

Tool Calling, Frameworks, & MCP

Shreya Havaladar

Announcements

- **Grading for HW 3 will be demo-based**
 - Aaditya's OH on 11/6, 11/13, or 11/20
 - In-class on 11/16

IMPORTANT: make sure you are able to download your submitted zip file and launch your agent without issues.

Today's Lecture

1. **Tool Calling**
2. Frameworks
3. MCP

The Agent Stack

LLM	Model weights. Predicts tokens, including tool requests. Stateless and cannot act on its own.	Gemini, GPT, Claude
Runtime	Code that runs the loop: calls the model, runs its tool calls, enforces limits, keeps state.	A while loop around the Gemini SDK
Harness	A runtime built for a job: prompts, tools, context, policies. Everything but the weights.	Coding agents with shell, files, planning, memory
Framework	A library for building runtimes: loops, state, retries, tracing.	LangGraph, CrewAI, PydanticAI
Protocol	A standard for how separate systems discover and call each other.	MCP

Key failures of plain prompting

- Stale knowledge
 - *“Who was the best figure skater at the 2026 winter olympics?”*
- Inability to take actions
 - *“Can you check if this code compiles?”*
- Inability to access private or real-time context
 - *“Check my calendar and see if I am free tomorrow”*

Tool Calling fixes these issues

- *“Who was the best figure skater at the 2026 winter olympics?”*
 - → Web search tool updates stale knowledge
- *“Can you check if this code compiles?”*
 - → Code execution or calculator tools enable actions
- *“Check my calendar and see if I am free tomorrow”*
 - → Integration with email, calendar, Slack, etc. allows access to private context

What is a tool call?

Tool calling: The model emits a structured request indicating that an external capability should be invoked.

Important distinction:

- Model chooses or recommends a tool call
- Application runtime executes the tool call
- *Model does not directly execute code or APIs*

What is a tool call?

A tool call is just a model prediction

User: "What's the weather in London?"

MODEL OUTPUT

```
tool_call:  
  name: get_weather  
  Arguments:  
    city: "London"  
call_id: 42
```

What is a tool call?

MODEL OUTPUT

```
tool_call:  
  name: get_weather  
  Arguments:  
    city: "London"  
call_id: 42
```

RUNTIME

```
Get_weather(city="London")  
→ {"temperature": 18, "condition": "rain"}
```

What is a tool call?

RUNTIME

```
Get_weather(city="London")
```

```
→ {"temperature": 18, "condition": "rain"}
```

MODEL

```
"The temperature in London is 18°C and it is  
raining."
```

What is a tool call?

MODEL

"The temperature in London is 18°C and it is raining."

Takeaway:

The model predicts an action.

Your program decides whether and how that action actually happens.

Tool calling loop

1. User asks for something
2. Model decides whether a tool is needed
3. Model emits tool name + arguments
4. Runtime validates arguments
5. Runtime executes tool
6. Tool result returns to model
7. Model produces final answer or calls another tool

Tool calling loop

```
def run_agent(user_message):  
    messages = [user_message]  
    while True:  
        response = model.generate(messages, tools=tool_schemas)  
        if not response.tool_calls:  
            return response.text  
        messages.append(response)  
        for call in response.tool_calls:  
            args = validate(call.arguments)  
            result = tools[call.name](**args)  
            messages.append(tool_message(call.id, result))
```

Tool calling loop

```
def run_agent(user_message):  
    messages = [user_message]  
    while True:  
        response = model.generate(messages, tools=tool_schemas)  
        if not response.tool_calls:  
            return response.text  
        messages.append(response)  
        for call in response.tool_calls:  
            args = validate(call.arguments)  
            result = tools[call.name](**args)  
            messages.append(tool_message(call.id, result))
```

***User queries the model, model
generates a response***

Tool calling loop

```
def run_agent(user_message):  
    messages = [user_message]  
    while True:  
        response = model.generate(messages, tools=tool_schemas)  
        if not response.tool_calls:  
            return response.text  
        messages.append(response)  
        for call in response.tool_calls:  
            args = validate(call.arguments)  
            result = tools[call.name](**args)  
            messages.append(tool_message(call.id, result))
```

If the model has no tool call in its output, generation is returned to user

Tool calling loop

```
def run_agent(user_message):  
    messages = [user_message]  
    while True:  
        response = model.generate(messages, tools=tool_schemas)  
        if not response.tool_calls:  
            return response.text  
        messages.append(response)  
        for call in response.tool_calls:  
            args = validate(call.arguments)  
            result = tools[call.name](**args)  
            messages.append(tool_message(call.id, result))
```

