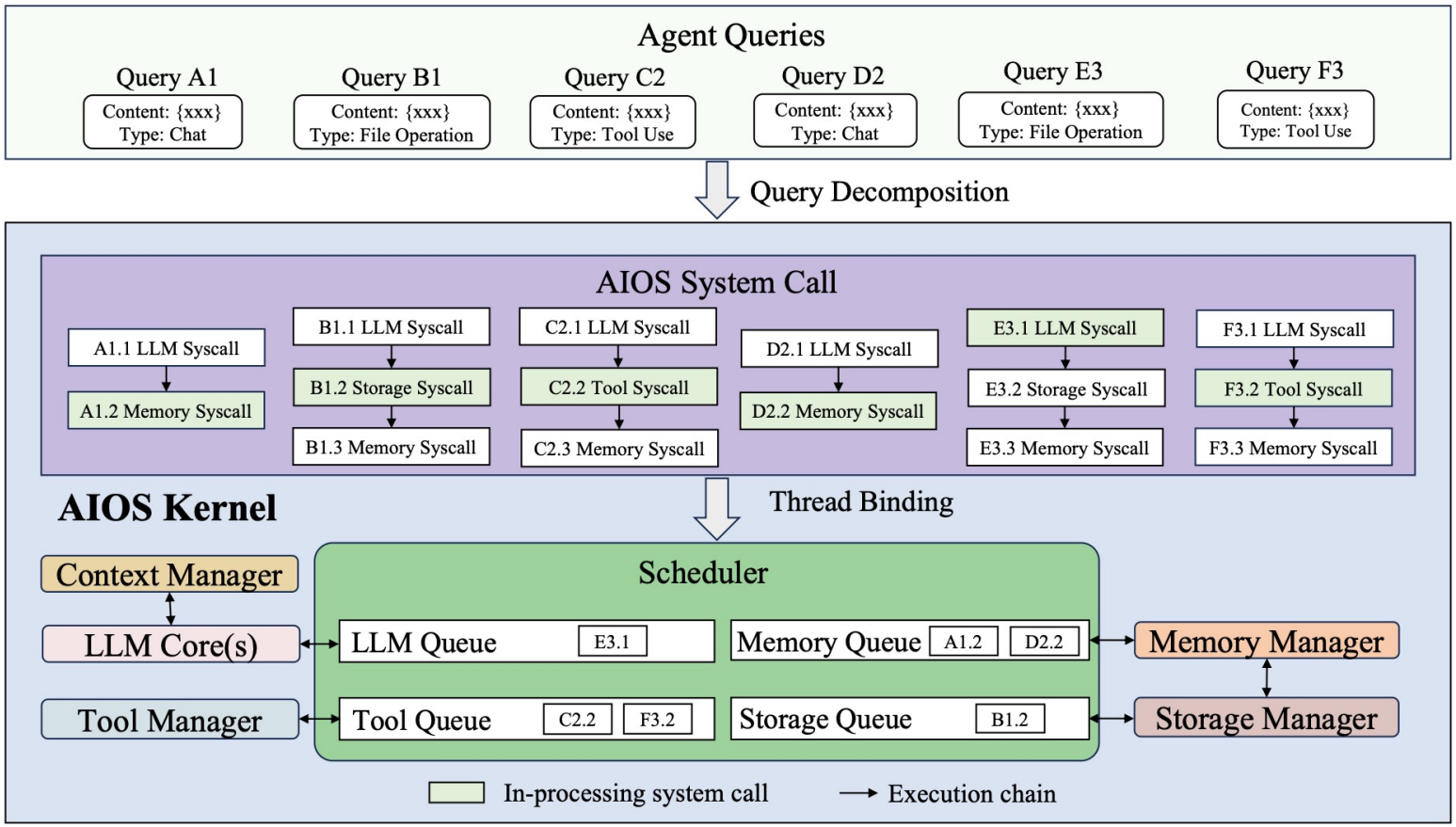
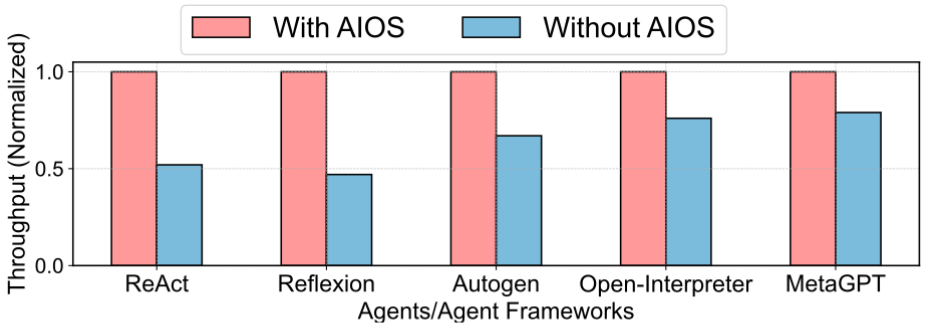


