AMERICAN UNIVERSITY OF BEIRUT

Fall 2025

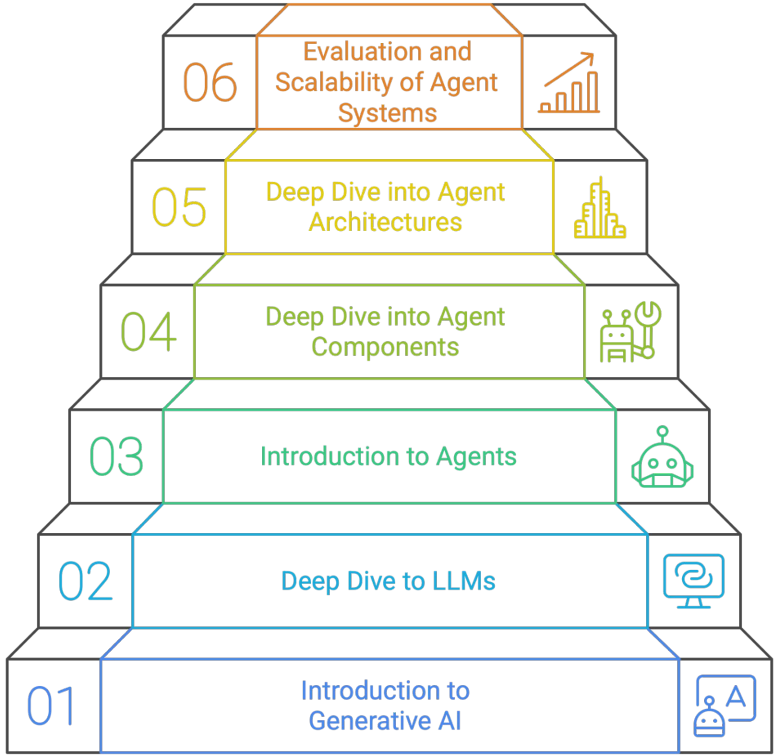
EECE 503P/798S: Agentic Systems

C5 - Agent Tools

Lesson Objectives

- Understand what **agent's tools** are
- Learn about **communication protocols** in python
- Learn about the **best practices** for creating an Agent's tools
- **Create your own tools** for an agent
- Understand **Model Context Protocol**
- Use **MCP to create tools**
- Understand **Agent Context Protocol**

Course Timeline

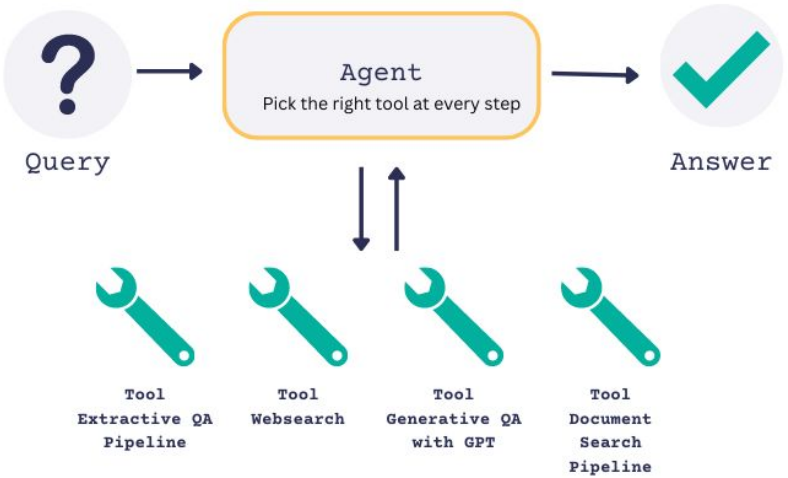


We are (still) here

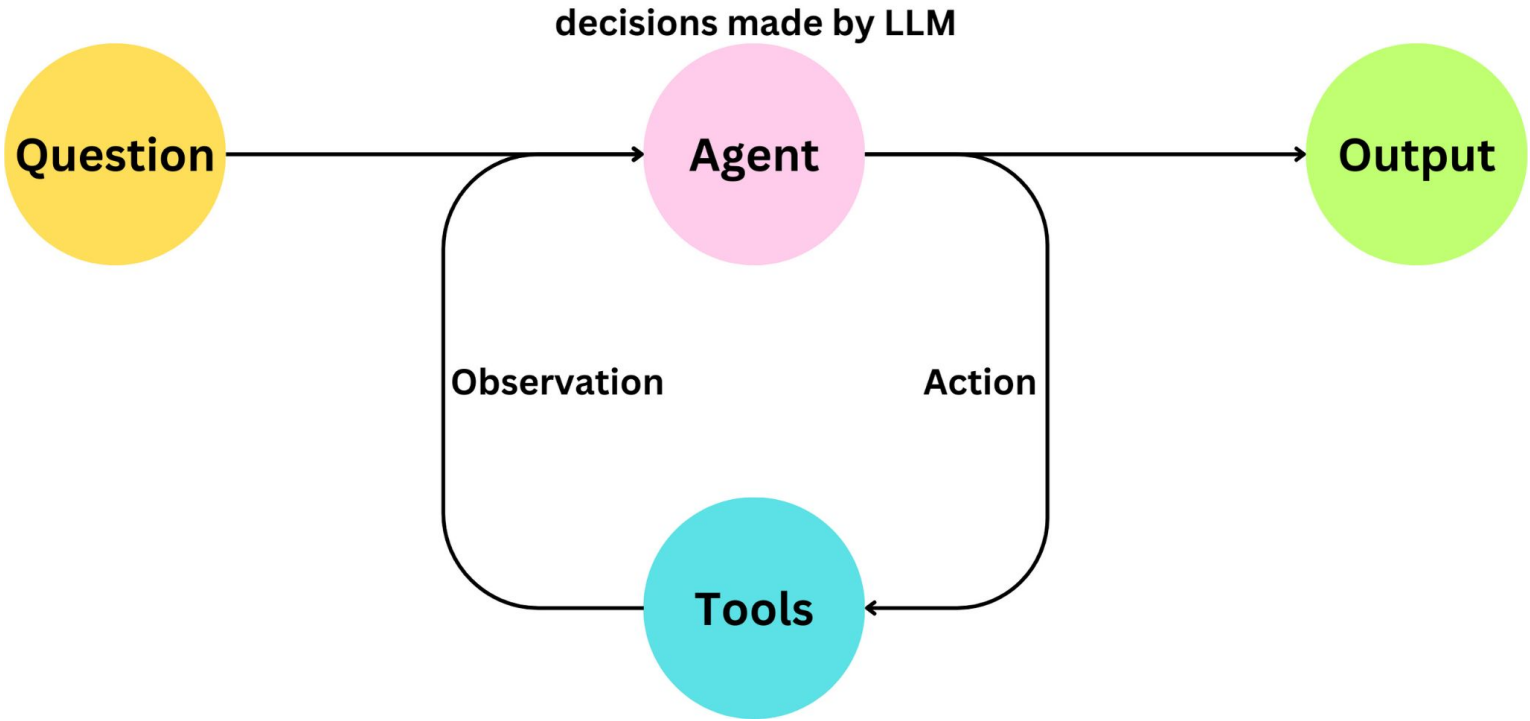
Reminder: What are tools?

Tool Definition

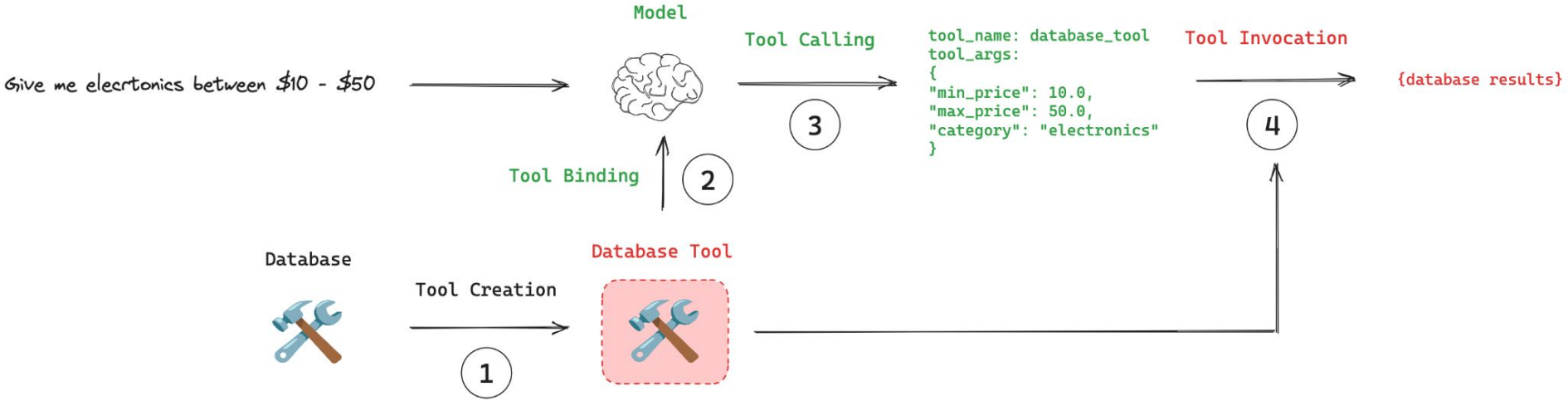
- Tools are what they sound like: **functions** that do something on behalf of the AI model
- These are typically operations that can have **effects** or **require computation** beyond the AI's own capabilities



Agent's Tools



Example of a Tool Flow



LLMs already know a lot, but are they good at everything?

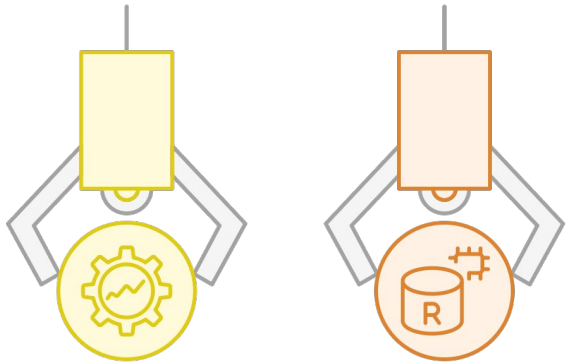
Why do we need tools?

- A good tool should be something that **complements** the power of an LLM.
- LLMs predict the **completion of a prompt** based on their training data, which means that their internal knowledge only includes **events prior** to their training.
- Therefore, if your agent needs **up-to-date** data you must provide it through some **tool**.
- For instance, if you ask an LLM directly (without a search tool) for today's weather, the LLM will potentially **hallucinate** random weather.