***If the model HAS tool calls
in its output, every call is
validated + run***

Tool calling loop

```
def run_agent(user_message):  
    messages = [user_message]  
    while True:  
        response = model.generate(messages, tools=tool_schemas)  
        if not response.tool_calls:  
            return response.text  
        messages.append(response)  
        for call in response.tool_calls:  
            args = validate(call.arguments)  
            result = tools[call.name](**args)  
            messages.append(tool_message(call.id, result))
```

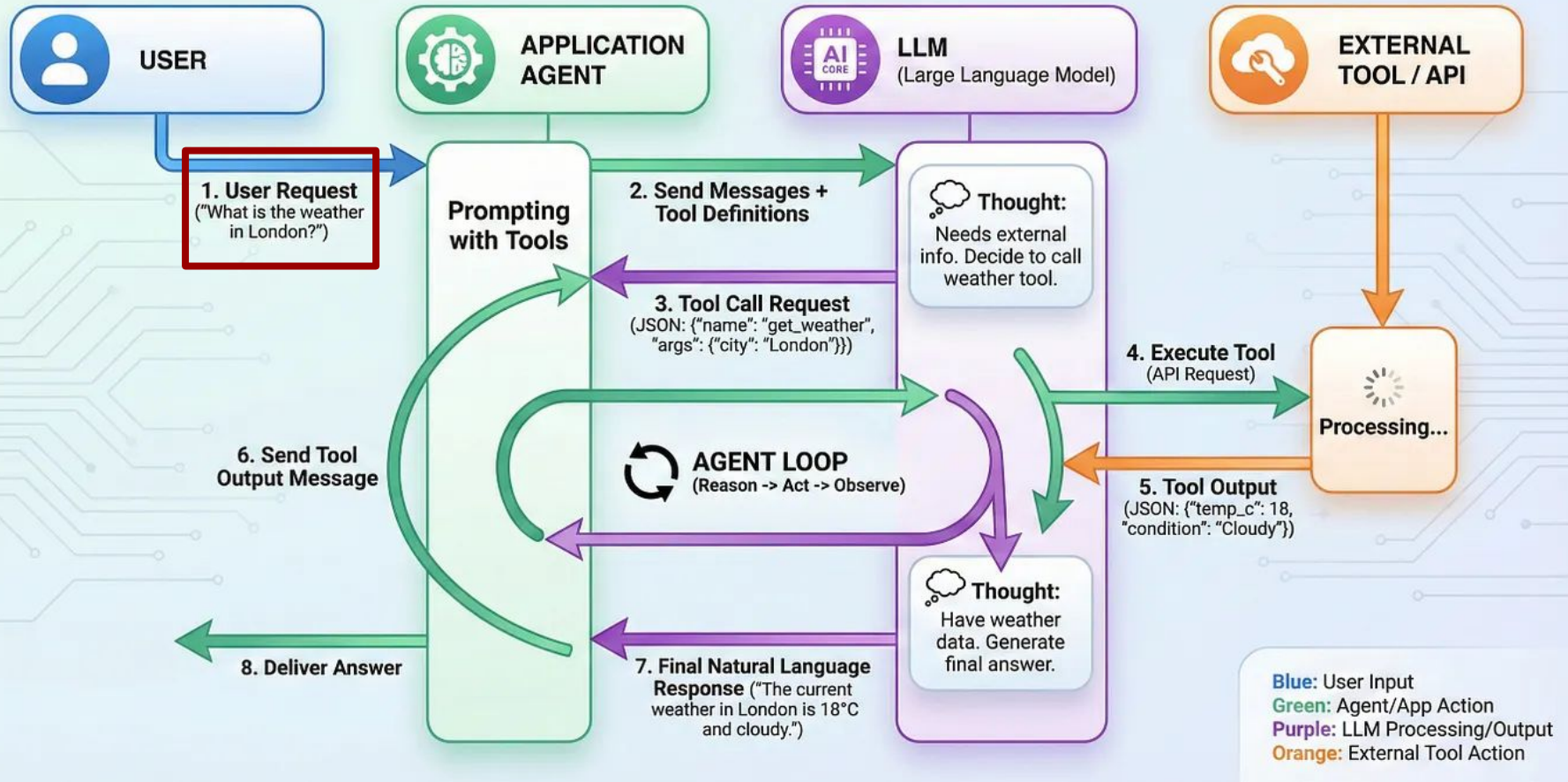
*Tool call outputs are then
used to re-generate the
response to the user*