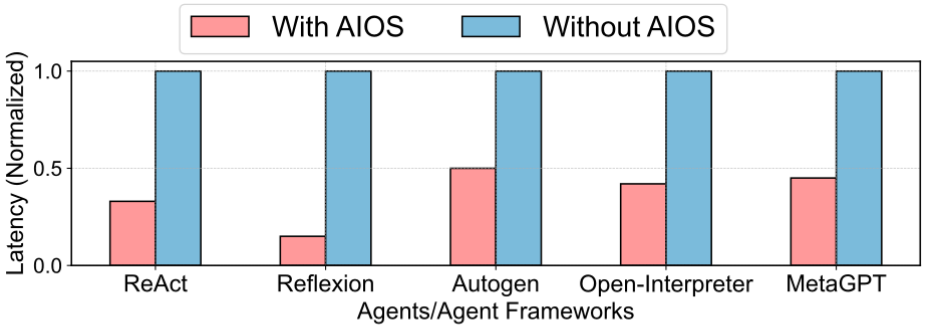
Figure 3: How agent queries are decomposed into AIOS system calls and how AIOS system calls are dispatched and scheduled. We omit the access manager module here as the access-related system calls will not be dispatched by the scheduler.

Evaluation

- **Throughput:** Up to 2.1× faster when multiple agents share limited GPU resources
- **Latency:** Reduced by scheduler control and pre-emptive LLM switching
- **Scalability:** Near-linear up to 2 000 agents
- **Correctness:** Context-switch BLEU / BERT = 1.0 → identical outputs after interruption

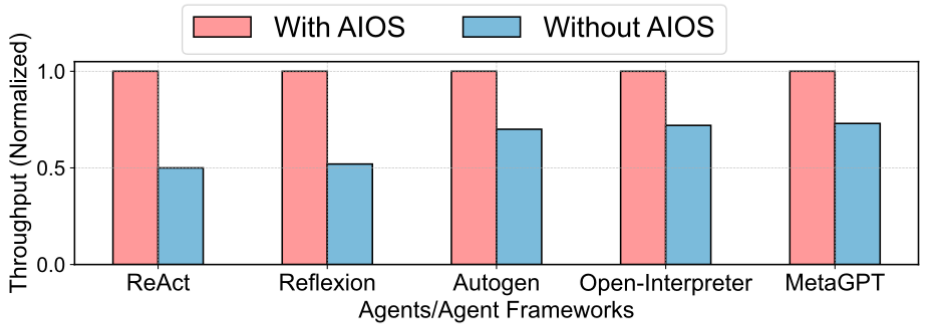


(a) Normalized throughput. Higher is better.

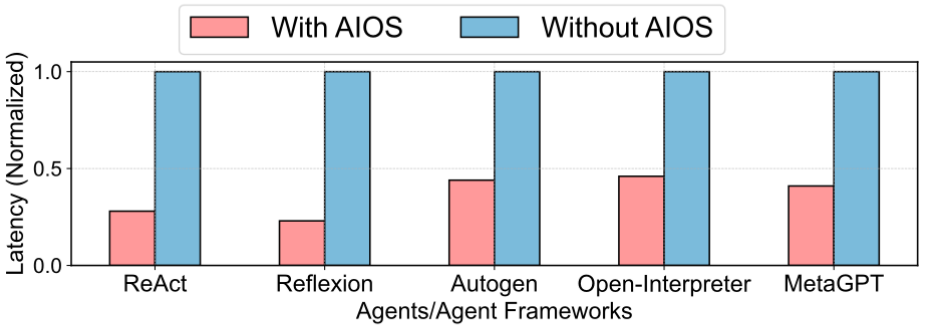


(b) Normalized latency. Lower is better.