Tool	Description
Web Search	Allows the agent to fetch up-to-date information from the internet.
Image Generation	Creates images based on text descriptions.
Retrieval	Retrieves information from an external source.
API Interface	Interacts with an external API (GitHub, YouTube, Spotify, etc.).

Why do we need tools?

Improve Efficiency:



Improve Efficiency

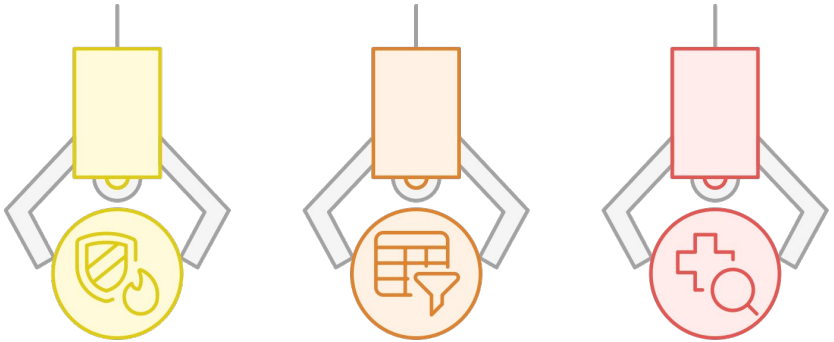
Tools help to improve efficiency by delegating heavy lifting. This includes tasks like data processing and file manipulation.

Reduce Token Usage

Tools reduce token usage by returning only parsed results, optimizing prompt efficiency.

Why do we need tools?

Enhance Reliability & Safety:



Enhance Reliability & Safety

Tools enhance reliability and safety by providing a controlled environment. They help ensure consistent and predictable outcomes.

Constrain Outputs

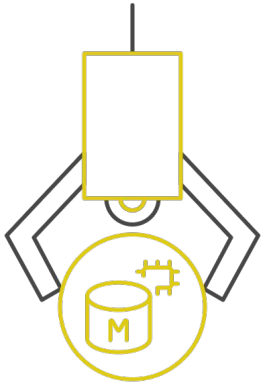
Tools constrain outputs to known API schemas, ensuring data consistency. This helps in maintaining compatibility across systems.

Validate & Sanitize

Tools validate and sanitize inputs and outputs in a controlled environment. This process improves data quality and security.

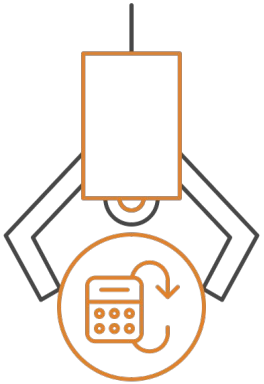
Why do we need tools?

Overcome LLM Limitations:



Memory Limitations

LLMs lack built-in memory for external data. They cannot access external databases.



Computation Limitations

LLMs cannot perform computations or side effects natively. They need external tools.

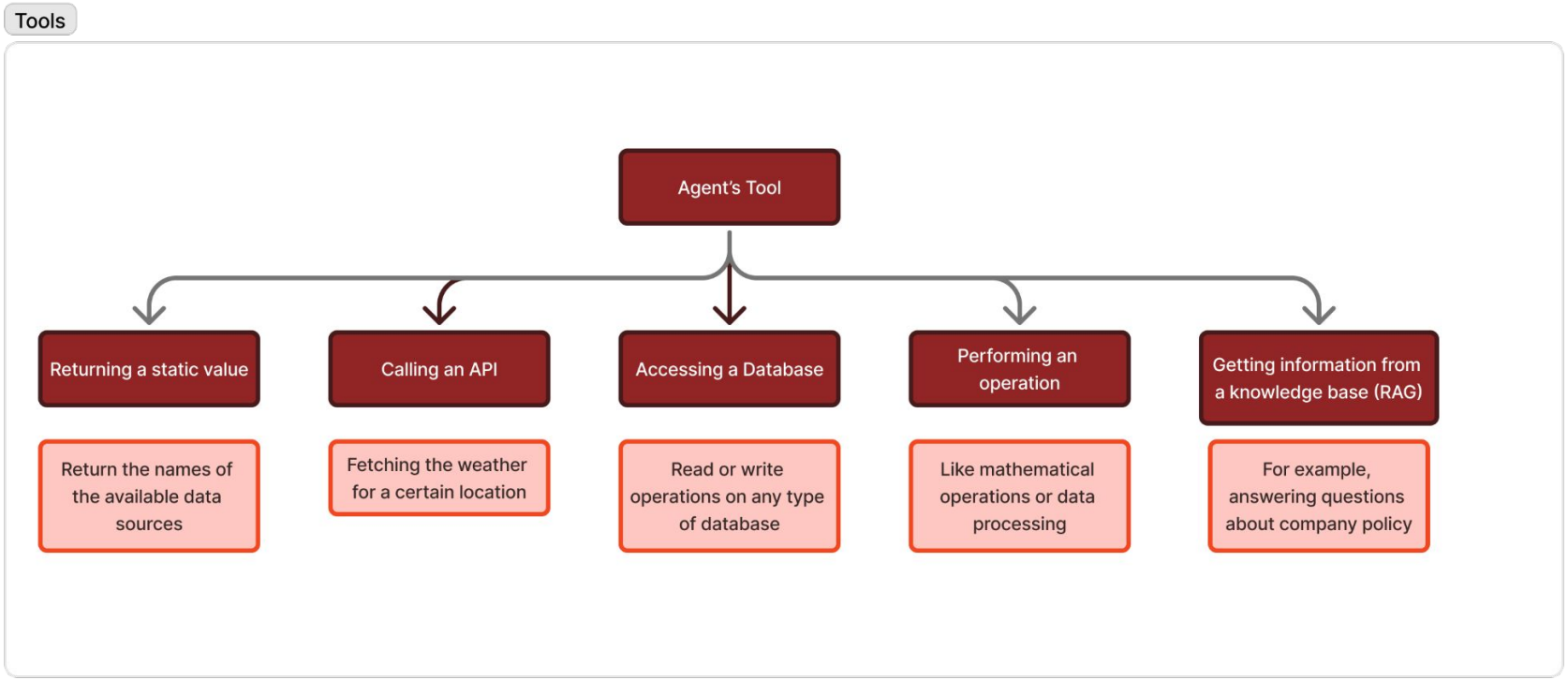
What do tools look like? How are they created?

Building Blocks of Tools

- A Tool is a **function** given to the LLM. This function should fulfill a **clear objective.**
- A Tool should contain:
  - A **textual description** of what the function does.
  - A ***Callable*** (something to perform an action).
  - ***Arguments*** with typings.
  - (Optional) **Outputs** with typings.

```python
def get_current_weather(city, country, time):
  """ Get the current weather in a city"""
  return WeatherAPi(city=city, country=country, time=time)
```

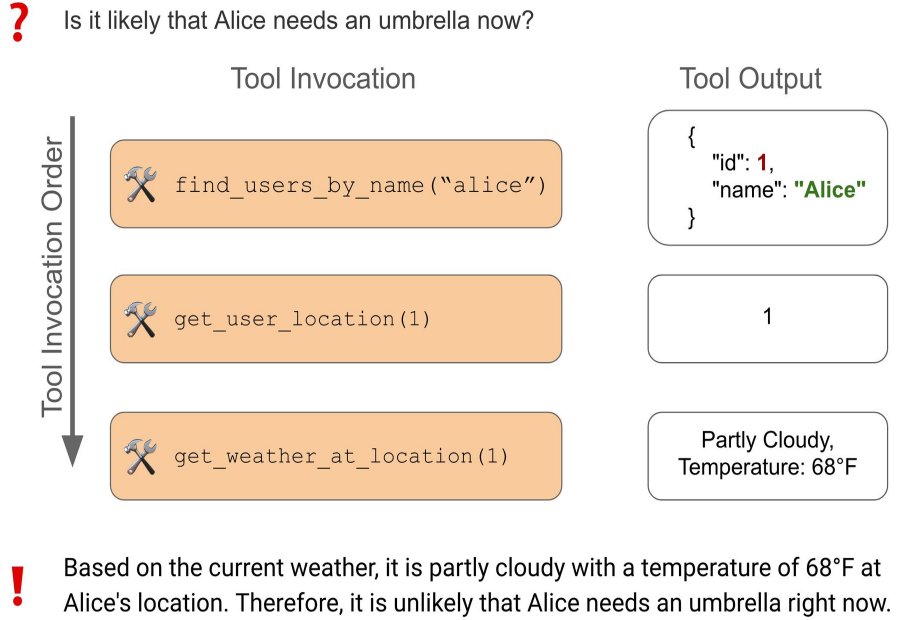
Building Blocks of Tools



Characteristics of Agent's Tools

Key Characteristics:

1. **Well-Defined API (if needed)**: clear input parameters & output schema
2. **Isolated Functionality**: single responsibility to simplify invocation
3. **Deterministic Behavior**: predictable results given valid inputs



What are APIs? How do they actually work?

Communication Protocols in Python

- Communication protocols are essentially the agreed-upon **rules and standards** that govern **how data is exchanged** between two or more devices or software entities.

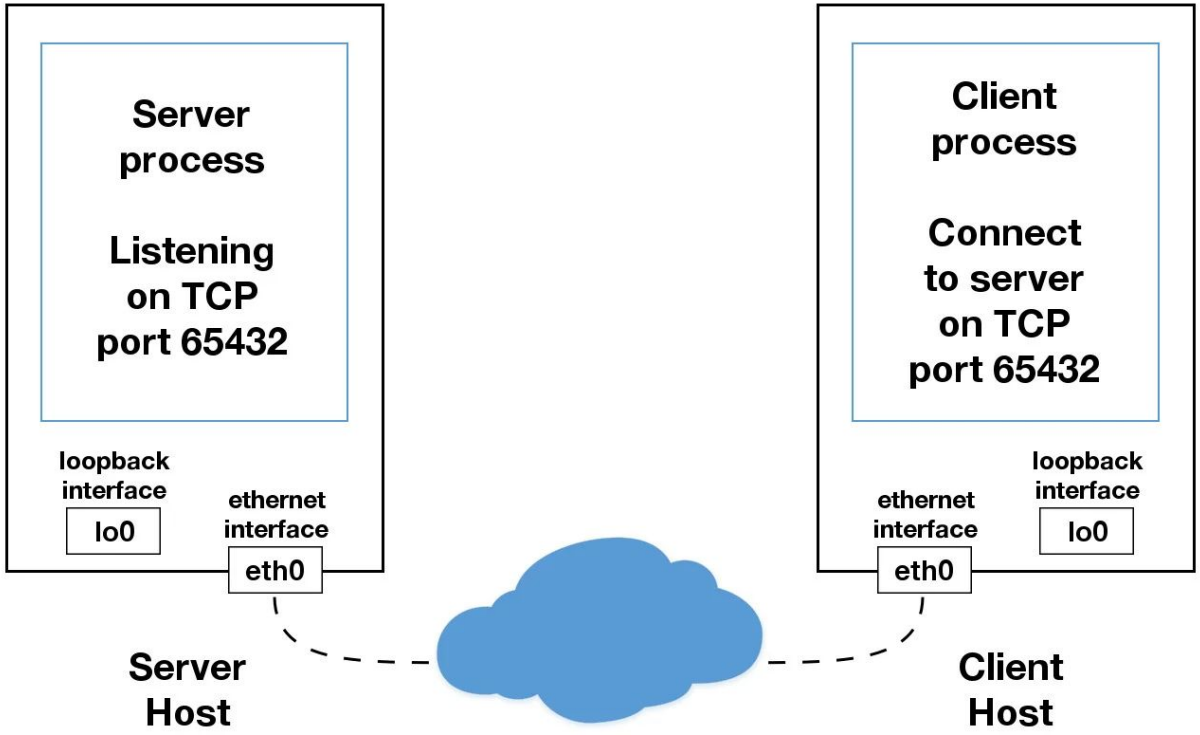
Rules for Data Exchange

Protocols dictate everything from how a **data packet** is structured to how **errors** are handled, ensuring orderly and predictable communication.