Tool calling loop

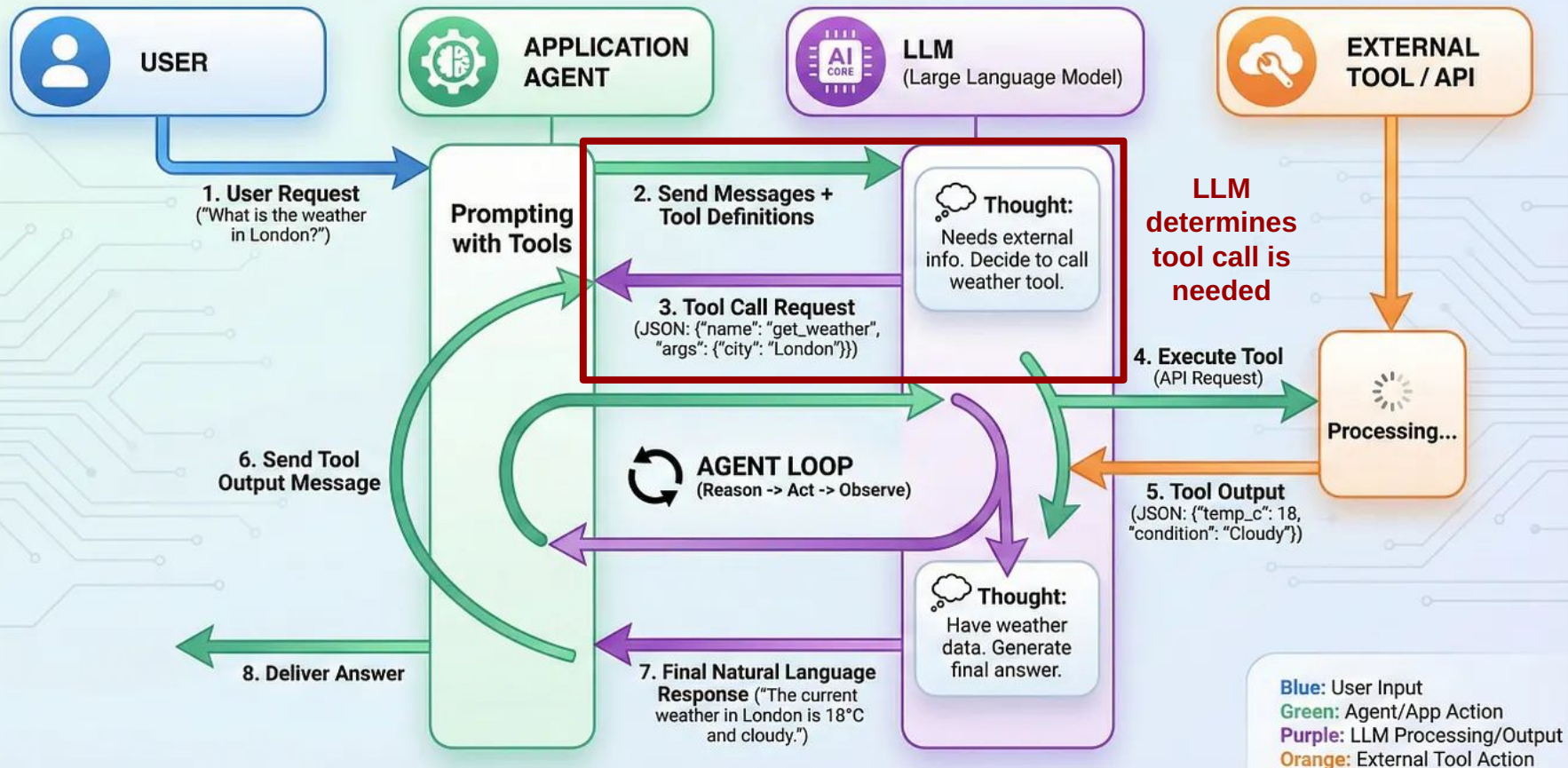
```
def run_agent(user_message):  
    messages = [user_message]  
    while True:  
        response = model.generate(messages, tools=tool_schemas)  
        if not response.tool_calls:  
            return response.text  
        messages.append(response)  
        for call in response.tool_calls:  
            args = validate(call.arguments)  
            result = tools[call.name](**args)  
            messages.append(tool_message(call.id, result))
```

After each tool result, the model runs again and may request more tools. The loop ends when a response has no tool calls.