Figure 6: Efficiency analysis on different agent frameworks evaluated on the Llama-3.1-8b model on the HumanEval benchmark.



(a) Normalized throughput. Higher is better.



(b) Normalized latency. Lower is better.

Figure 7: Efficiency analysis on different agent frameworks evaluated on the Mistral-7b model on the HumanEval benchmark.

Alternative architectural patterns

Pattern	Core Idea	Best Used When	Analogy
Blackboard System	Agents share a *common knowledge base* (the "blackboard") where each posts partial solutions or hypotheses. A control shell decides which agent acts next.	Problems are decomposable and benefit from *opportunistic collaboration* — e.g., perception-planning pipelines, scientific data analysis with shared intermediate results.	Shared whiteboard; each specialist adds insights until consensus.
Contract-Net Protocol (CNP)	A *manager agent* broadcasts a task; *contractors* bid based on capability or cost; manager awards the task.	Resource allocation and scheduling across distributed or heterogeneous agents — e.g., selecting which lab, robot, or compute node runs a sub-experiment.	Job auction or dynamic marketplace.
Actor / Graph Workflows	Agents (actors) communicate through *message passing* along explicit edges; control flow is deterministic and inspectable.	When reliability, debugging, or reproducibility are priorities — e.g., workflow orchestration (LangGraph, Ray, Dask) or multi-stage simulation chains.	Directed graph of cooperating workers.

Orchestration and Scale → Agents Meet HPC

- **Long-running workflows**: Scientific agents must persist across long experiments, spanning hours, days, or even weeks of compute.
- **Scalable inference**: Applications may require many 1000s of LLM instances
- **Fault tolerance**: Agents should resume gracefully after GPU/LLM/API failures via checkpointing and replay.
- **Scalable compute integration**: Agents must leverage HPC schedulers (Slurm, PBS) to launch simulations, transfer data, and monitor progress.
- **Hybrid orchestration**: Combine interactive reasoning loops with queued batch jobs — the "agent ↔ scheduler handshake."
- **System design challenge**: How do we preserve agent autonomy while embedding it in HPC-style reliability?