The "Language" of Computers

Think of protocols as specialized languages that **computers use to "talk" to each other.**

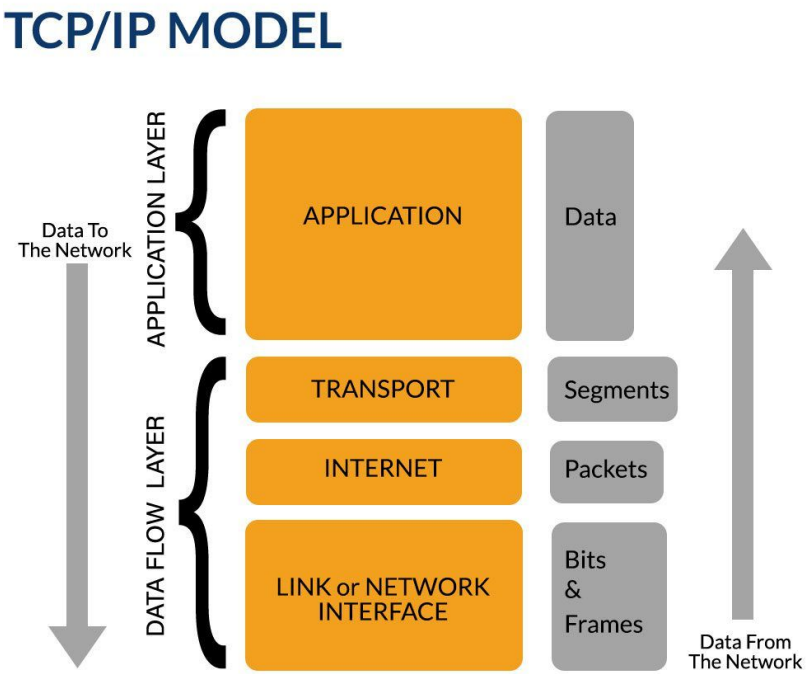
Communication Protocols in Python



Protocols in Action: TCP/IP

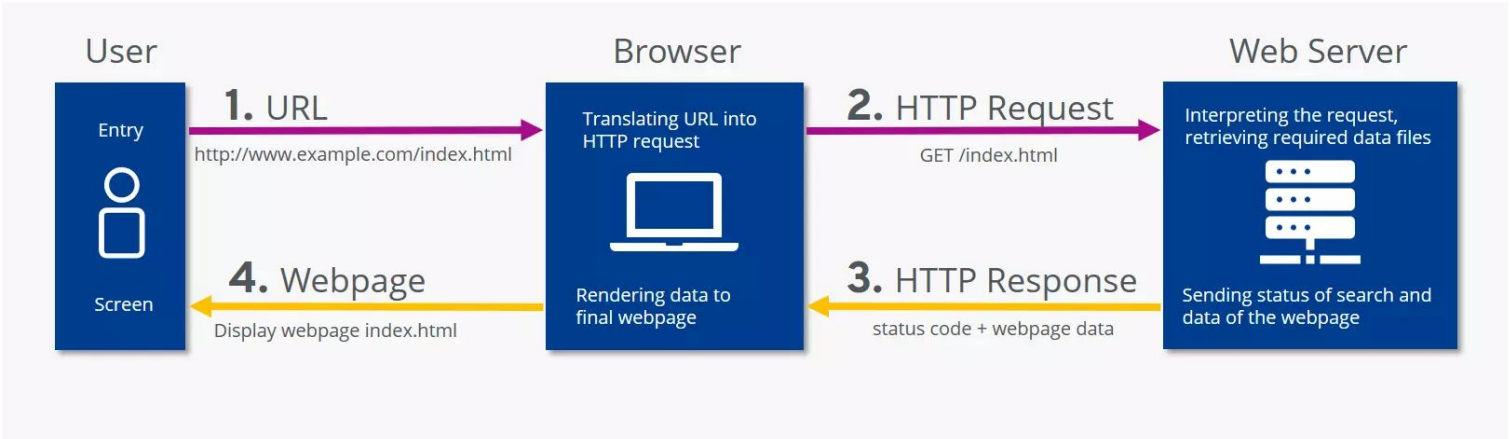
The Transmission Control Protocol/Internet Protocol (TCP/IP) suite is the fundamental set of protocols for internet communication.

1. TCP handles reliable, ordered, and error-checked **delivery of data streams**, breaking data into packets and reassembling them at the destination.
2. IP is responsible for addressing and **routing these packets across networks.**



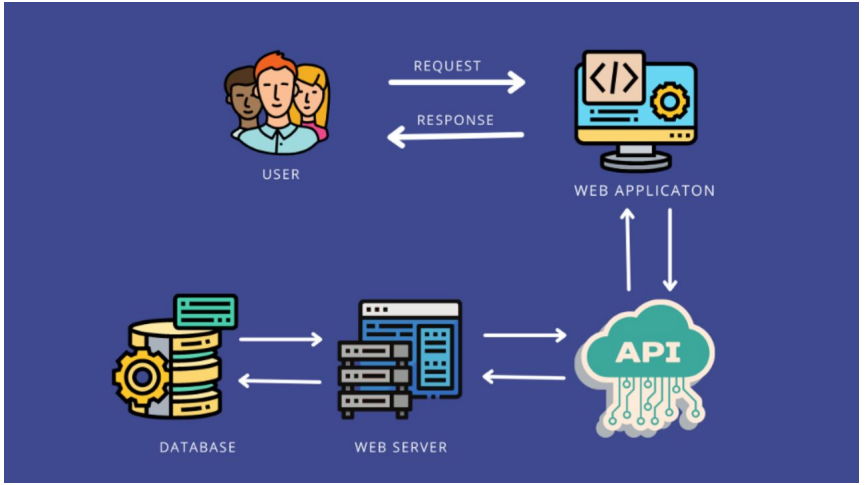
Protocols in Action: HTTP

The Hypertext Transfer Protocol (HTTP) is an **application-layer protocol** built on top of TCP/IP. It's the protocol that enables communication between **web clients** (like your browser) and web servers.



APIs in Python

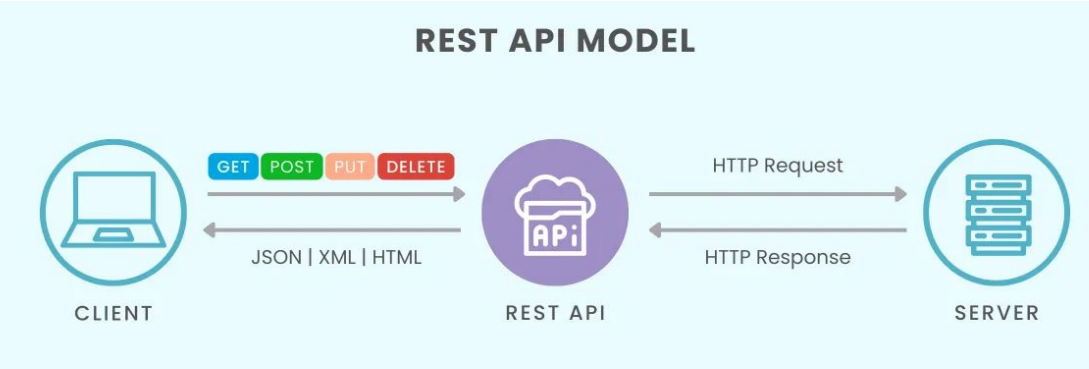
- An **Application Programming Interface (API)** is a set of defined rules, specifications, and tools for building software applications.
- It serves as an intermediary that **allows different software components or applications** to communicate and interact with each other.



Imagine that every time you need to use chatGPT, you need to load it on your machine. This is what APIs are for.

REST APIs: The most popular web API style

- While there are various styles of APIs (like SOAP, GraphQL, etc.), **REST (Representational State Transfer)** is by far the most widely used architectural style for **designing networked applications**, particularly web services.
- REST APIs are **built on standard HTTP methods**, making them highly compatible with web infrastructure.



REST APIs: The most popular web API style

HTTP Methods

REST APIs primarily use standard HTTP methods like `GET` (retrieve data), `POST` (create new data), `PUT` (update existing data), and `DELETE` (remove data).

Resource-Oriented

Data is treated as "resources," each uniquely identified by a Uniform Resource Locator (URL), also known as an endpoint. For example, `/users` or `/products/123`.

Stateless Communication

Each request from a client to a server contains all the necessary information for the server to understand and fulfill the request. The server doesn't "remember" past client interactions, which improves scalability and reliability.

Autonomy of an API Request

URL (Endpoint)

This is the specific web address that identifies the resource you want to interact with on the API server. It typically follows a structured path, e.g., `https://api.example.com/users/123`.

HTTP Method

Also known as the HTTP verb, this indicates the type of action you want to perform on the resource. Common methods include **GET** (retrieve), **POST** (create), **PUT** (update/replace), **PATCH** (partial update), and **DELETE** (remove).

Headers

These are key-value pairs that provide metadata about the request. Headers can include information like **Authentication tokens** (e.g., API keys), **Content-Type** (e.g., `application/json`), **Accept** (what response format the client prefers), and **User-Agent**.