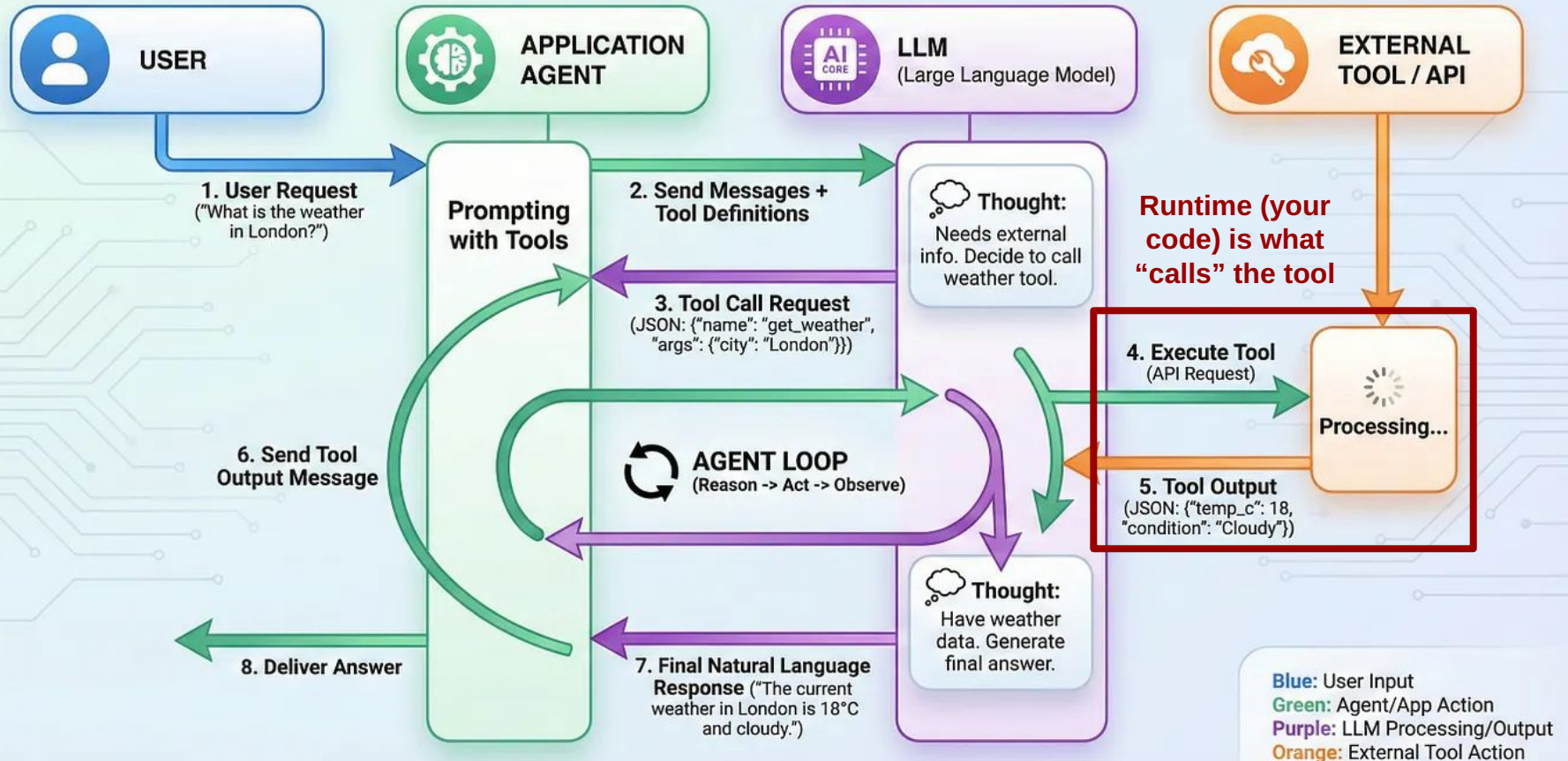
LLM Tool Calling & Agent Loop Visualized