Three 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

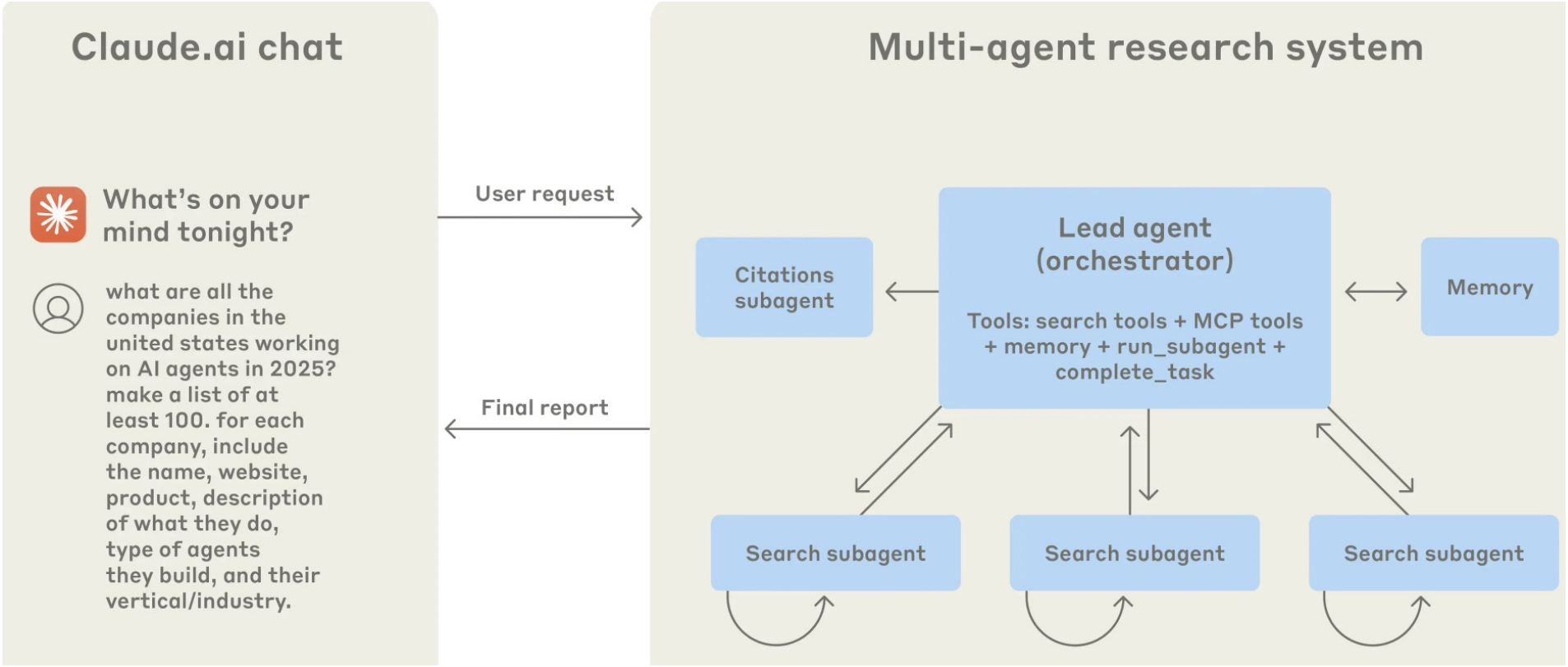
From LangChain to AIOS: A stack for agentic science

	LangChain	**LangGraph**	**AutoGen**	**AIOS**
Control	Programmatic	Graph-defined	Conversational	Kernel-scheduled
Scope	Single agent	Multi-step	Multi-agent	Multi-system

From LangChain to AIOS: A stack for agentic science

Layer	Framework / Concept	Focus	Analogy
Application	*LangChain*	Single-agent reasoning, tool use, ReAct loops	"App logic" — a script that calls the LLM and tools
Orchestration	*LangGraph*	Deterministic control-flow graphs, state persistence, human-in-loop	Workflow engine
Collaboration	*AutoGen*	Multi-agent conversation, planner–critic–executor roles	Distributed process management
Runtime / System	*AIOS*	Kernel for scheduling, memory, storage, access control	Operating system for agents

High-level architecture of Advanced Research



The multi-agent architecture in action: user queries flow through a lead agent that creates specialized subagents to search for different aspects in parallel.

https://www.anthropic.com/engineering/multi-agent-research-system

Benefits of a multi-agent system (Anthropic)

- “Multi-agent research systems excel especially for breadth-first queries that involve pursuing multiple independent directions simultaneously”
- “Multi-agent systems work mainly because they help spend enough tokens to solve the problem”
- “multi-agent systems excel at valuable tasks that involve heavy parallelization, information that exceeds single context windows, and interfacing with numerous complex tools”

https://www.anthropic.com/engineering/multi-agent-research-system

Lessons learned (Anthropic)

- Think like your agents
- Teach the orchestrator how to delegate
- Scale effort to query complexity
- Tool design and selection are critical
- Let agents improve themselves
- Start wide, then narrow down
- Guide the thinking process
- Parallel tool calling transforms speed and performance

https://www.anthropic.com/engineering/multi-agent-research-system

Discussion questions

- Where should **reasoning** stop and **system management** begin?
- What belongs inside the **agent** vs. in the **runtime**?
- How do we ensure **safety and reproducibility** as agents gain autonomy?
- Could scientific discovery platforms one day **run on AIOS-like kernels**?