Body (Payload)

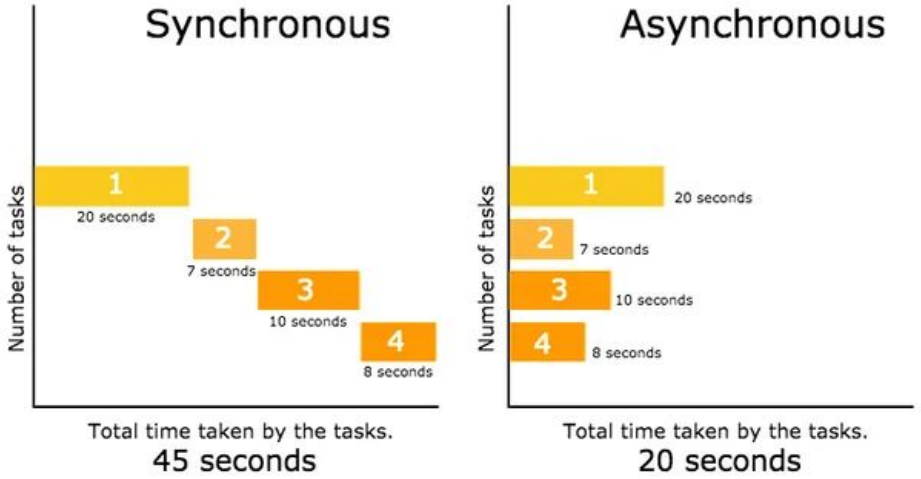
The body contains the actual data being sent to the server. This is primarily used with `POST`, `PUT`, and `PATCH` requests when you are creating or updating resources. The data format is usually JSON or form-encoded data.

Hands on: requests_and_api_wrappers_tutorial.ipynb

Since there are multiple processes happening at the same time, we cannot wait for one process to be over before starting a new one. Thus, we need Async functions.

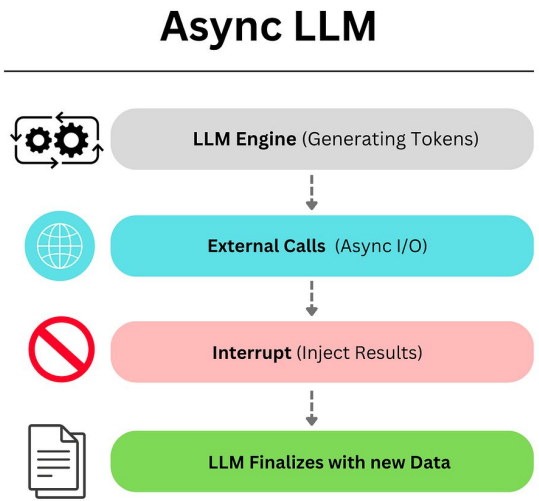
Async Functions in Python

"Async" is short for "**asynchronous**", meaning "not at the same time". In programming, it refers to a technique that allows tasks to **run concurrently** without blocking the main execution thread, making applications **more responsive and efficient**.



Async Processes for LLM agents

LLM inferences and external API calls (e.g., to databases, tools, or other LLMs) can be **time-consuming**. Async allows these operations to **run in the background** without blocking the main application thread.



Agent and Tools

Now that we can create tools. How do we expose those tools to the agent?

How do agents use tools?

To expose an agent to a tool, we need to do two things:

First, we need the agent to **know that it has access** to tools through its system **prompt**.

```
system_message="""You are an AI assistant designed to help users efficiently and accurately. Your
primary goal is to provide helpful, precise, and clear responses.

You have access to the following tools:
{tools_description}
"""
```

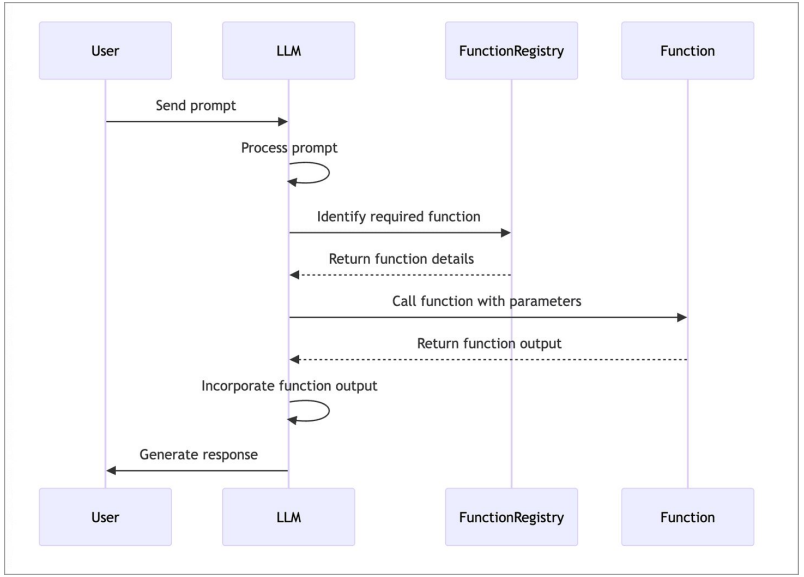
Let's say an agent chose to use a specific tool, how does it actually trigger the tool function?

How can we go from LLM output to executing a specific piece of code?

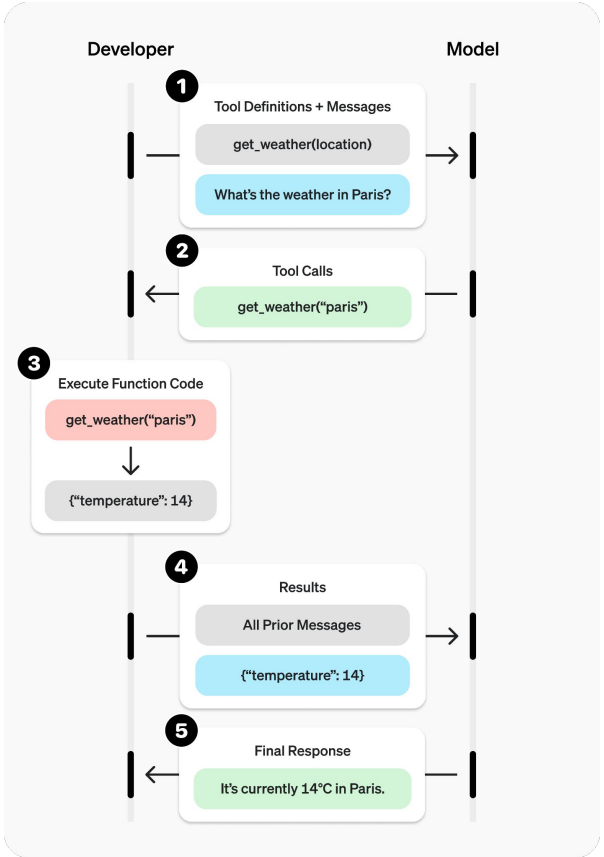
This is where function calling comes in.

Function Calling

- Powerful capability that enables Large Language Models (LLMs) to **interact** with your **code** and **external systems** in a structured way.
- Instead of just generating text responses, LLMs can understand **when** to call specific functions and **provide** the necessary **parameters** to execute real-world actions.



Function Calling



Function Calling

- In function calling, we define the **tools**, their **inputs**, their **outputs**, and their description in a pre-structured list.
- The format of this list may **vary** depending on the LLM provider.

```
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_n_day_weather_forecast",
            "description": "Get an N-day weather forecast",
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {
                        "type": "string",
                        "description": "The city and state, e.g. San Francisco, CA",
                    },
                    "format": {
                        "type": "string",
                        "enum": ["celsius", "fahrenheit"],
                        "description": "The temperature unit to use",
                    },
                    "num_days": {
                        "type": "integer",
                        "description": "The number of days to forecast",
                    },
                },
                "required": ["location", "format", "num_days"],
            },
        },
    }
]
```

Function Calling

After we provided the agent with a list of available tools, and added a tool schema, what does the agent output when it selects a tool?

Let’s move to a hands-on to find out.

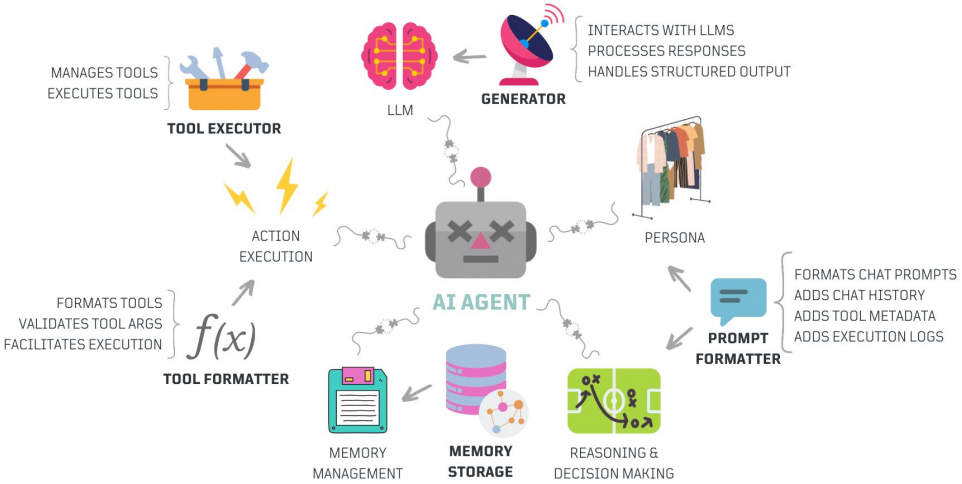
Hands on: openai_function_calling.ipynb

Types of Tools

Why are we always making one tool per functionality? Can’t we group functionalities in one function?

Why do we need single purpose tools?

- Researchers have found that agents produce **more accurate outputs** when they solve problems one **small step at a time**.
- Combining multiple functionalities and outputs in one call confuses the agent's sequential logic.



Why do we need single purpose tools?

- Having single purpose tools also **simplifies the process** for the developers.
- Having **clear** and **straightforward descriptions** for tools simplifies prompt design and schema definitions.

Why do we need single purpose tools?

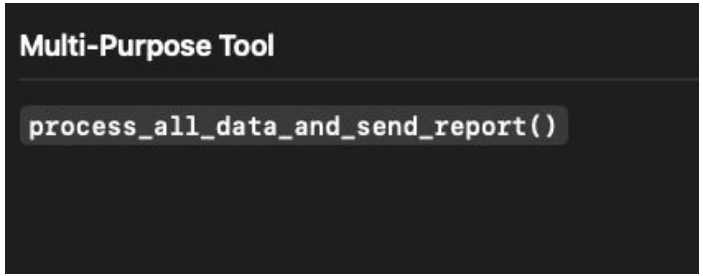
When creating production-grade code, error handling is very important. Having multi-purpose tools might cause errors to multiply for different functionalities.