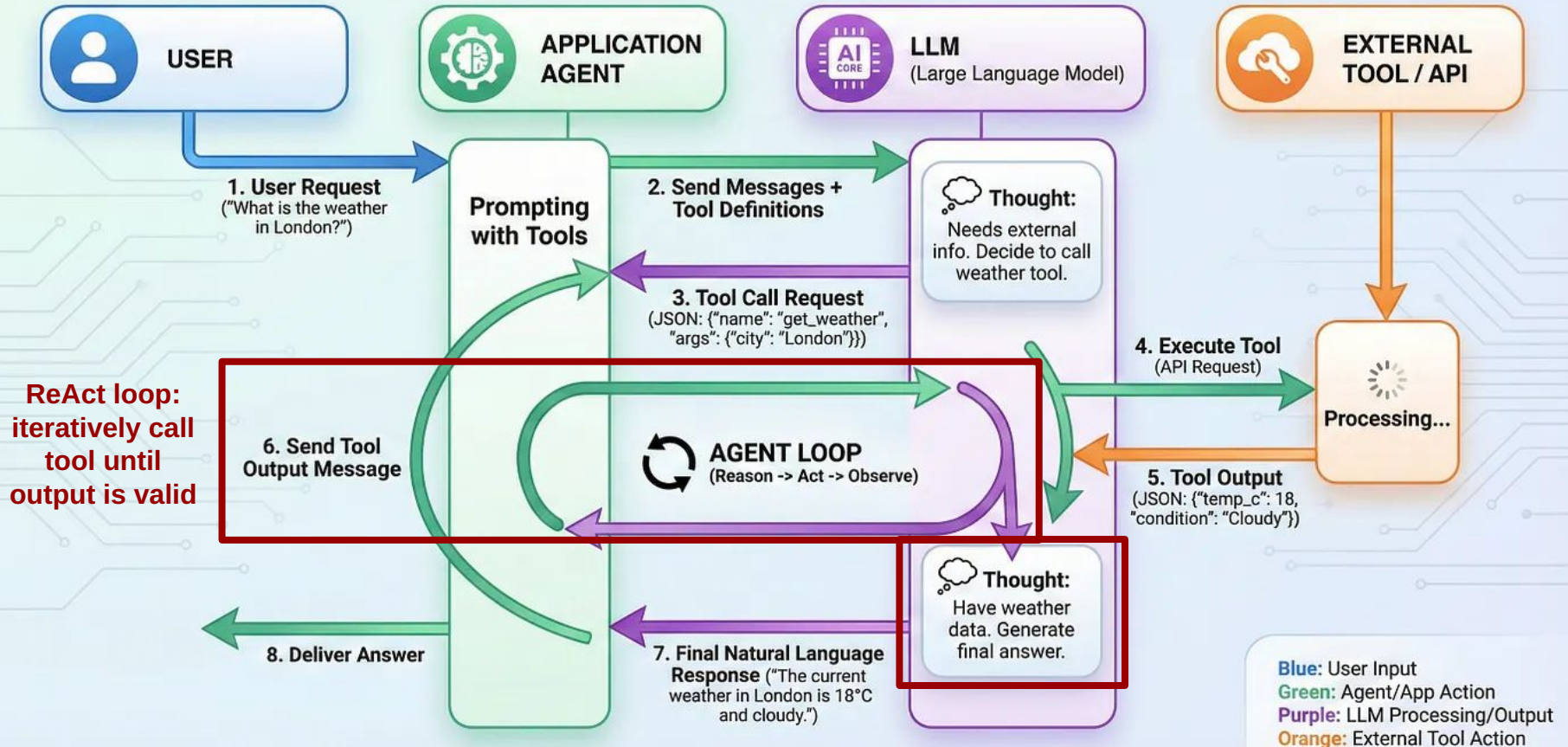
LLM Tool Calling & Agent Loop Visualized



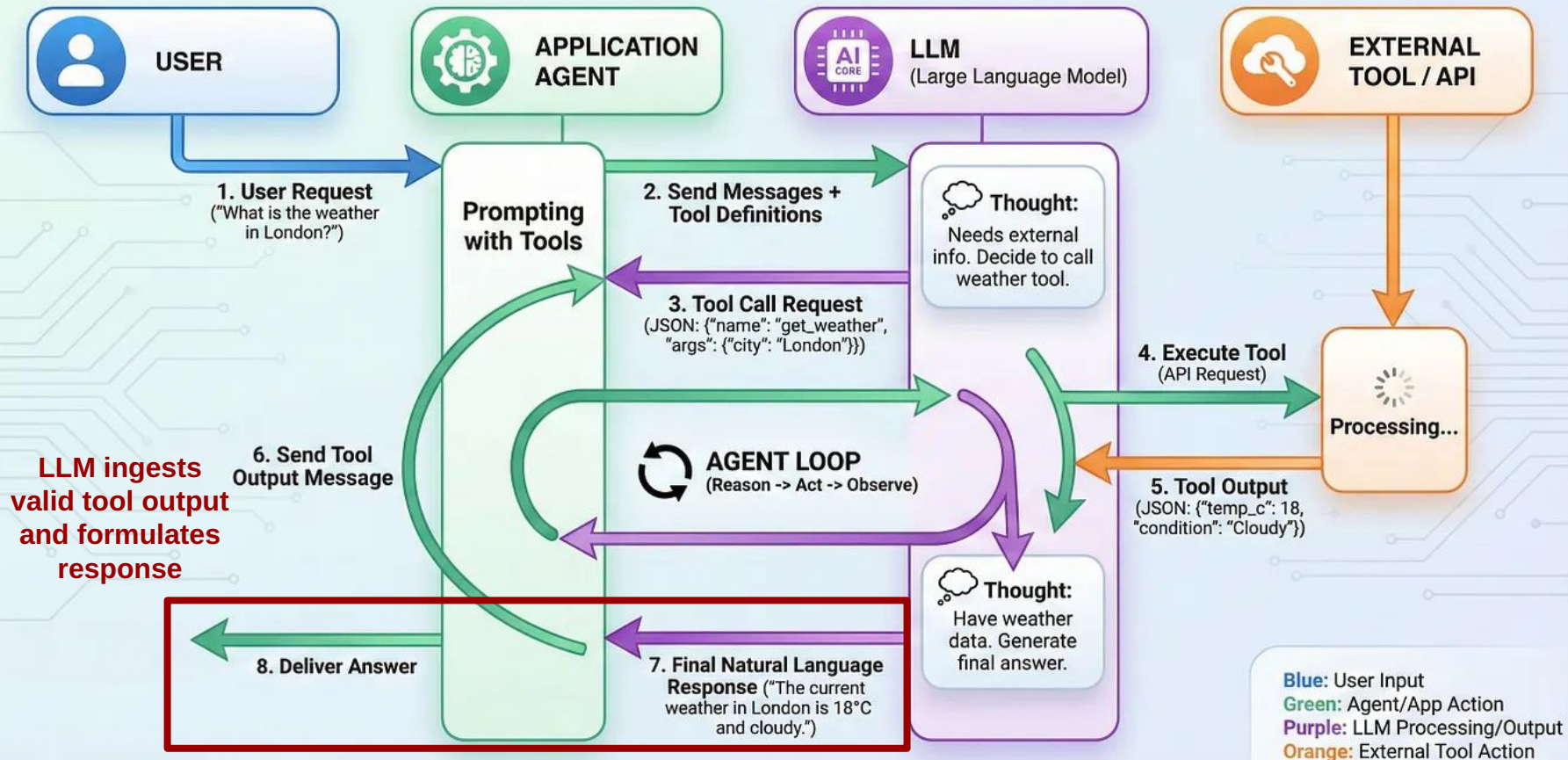
LLM Tool Calling & Agent Loop Visualized