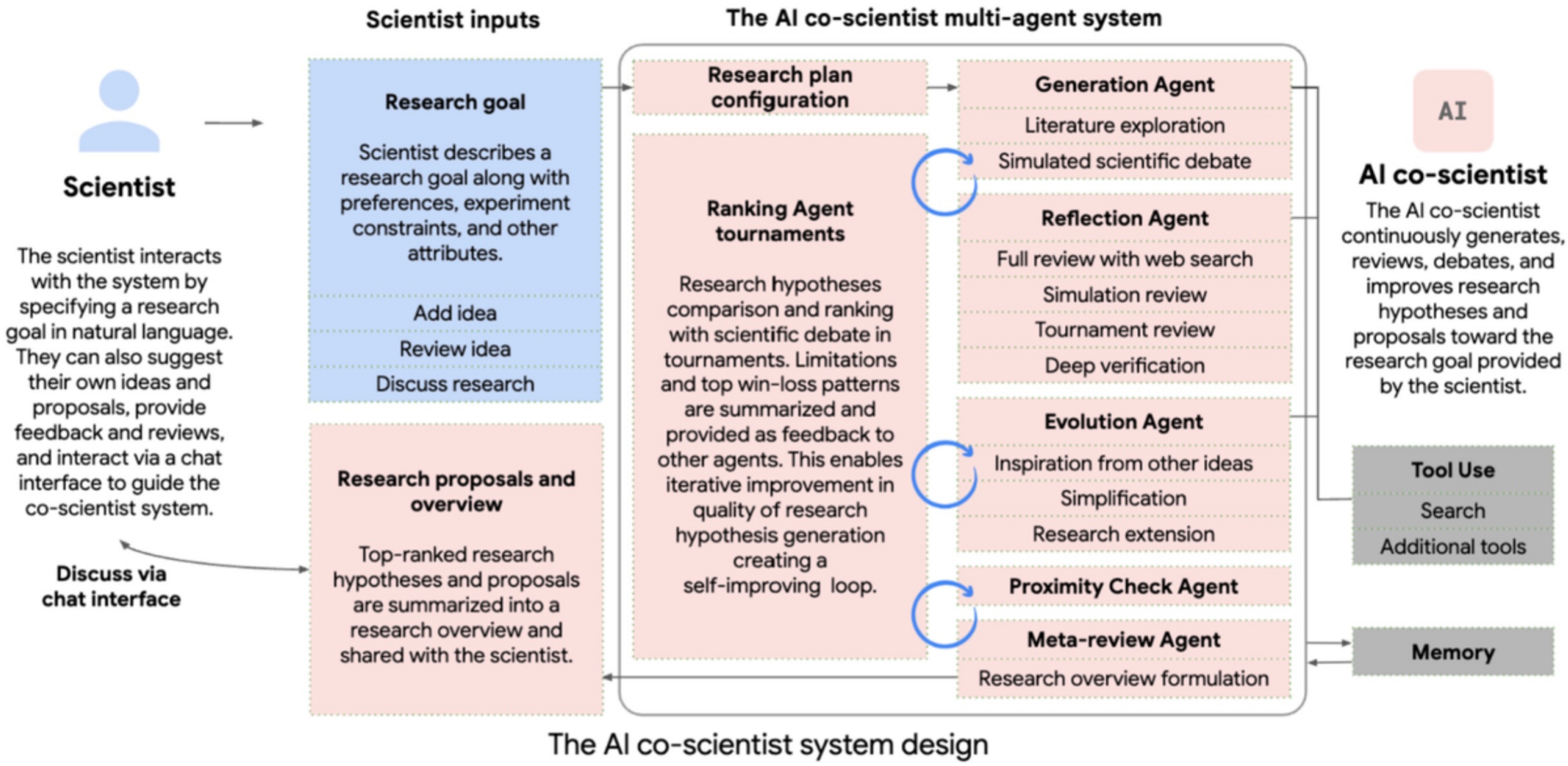
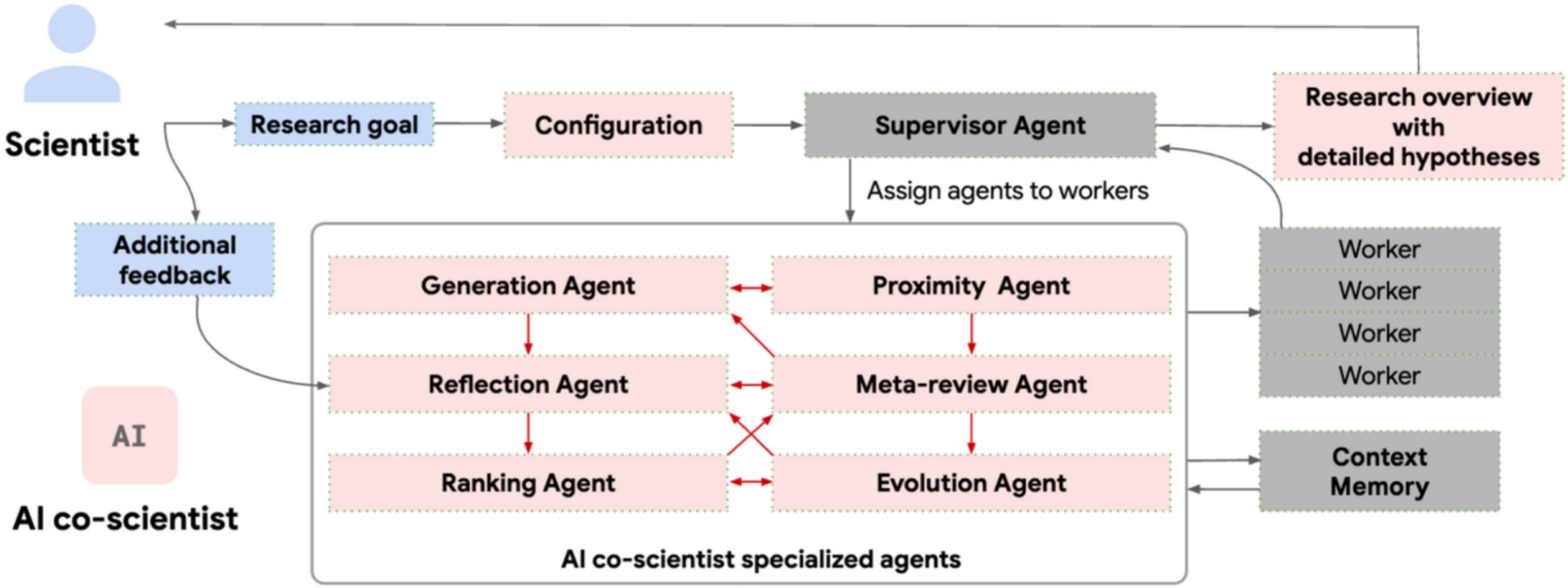