For single purpose tools, updates to one tool don't risk **error propagation** in unrelated functionality.

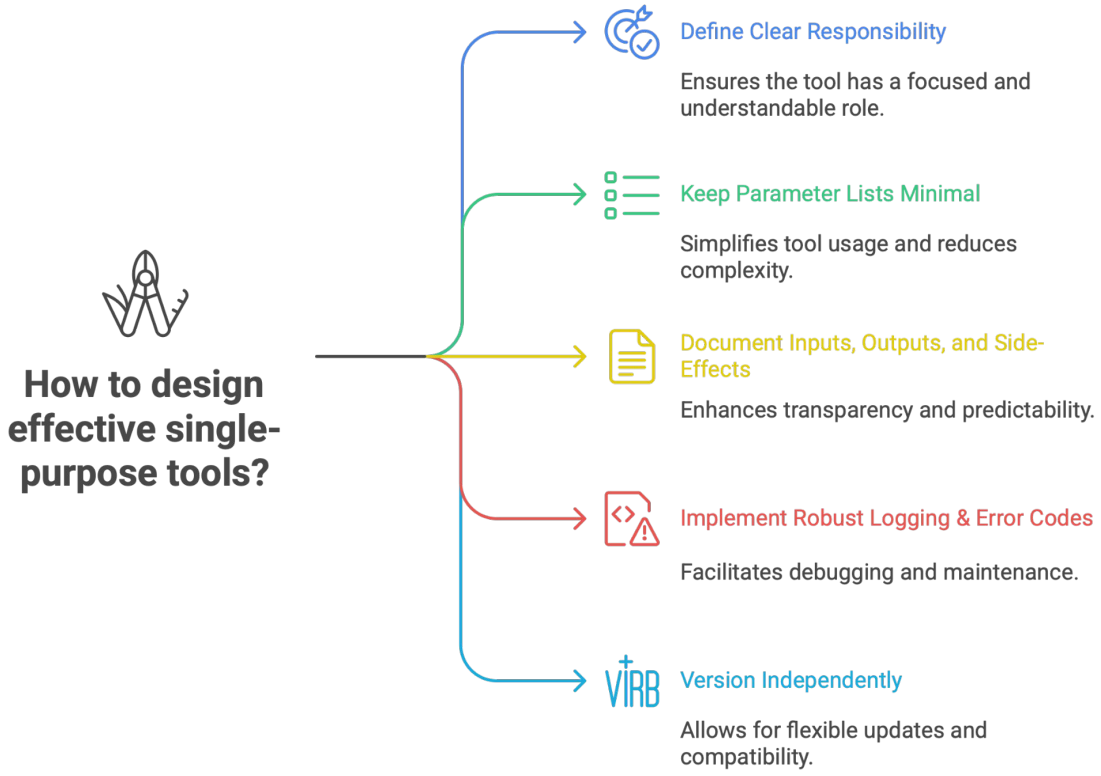
- Failures are confined to **one operation**
- Easier to implement **retries or fallbacks** per tool
- **Unit tests** cover a single code path
- Higher **confidence in reliability**

Creating Single Purpose Tools

How can we transform this **multi-purpose tool** into a **collection of single purpose tools** that make the process well-defined?

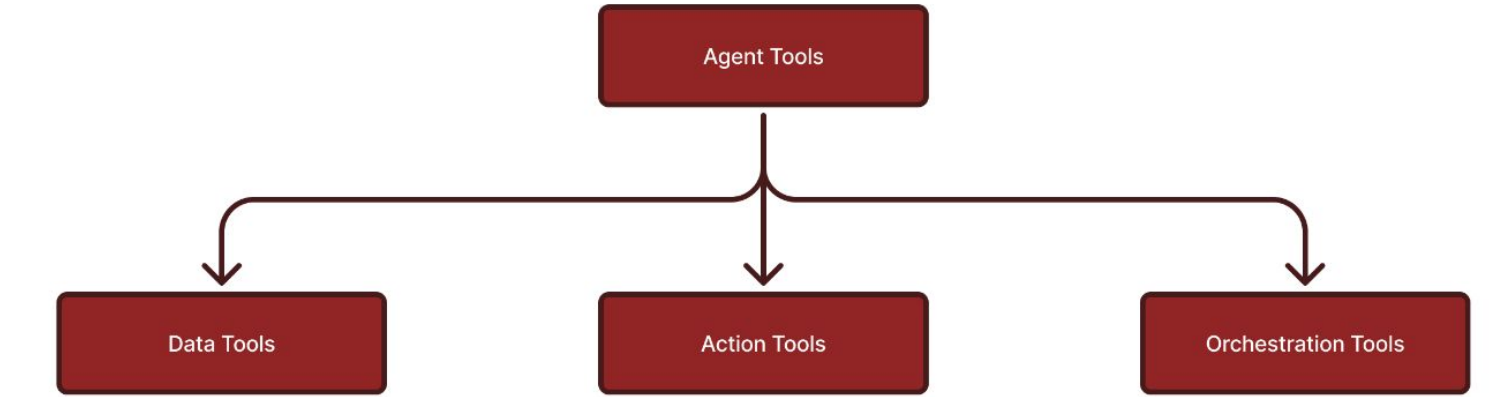


Crafting The Perfect Tool



Now that we know how a singular tool should look like, what are the different categories of tools we can make?

Types of Tools



Enable agents to retrieve context and information necessary for executing the workflow.

Enable agents to interact with systems to take actions such as adding new information to databases, updating records, or sending messages.

Agents themselves can serve as tools for other agents, which we will get to in the upcoming chapters