LLM Tool Calling & Agent Loop Visualized



LLM Tool Calling & Agent Loop Visualized



Sequential vs parallel tool calls

Sequential

“What is the weather at the headquarters of Bloomberg?”

```
Find_company_hq( "Bloomberg" )
```

↓

```
"New York"
```

↓

```
get_weather( "New York" )
```

Second tool call depends on the first result.

Sequential vs parallel tool calls

Parallel

“Compare the weather in New York, London, and Tokyo.”

```
get_weather("New York") —┐  
get_weather("London")  —┤ → compare  
get_weather("Tokyo")  —┘
```

Sequential vs parallel tool calls

In order to call tools appropriately, the runtime must:

1. Understand dependencies
2. Understand concurrency
3. Handle failures

Function-calling APIs increasingly expose both compositional/sequential and parallel invocation patterns.

What counts as a tool?

A tool can be:

- Generic Python function
- Database query function
- Web search endpoint
- Calendar or email action
- Code executor
- Browser or shell interface
- Remote MCP-exposed capability
- Another LLM!

A “tool” is not defined by complexity. It is defined by the fact that the model can request it.

Benefits of Tool Calling

Tool calling enables:

- Access to current information
- Access to proprietary/private information
- Exact computation
- Structured changes to environment state
- Decomposition of complex tasks

Overall, tools improve agent reliability + accuracy

- Some tasks are better handled by deterministic systems than by text generation alone

Retrieval, structured output, and tool calling

	Retrieval	Structured output	Tool calling
Purpose	Get context	Format model answer	Perform intermediate operation
External action?	Read only	No	Possible
Model output	Text answer	Structured data	Tool + arguments
Example	Search documents	Extract {name, date}	Send email

Tool Schemas

Typical schema includes:

- tool name
- description
- Parameters
- parameter types
- required fields
- constraints

Descriptions matter because the model uses them to infer when to call the tool, what arguments to provide, and which tool is best (if multiple exist).

Good Tool Descriptions

Bad:

```
search(query)
```

Better:

```
'''
```

```
Searches internal documents for factual answers. Use when the  
user asks for policy, docs, or recent project context. Do not use  
for sending messages or editing files.
```

```
'''
```

```
search_documents(query: string, sources: list[str])
```

Common tool failure modes

At the tool level:

- LLM generates wrong tool arguments
- Lack of robust argument validation
- LLM may hallucinate tool names or parameters

At the system level:

- Wrong tool selection
 - Tools too granular → too many tools to parse through
 - Tools not granular enough → LLM may erroneously call tool
- Tools might open up security vulnerabilities

How to evaluate tool calling?

- **Tool selection**

- Did it call a tool when necessary?
- Did it pick the right one?

- **Arguments**

- Schema validity
- Semantic correctness

- **Trajectory**

- Necessary number of calls?
- Looping? Or Redundant work?

