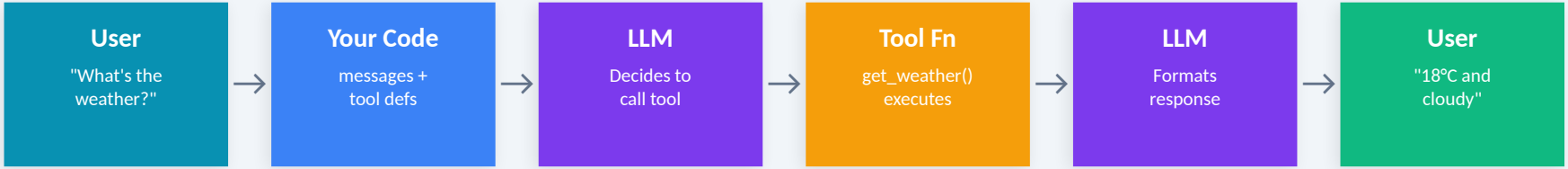
02

Tools & Function Calling

Give LLMs the ability to take actions in the real world

Tools & Function Calling

Tools transform an LLM from a text generator into an agent — the ability to take actions.



Key Concepts

`pydantic_function_tool()`

Type-safe tool definitions with strict mode

`tools=[...]` parameter

Modern pattern (not deprecated "functions")

`"tool"` role results

Return results with role: "tool" (not "function")

Parallel tool calls

LLM can request multiple tools in one response

Exercise: Function Calling — The 4-Step Cycle

`exercises/02_tool_use/01_function_calling.py`

Step 1: Define Tools with Pydantic

```
class GetWeatherParams(BaseModel):
    city: str = Field(description="City name")
    unit: str = Field(default="celsius")
```

Step 2: Send Request with Tools

```
response = client.chat.completions.create(
    model=model,
    messages=messages,
    tools=TOOLS,  # NOT deprecated "functions"
)
```

Step 3: Execute Tool Locally

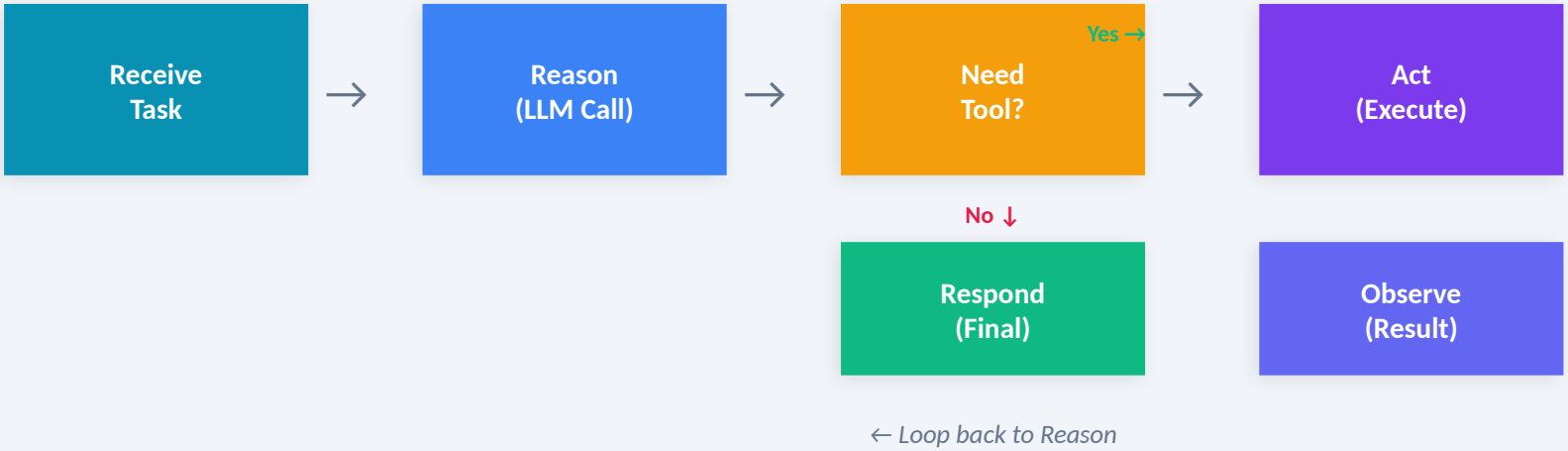
```
for tool_call in assistant_message.tool_calls:
    name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)
    result = TOOL_FUNCTIONS[name](**args)
```

Step 4: Model Composes Final Answer

```
# Re-send with tool results appended
response = client.chat.completions.create(
    model=model,
    messages=messages,  # includes tool results
)
```

Define → Send → Execute → Return: You always control execution. The LLM only decides WHAT to call, never runs code.

The Agent Loop: Reason → Act → Observe



The Core Insight

An agent = LLM + Tools + Loop. The loop runs until the model decides it has enough information (finish_reason='stop') rather than requesting more tool calls. This is implemented in exercises/commons/agent.py and reused throughout the workshop.

Exercise: The Explicit Agent Loop

`exercises/02_tool_use/02_tool_loop.py`

The While Loop — What agent.run() Encapsulates

```
while iteration < MAX_ITERATIONS:
    iteration += 1

    response = client.chat.completions.create(
        model=model,
        messages=messages,
        tools=TOOLS,
    )
    assistant_message = response.choices[0].message
    messages.append(assistant_message.model_dump())

    # No tool calls? We're done.
    if not assistant_message.tool_calls:
        logger.info("Final: %s", assistant_message.content)
        break
```

How It Works

1. Send messages + tool defs
2. Model returns tool_calls or text
3. If tool_calls: execute & append
4. Loop until finish_reason=stop
5. MAX_ITERATIONS prevents runaway

Key API Pattern

```
# Serialize assistant reply
msg.model_dump()

# Tool result format
{"role": "tool",
 "tool_call_id": tc.id,
 "content": json.dumps(result)}
```

This explicit loop is exactly what commons/agent.py run() encapsulates — you'll use run() from Chapter 3 onward.

Mock Tools: exercises/02_tool_use/tools/

8 mock tools simulating real APIs — reused across exercises 02 through 07.

`get_weather()`
Mock weather data for 6 cities

`calculate()`
Safe math expression evaluator

`convert_temperature()`
Celsius ↔ Fahrenheit conversion

`lookup_order()`
Order lookup by ID

`search_database()`
Product catalog with filtering

`search_faq()`
FAQ knowledge base search

`get_stock_price()`
Stock prices for 6 tickers

`process_refund()`
Refund processing (always succeeds)

Static mock data — no external API calls needed. Focus on the patterns, not the plumbing.

Chapter 2 Recap: Tools & Function Calling

1. Tools = Pydantic models + openai.pydantic_function_tool() with strict mode
2. 4-step cycle: Define → Send → Execute locally → Return result with role: "tool"
3. The agent loop (while + tool_calls check) is the foundation of every agent
4. commons/agent.py run() encapsulates this loop — used from Chapter 3 onward
5. LLMs never execute code — they decide WHAT to call, you control HOW it runs

Next: Chapter 3 — Single Agent Pattern → One agent with tools and persistent conversation