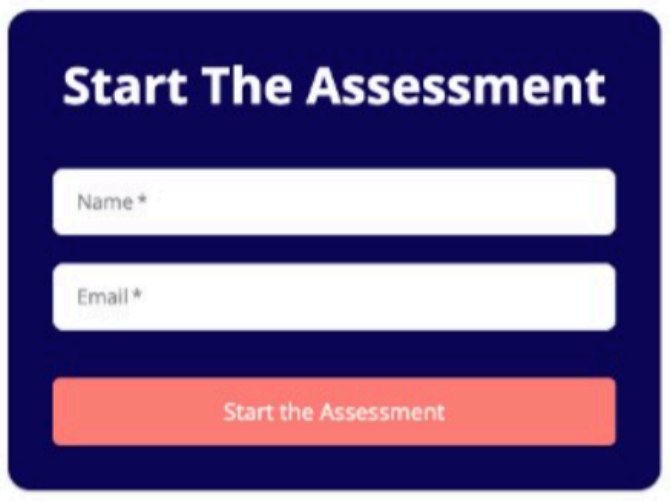
Make the Most of This Book and Your Time

Reading this book takes approximately 3 hours. However, not every chapter may be relevant to your current skill level. To help you focus on what matters most, we offer a quick 2-minute assessment. Based on your responses, we'll recommend the chapters most suited to your needs.

Are You MCP-Aware?

Take a quick quiz—just 8 yes/no questions—and receive a personalized list of chapters to improve your MCP skills.

Start Your Personalized Assessment



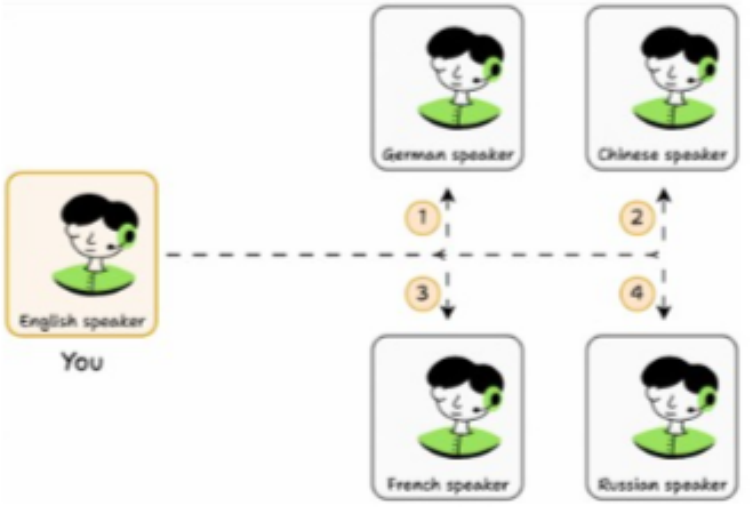
This assessment takes less than 2 minutes to complete.

Table of contents

Model Context Protocol (MCP)