- **Execution**

- Success rate
- Latency
- Cost

- **End task**

- Did the user actually get the right outcome?

How to evaluate tool calling?

Final-answer accuracy can hide a broken trajectory.

Wrong tool → error → retry → error → right tool →
correct answer

Today's Lecture

1. Tool Calling
2. **Frameworks**
3. MCP

From tools to frameworks

Why not just call functions manually?

Because production agents need:

- Retries upon failure
- Branching
- Evaluation
- Human approval points
- Multi-step workflows
- etc.

Frameworks provide
structure + organization
when implementing
complex agents!

What Is an Agentic Framework?

An agentic framework is the foundational structure that governs how agents communicate, coordinate, reason, and act across systems.

It provides the structure for:

- Reasoning loops
- Message passing
- Tool orchestration
- Memory/state
- Workflow coordination
- Governance and observability

```
def run_agent(user_message):
    messages = [user_message]
    while True:
        response = model.generate(messages, tools=tool_schemas)
        if not response.tool_calls:
            return response.text
        messages.append(response)
        for call in response.tool_calls:
            args = validate(call.arguments)
            result = tools[call.name](**args)
            messages.append(tool_message(call.id, result))
```

***Without
framework***

```
agent = Agent( model=model, tools=tools,)agent.run(task)
```

***With
framework***

Frameworks increasingly overlap

Almost every framework can now:

- call tools
- maintain state
- use multiple models
- connect MCP servers
- implement workflows

To determine what framework is right for you, think about **what layer(s) of the agentic system it targets**

Layers of an agentic system

Think of an agentic system as having 4 layers:

1. **Application / UI** — user-facing
2. **Agent / workflow orchestration** — loops, routing, state
3. **Models + tools + MCP** — LLM provider, available tools
4. **Infrastructure** — DB, sandbox, logging

A framework usually spans 1-2 layers

Activity (20 min)



LangChain



LangGraph



smolagents

Get into groups of 3-4! Pick a slide and research your assigned framework.

Today's Lecture

1. Tool Calling
2. Frameworks
- 3. MCP**

Model Context Protocol (MCP)

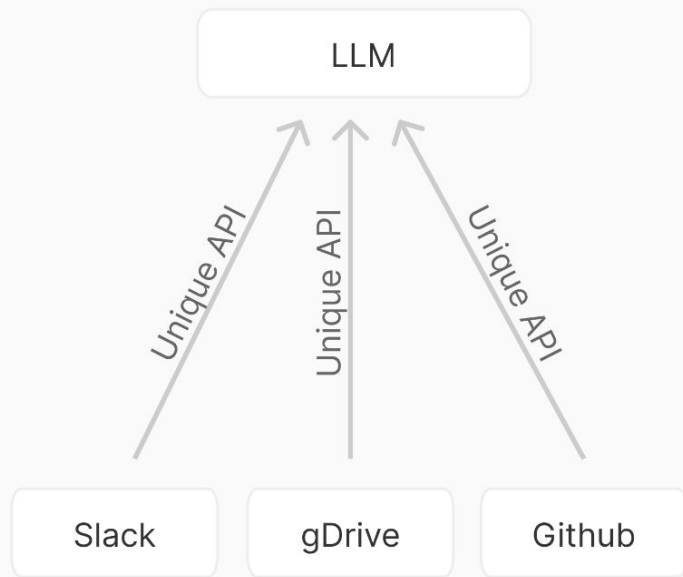
Before MCP:

- Every model + tool combo needed custom integration
- Integration burden scaled significantly as more tools were made

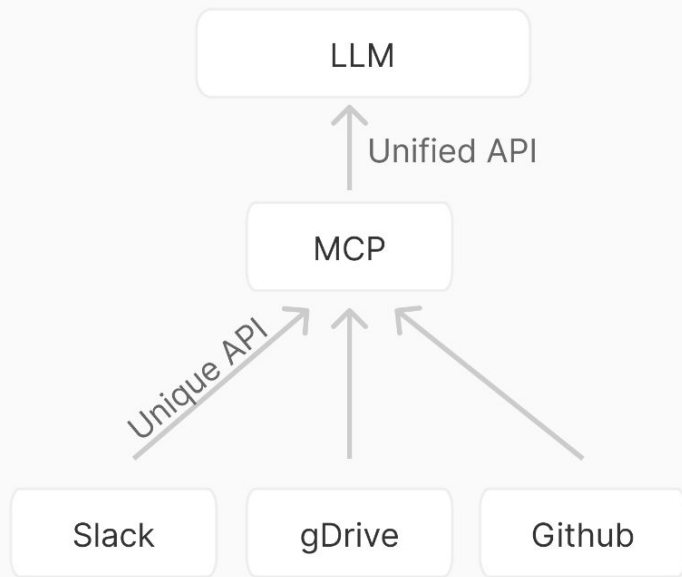
With MCP:

- Agents and tool providers speak a common “language”
- Tool providers only need to make 1 version of their tool that follows MCP
 - Not separate tools for GPT, Gemini, Claude, etc.