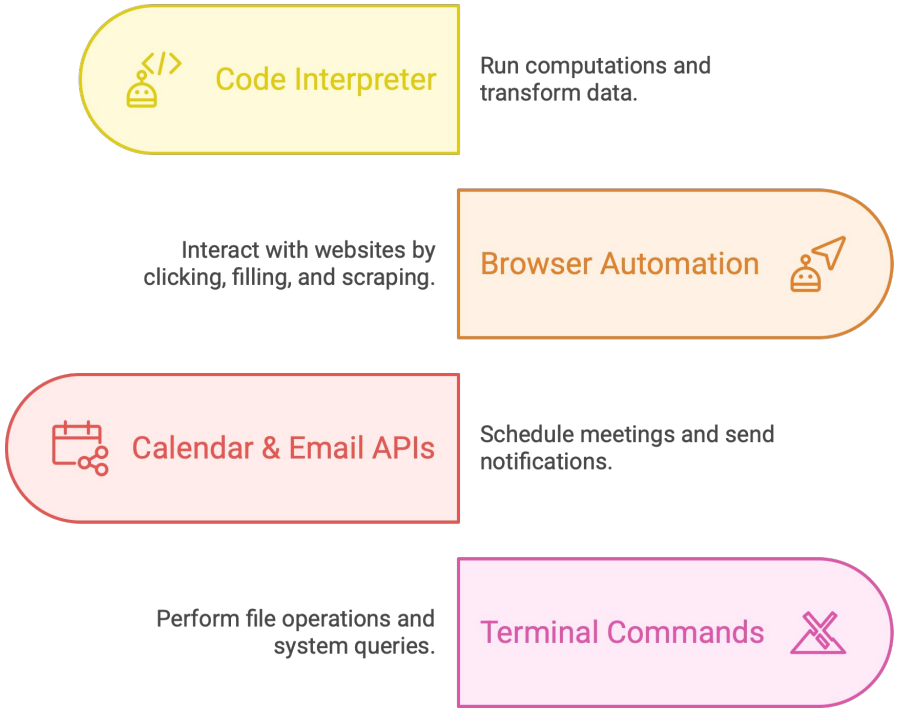
Data Tools

Examples

Hands on: data_tools_tutorial.ipynb

Action Tools

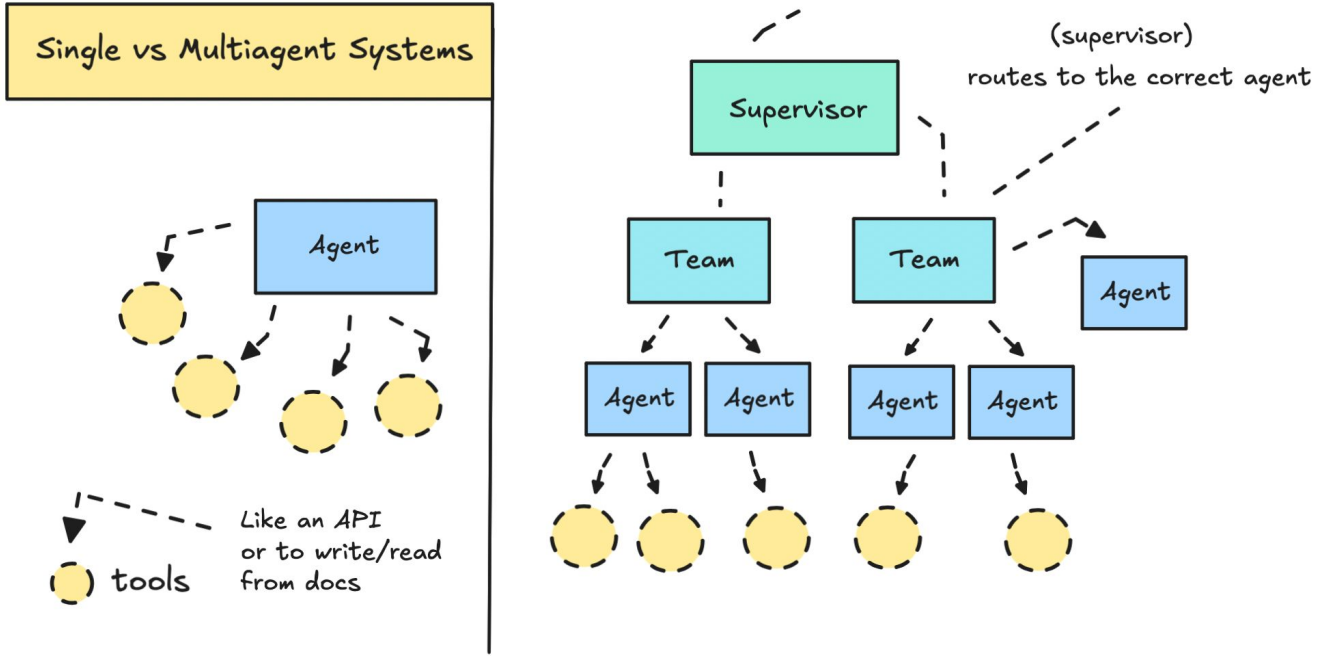
Examples



Hands on: action_tools_tutorial.ipynb

Orchestration Tools

Examples

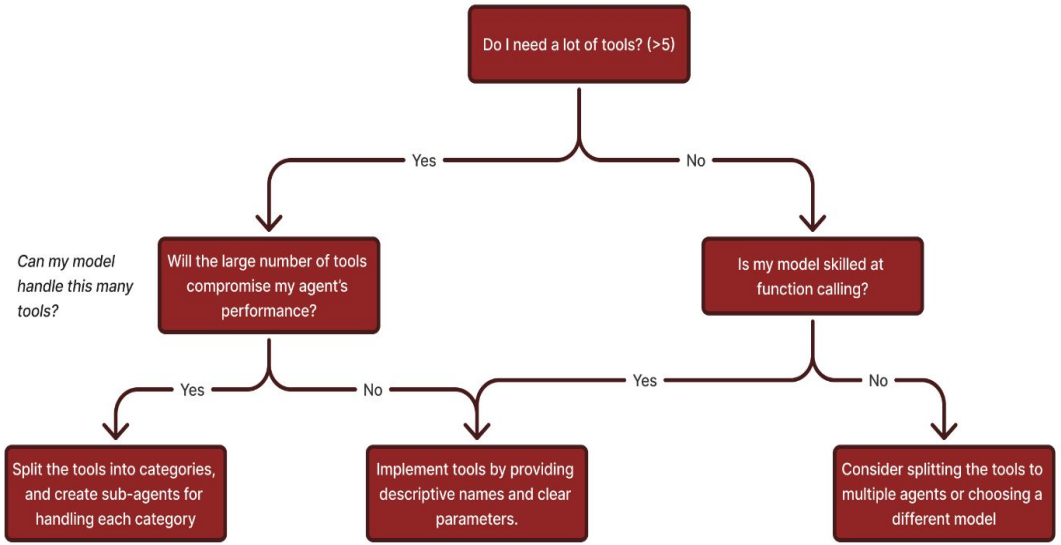


Hands on: orchestration_tools_tutorial.ipynb

What if you're building a complex agent that has access to many tools. Is there such a thing as too many tools?

How Many Tools Should an Agent Have?

- The issue isn't solely the number of tools, but their **similarity or overlap**.
- Some implementations successfully manage more than 15 well-defined, distinct tools while others struggle with fewer than 10 overlapping tools.
- Manus reported using around 50 tool call per average usage.



Model Context Protocol

Now that we have created multiple types of tools, we notice that each tool has a different return structure and different trigger parameters.

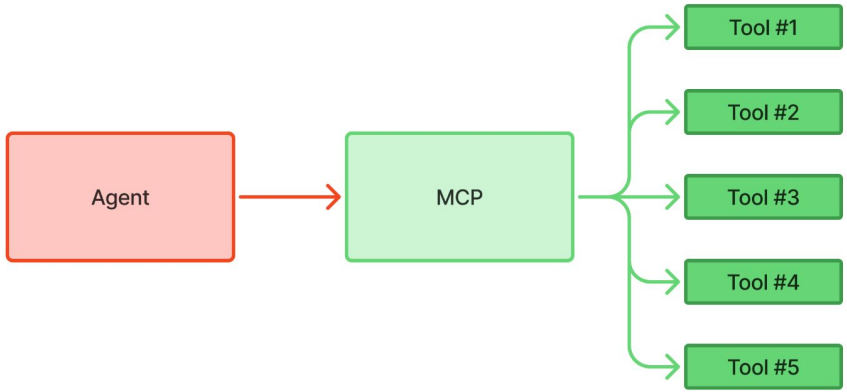
If I need to use the same tool while developing with the OpenAI SDK, you would have to completely rewrite it to use with LangChain or Autogen agents.

Thus, we need a unified communication protocol for agent-tool communication: Model Context Protocol

The USB-C of Agentic AI

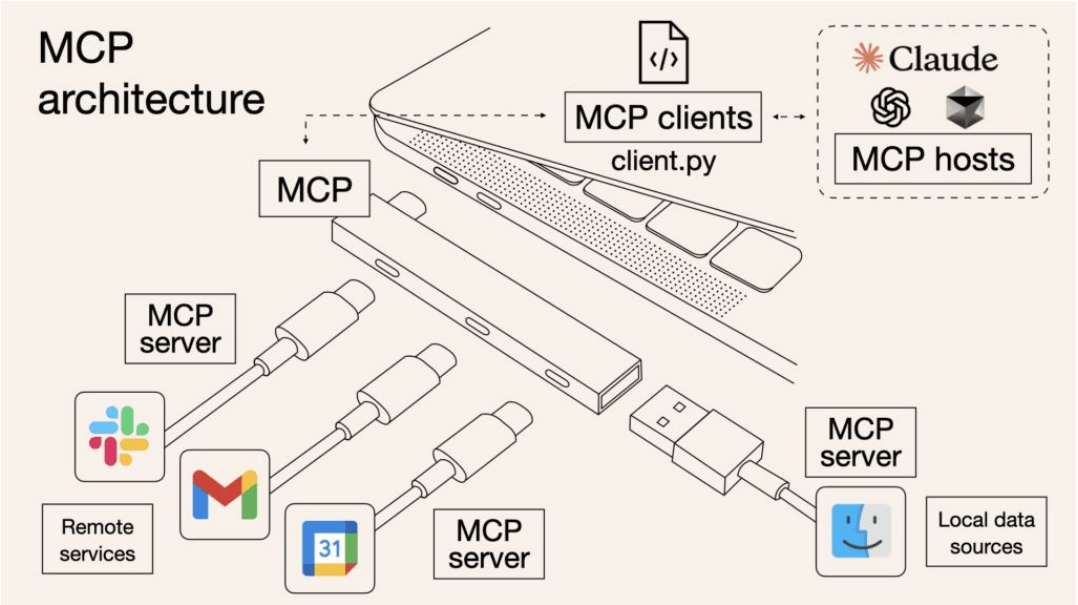
In November of 2024, **Anthropic released Model Context Protocol**.

Whether it's reading files from your computer, searching through an internal or external knowledge base, or updating tasks in an project management tool, MCP provides a **secure**, **standardized**, ***simple*** way to **communicate with the tools.**



Why do we need MCP?

MCP allows you to create a **communication interface** with your tools so that you can **easily share resources and tools** (between developers, agents, and versions of one agent).



Previously, we saw that it is relatively simple to create a tool, so why do we need to go through this hassle?

Why do we need MCP?

- Imagine that your agent **needs access** to a specific communication application, like Slack.
- It turns out that another developer has already hacked out using slack as a tool, but now you need to **refactor, customize, and integrate their tool in your code.**
- However, if the tool was developed as an MCP server, you can just **plug it in** to your MCP Host!

MCP Example

Fetch is an MCP server that fetches web pages through headless chrome and supplies the model with websites.

Available Tools

- `fetch` - Fetches a URL from the internet and extracts its contents as markdown.
  - `url` (string, required): URL to fetch
  - `max_length` (integer, optional): Maximum number of characters to return (default: 5000)
  - `start_index` (integer, optional): Start content from this character index (default: 0)
  - `raw` (boolean, optional): Get raw content without markdown conversion (default: false)

Existing MCP Server Deployments

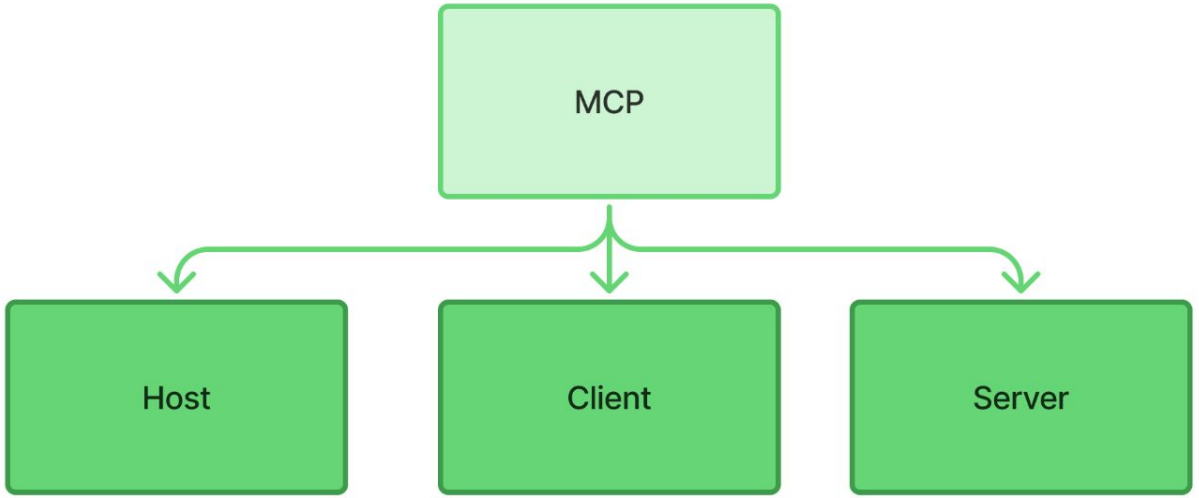
https://mcpservers.org/

https://github.com/modelcontextprotocol/servers

…

How does MCP Work?