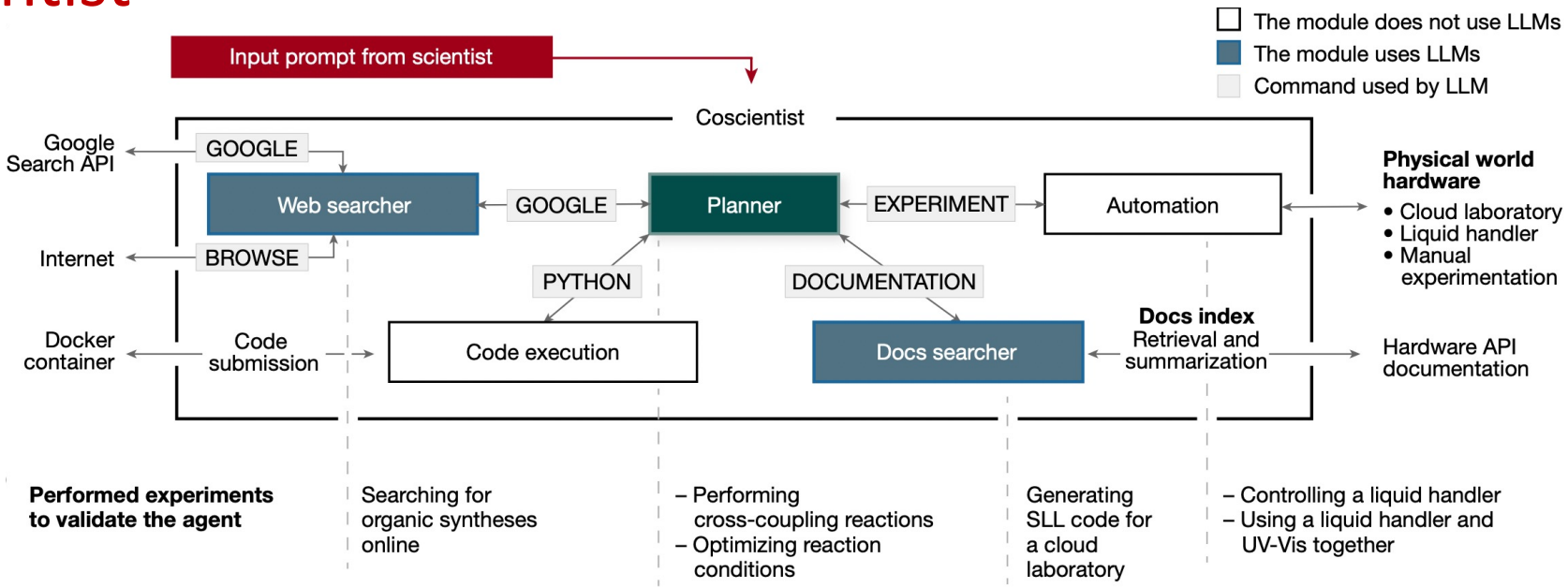