Before MCP



After MCP



What is MCP?

A standardized protocol for connecting AI agents to:

1. Tools
2. Resources
3. Prompts
4. External systems

MCP is a clear, standard way to extend models with additional tools and knowledge.

MCP vs tool calling

Tool calling

Model



“I want
`get_weather(city='NYC')`”

MCP

Application



standard protocol



MCP server exposing
`get_weather`

MCP vs tool calling

Combined end-to-end process

Model

↓ tool call

Host/runtime

↓ MCP

MCP server

↓

External service

Tool calling answers: “What tool should I invoke to answer a question?”

MCP answers: “How can my application discover and invoke the tool?”

MCP Architecture

Host: user-facing AI application (e.g. chat app, IDE, agent runtime)

MCP client:

- Lives inside the host
- Speaks MCP
- Maintains connection to a server

MCP Server:

- Exposes tools/resources/prompts
- Connects to underlying data/services
- Does not need to know which model the host uses

Example MCP server (official Python SDK's FastMCP interface)

```
python

# server.py
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("Demo")

@mcp.tool()
def get_weather(city: str) -> str:
    """Get the current weather for a city."""
    return f"It is 18°C and raining in {city}."

if __name__ == "__main__":
    mcp.run()
```

The `@mcp.tool()` decorator turns the function into an MCP tool

What happens when an MCP connection starts?

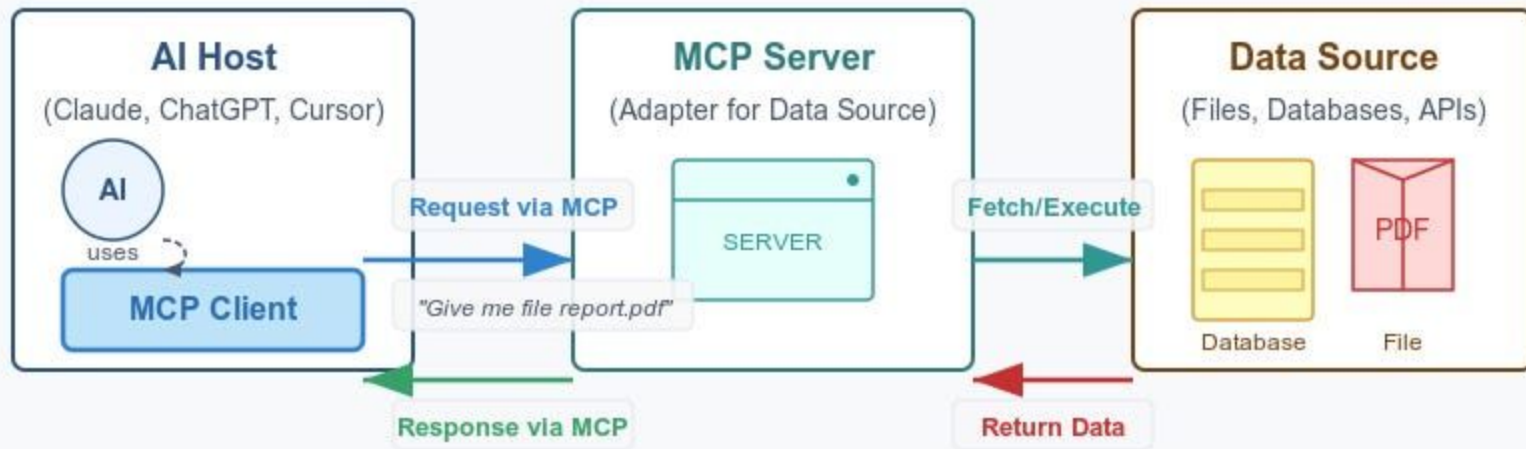
1. Connect
2. Negotiate protocol + capabilities
3. Discover tools/resources/prompts
4. Host decides what to expose/use
5. Calls/read requests happen
6. Results return

What happens when an MCP connection starts?

1. Connect
2. Negotiate protocol + capabilities
3. Discover tools/resources/prompts
4. Host decides what to expose/use
5. Calls/read requests happen
6. Results return

The model does not magically see every server on the internet. The host connects to servers, gets their capabilities, and decides what is available to the model/application.

Model Context Protocol (MCP) Architecture



Model Context Protocol (MCP) Flow

The MCP Client translates AI requests into the standardized protocol format, communicates with MCP Servers, which then interact with external Data Sources.