MCP is made up of a **Host, Clients,** and **Servers.**

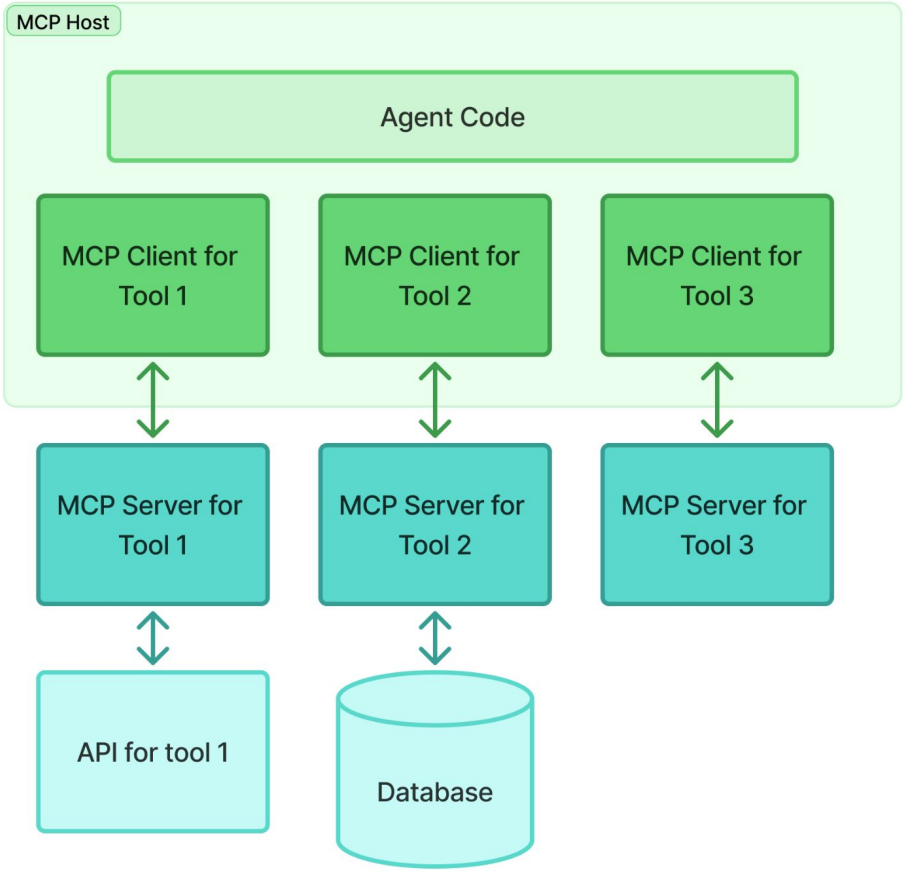


How does MCP Work?

This diagram shows how the MCP system connects an AI agent to multiple external tools (like APIs, databases, etc.).

It's organized into three main layers:

- The MCP Host
- The MCP Clients (inside the host)
- The MCP Servers (outside the host)



How does MCP Work?

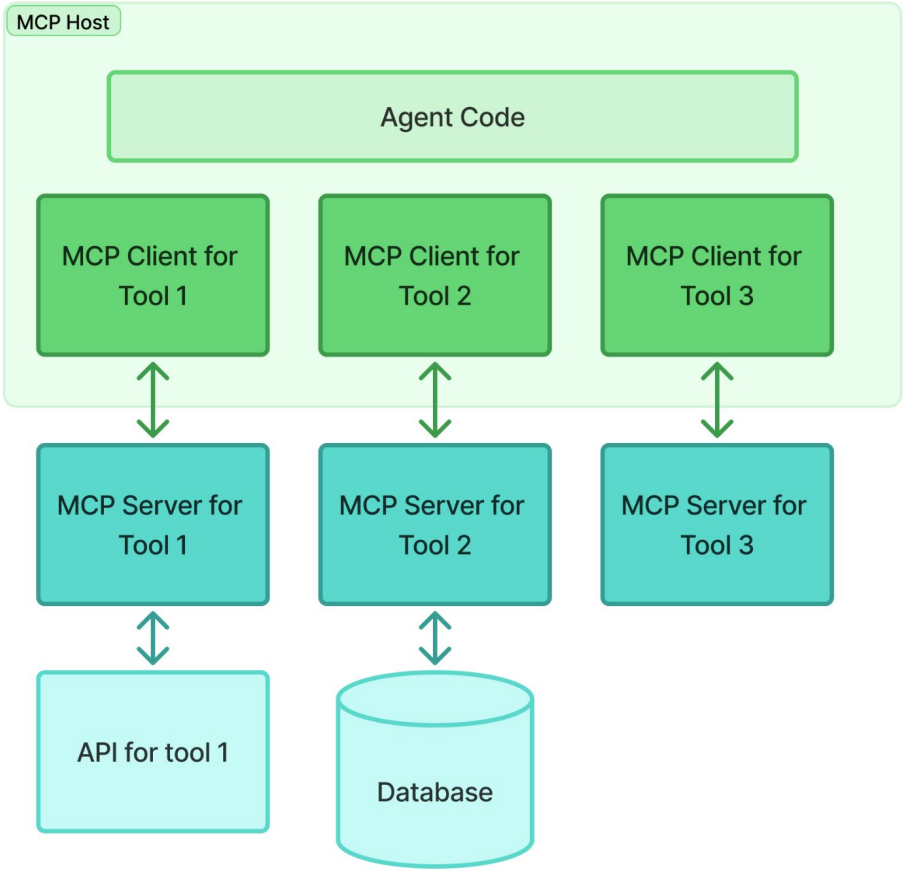
MCP Host (the “environment”): This is the main space where **your** AI agent code runs.

Think of it as a “container” that manages everything.

Inside it, you have the agent, the piece of code that thinks, plans, and decides what to do.

The host also includes MCP clients, one for each external tool the agent needs to talk to.

In short: Host = manager + agent + connectors (clients)



How does MCP Work?

MCP Clients (the “connectors”): Each MCP client is like a translator that helps the agent talk to one tool.

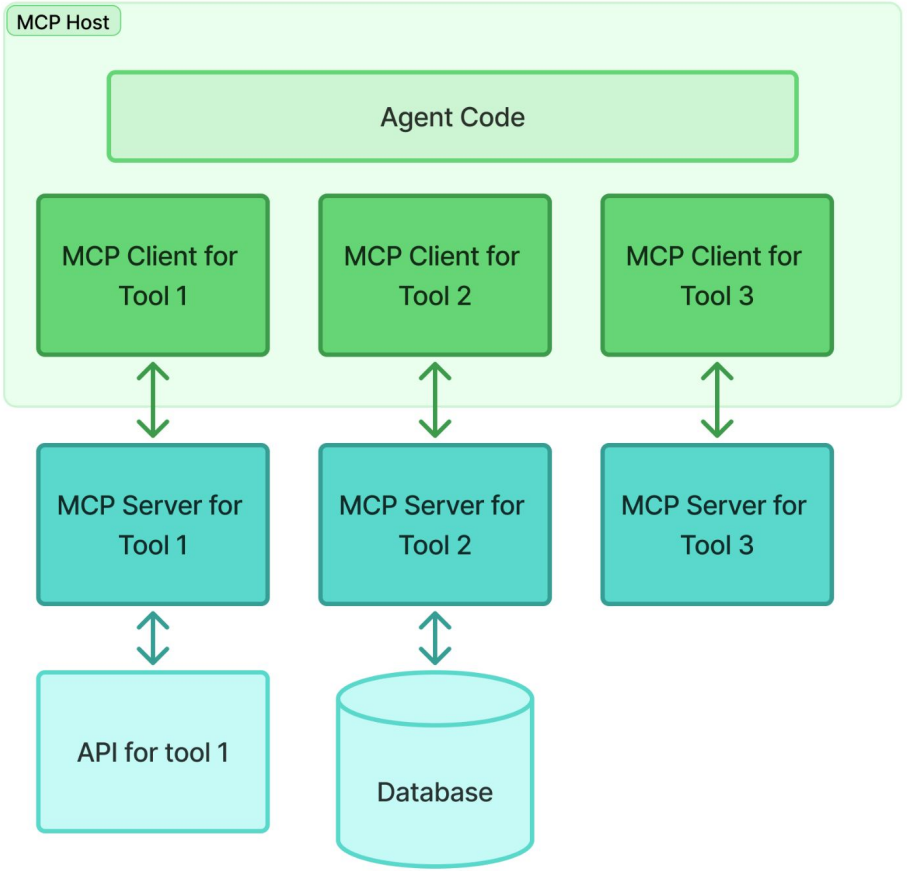
MCP Client for Tool 1, Tool 2, and Tool 3 each know how to communicate with a specific server.

When the agent needs something, say, database info, it sends the request through the correct client.

Example:

- Agent says: “Get me all users from the database.”
- MCP Client for Tool 2 sends that request to the corresponding MCP Server for Tool 2.

In short: Clients carry messages between the agent and the servers.



How does MCP Work?

MCP Servers (the “workers”): Each MCP server sits outside the host and actually does the job.

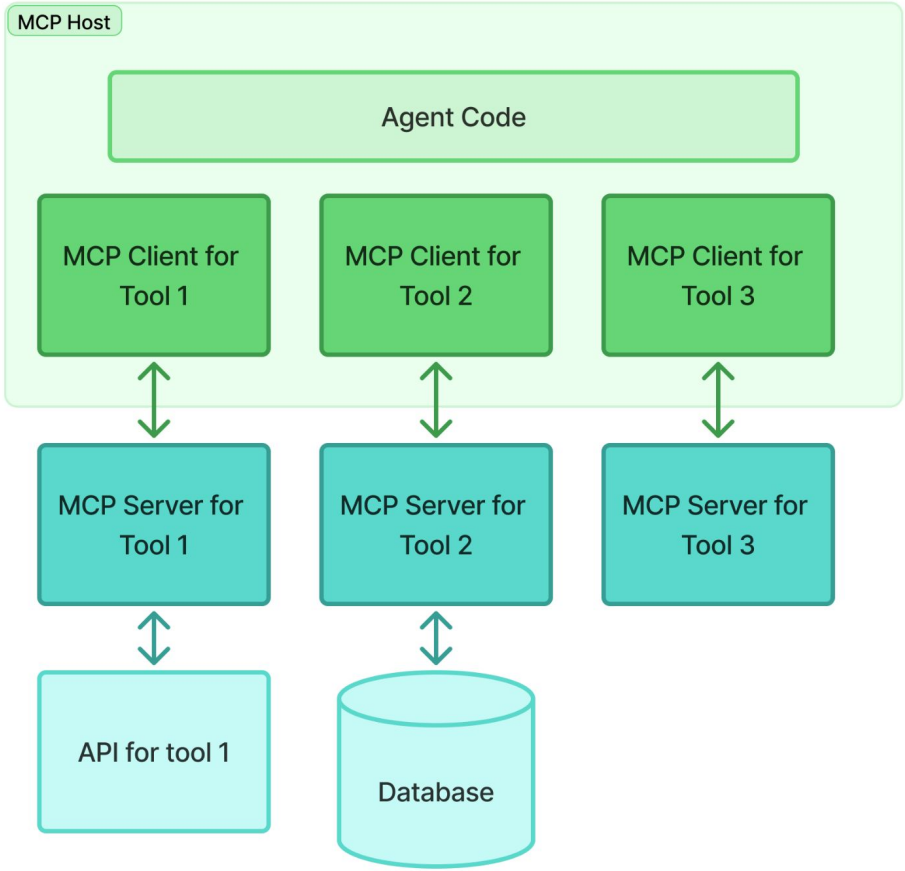
These servers connect to real systems; APIs, databases, etc.

They know how to perform the task and return the results back through the MCP client.

Example:

- MCP Server for Tool 1 talks to an API.
- MCP Server for Tool 2 talks to a database.
- MCP Server for Tool 3 could talk to another service or local file system.