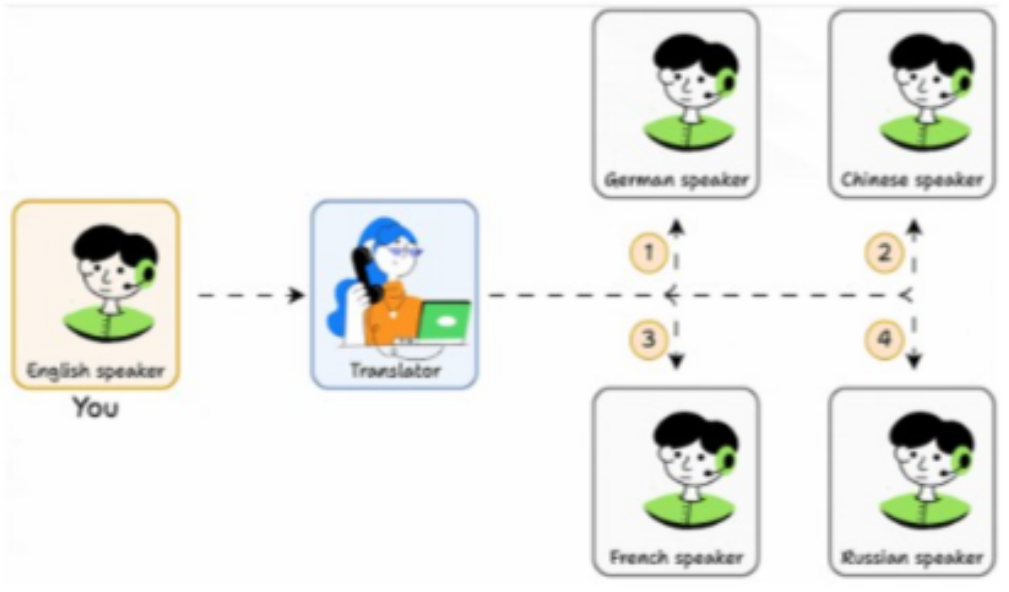
What is Model Context Protocol (MCP)?

Imagine you speak only English, and you want to communicate with others who speak different languages:



- To talk to someone in French, you'd have to learn French.
- For German, you'd need to learn German.
- And so on...

But what if there were a universal translator that could understand and translate every language for you?



Understanding MCP — A Simple Analogy

Think of MCP like a translator—it helps agents (like AI models) communicate with various tools and systems using a single, standardized interface.

While large language models (LLMs) are incredibly knowledgeable and capable of reasoning, they are limited by the information available in their training data. This means they can't naturally access live or real-time data unless given a way to do so.

What is MCP?
MCP (Model Context Protocol) is a unified interface that bridges AI applications with external resources such as:

- Web APIs
- Databases
- GitHub
- Gmail
- Slack
- Local file systems

If an AI system needs real-time information or external capabilities, MCP enables that interaction smoothly—without the need for custom integrations each time.

Just like USB-C provides a common standard to connect different devices, MCP standardizes communication between AI models and external tools, making integration efficient, scalable, and universal.

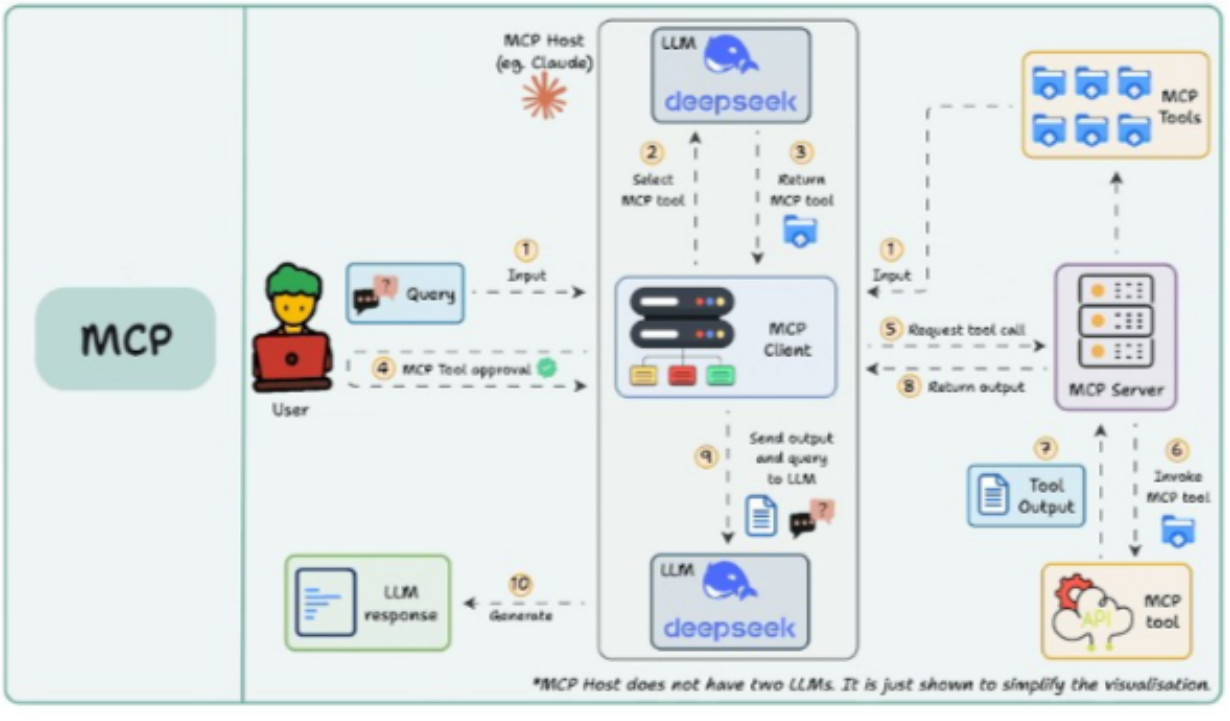
Why Was MCP Created?