Illustration of the different components in the AI co-scientist multi-agent system and the interaction paradigm between the system and the scientist.



*AI co-scientist system overview. Specialized agents (**red boxes**, with unique roles and logic); scientist input and feedback (**blue boxes**); system information flow (**dark gray arrows**); inter-agent feedback (**red arrows** within the agent section).*

Coscientist



https://doi.org/10.1038/s41586-023-06792-0

```
You have a source plate with three samples in unknown positions (yellow, blue, and red), each potentially
located in one of the wells (A1, B1, and C1). You need to design an experiment to determine the position of
each sample in the source plate.    <setup description>
```

User prompt

Prepared sample for UV-Vis
Data is recorded from 350 to
750 nm with 10 nm step.

```
First, I will transfer a small amount of each sample from the source plate to separate wells in the target
plate. Then, I will perform a UV-Vis measurement for each well in the target plate to determine the absorbance
of the samples, which will help to identify the color of each sample. Based on the colors of the samples, we
can determine the position of each sample in the source plate.

EXPERIMENT
from opentrons import protocol_api
...

# Perform UV-Vis measurement on the target plate
UVVIS plate 1
```

In the code the Agent moves liquids from A1, B1, and C1 to A1, A2, and A3. Then it dilutes them.

Planner's output

```
UV-Vis data from plate 1 is ready. You can access it using PYTHON, it will be in the file named
'plate1.npy'. You will have NumPy installed. The file contains NumPy array of shape (8, 12, 41). There are
41 wavelengths from 350 to 750 with 10 nm step. Think about how would different colors absorb first.
```

Automated answer
+ guiding prompt

```
Different colors absorb light at different wavelengths. Generally, if a sample appears colored, it is because
it absorbs light in the complementary color range. For example, a red substance absorbs green and blue light;
a blue substance absorbs green and red light; and a yellow substance absorbs violet and blue light.

First, I will load the UV-Vis data from the file 'plate1.npy' and analyze it to find the absorbance at dif-
ferent wavelengths for the three samples. Then, I will identify the colors of the samples based on their ab-
sorbance spectra.

PYTHON
import numpy as np
...
print(samples_max_wavelength_nm)
```

```
[530, 430, 630]
```

Automated answer

```
The maximum absorbance wavelengths for the three samples are 530 nm, 430 nm, and 630 nm. Based on the absor-
bance spectra, we can deduce their colors:

1. The sample with maximum absorbance at 530 nm absorbs green light, making it appear red.
2. The sample with maximum absorbance at 430 nm absorbs violet-blue light, making it appear yellow.
3. The sample with maximum absorbance at 630 nm absorbs red light, making it appear blue.
```

https://doi.org/10.1038/s41586-023-06792-0