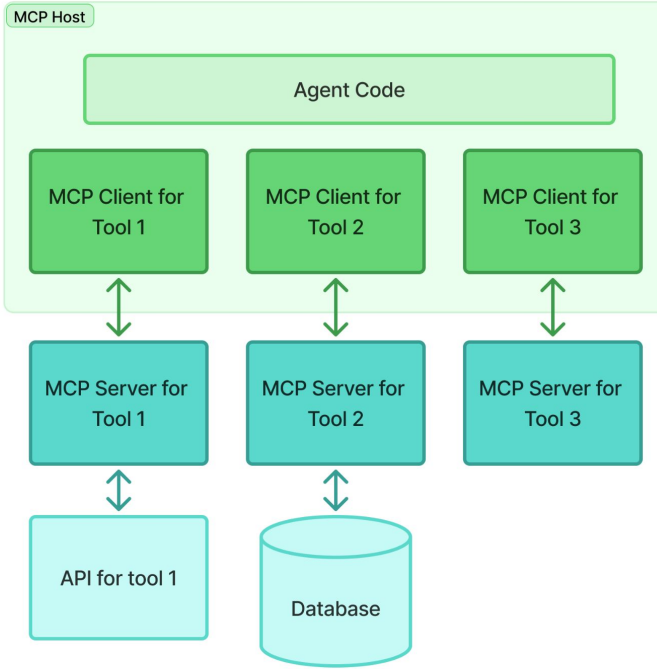
In short: Servers are the actual doers that know how to talk to real tools.



Flow Example (Step-by-Step)

Let's follow what happens when the agent needs something:

- Agent Code decides to get some data.
- It sends a request to the MCP Client for Tool 2.
- That client passes the request to MCP Server for Tool 2.
- The server fetches info from the database.
- The server sends the result back to the client, which returns it to the agent.



So the flow is:

Agent → Client → Server → External Tool → Server → Client → Agent

Using MCP

AMERICAN UNIVERSITY OF BEIRUT

1 Choose MCP servers

Pick from pre-built servers for popular tools like GitHub, Google Drive, Slack and hundreds of others. Combine multiple servers for complete workflows, or easily build your own for custom integrations.

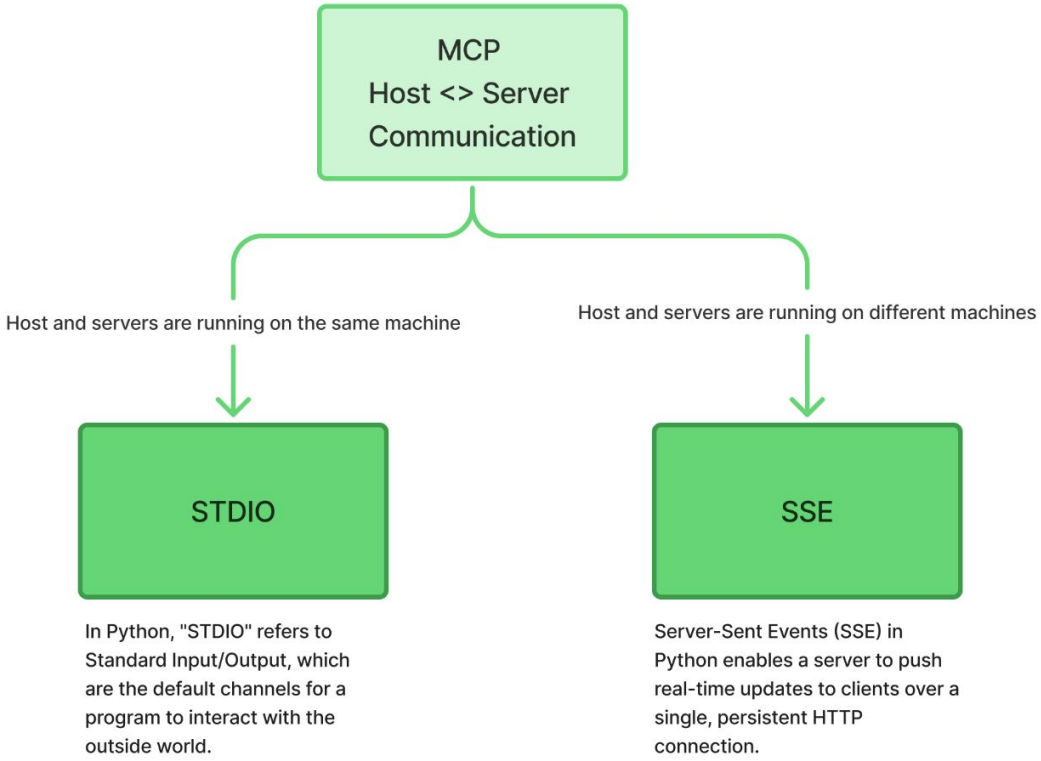
2 Connect your AI application

Configure your AI application (like Claude, VS Code, or ChatGPT) to connect to your MCP servers. The application can now see available tools, resources and prompts from all connected servers.

3 Work with context

Your AI-powered application can now access real data, execute actions, and provide more helpful responses based on your actual context.

Communication between Host and Servers



To understand how MCP works exactly, let's implement it in action.

MCP In Action

- Luckily, Anthropic released an MCP Library that we can use to avoid implementing all host, server, and client communication from scratch.
- The **FastMCP** server is your core interface to the MCP protocol.
- It handles **connection management, protocol compliance,** and **message routing**

```python
from mcp.server.fastmcp import FastMCP

# Create an MCP server
mcp = FastMCP("Demo")
```

MCP In Action

- To add a tool to your MCP server, you can use the **@mcp.tool** decorator

```python
from mcp.server.fastmcp import FastMCP

# Create an MCP server
mcp = FastMCP("Demo")


# Add an addition tool
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b
```

MCP In Action

- To add a resource to your MCP server, you can use the **@mcp.resource** decorator
- Resources are used to **expose data** to the agent. (think of these sort of like GET endpoints; they provide data but shouldn't perform significant computation or have side effects)

```python
from mcp.server.fastmcp import FastMCP

# Create an MCP server
mcp = FastMCP("Demo")


# Add an addition tool
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b


# Add a dynamic greeting resource
@mcp.resource("greeting://{name}")
def get_greeting(name: str) -> str:
    """Get a personalized greeting"""
    return f"Hello, {name}!"
```

MCP In Action

- To add a **prompt template** to your MCP server, you can use the **@mcp.prompt** decorator
- These are **reusable templates** for LLM interactions.

```python
from mcp.server.fastmcp import FastMCP

# Create an MCP server
mcp = FastMCP("Demo")


# Add an addition tool
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b


# Add a dynamic greeting resource
@mcp.resource("greeting://{name}")
def get_greeting(name: str) -> str:
    """Get a personalized greeting"""
    return f"Hello, {name}!"


# Add a prompt
@mcp.prompt()
def greet_user(name: str, style: str = "friendly") -> str:
    """Generate a greeting prompt"""
    styles = {
        "friendly": "Please write a warm, friendly greeting",
        "formal": "Please write a formal, professional greeting",
        "casual": "Please write a casual, relaxed greeting",
    }

    return f"{styles.get(style, styles['friendly'])} for someone named {name}."
```

MCP In Action

The previous examples would be the **server file**. To run your **MCP Server** locally, you need to perform the following steps

```
uv run mcp dev server.py

# Add dependencies
uv run mcp dev server.py --with pandas --with numpy

# Mount local code
uv run mcp dev server.py --with-editable .
```

Running MCP with OpenAI's Agents

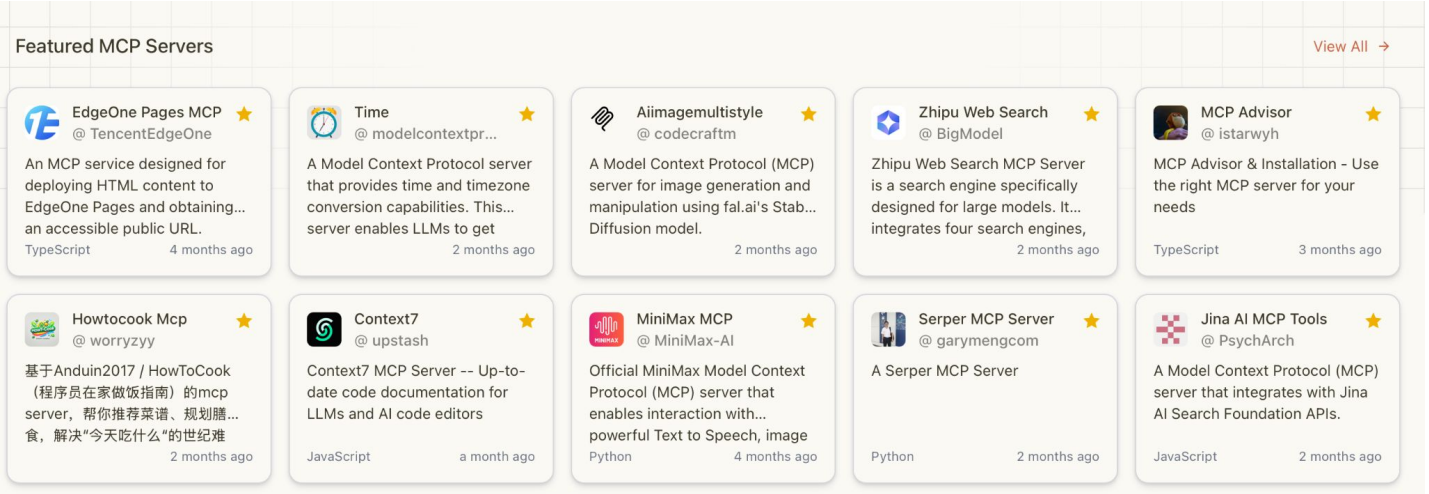
- Recently, OpenAI updated their Agents SDK to take MCP servers as a separate input.
- Before then, we needed to parse the tools to input in the 'tools' parameter of the agent.

```python
async def get_accounts_tools_openai():
    openai_tools = []
    for tool in await list_accounts_tools():
        schema = {**tool.inputSchema, "additionalProperties": False}
        openai_tool = FunctionTool(
            name=tool.name,
            description=tool.description,
            params_json_schema=schema,
            on_invoke_tool=lambda ctx, args, toolname=tool.name: call_accounts_tool(toolname, json.loads(args))

        )
        openai_tools.append(openai_tool)
    return openai_tools
```

Browsing MCP Servers

Just like agents, MCP servers have their own marketplace created by the community like mcp.so, glama.ai/mcp, and smithery.ai



Thank You