- Before MCP, integrating new tools or models into AI systems was complex and inefficient.

- Imagine having three AI applications and three external tools. Without a standard protocol, you'd need to build nine separate integration modules—one for every possible combination. This approach quickly becomes unmanageable and doesn't scale well.

- Developers had to repeatedly build custom integrations, and tool providers faced the challenge of supporting many different, incompatible APIs to work across various AI platforms. This led to wasted effort, slower innovation, and inconsistent performance.

- MCP solves this by providing a common integration standard, streamlining the connection between AI applications and tools.

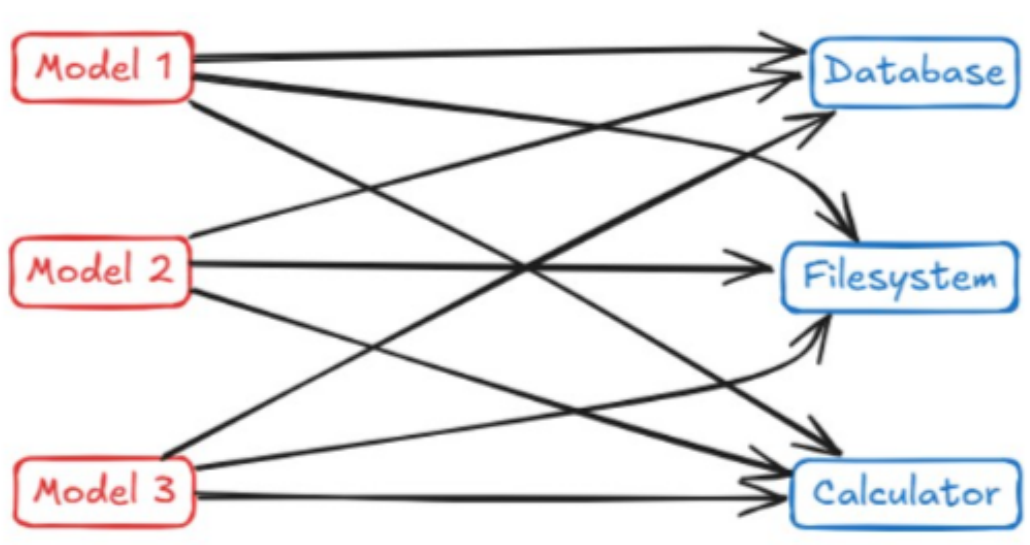


The Problem: Complex and Unscalable AI Integrations

Before MCP, connecting AI systems to external tools and data sources was chaotic. The ecosystem was filled with one-off solutions that lacked scalability and consistency.
Developers had to:

- Hard-code logic for every tool,
- Manage fragile prompt chains,
- Or rely on closed, vendor-specific plugin systems.
-

This created the well-known M × N integration nightmare:
If you had M AI models and N tools, you would need to build and maintain M × N unique integrations.



The Solution: Simplifying with MCP

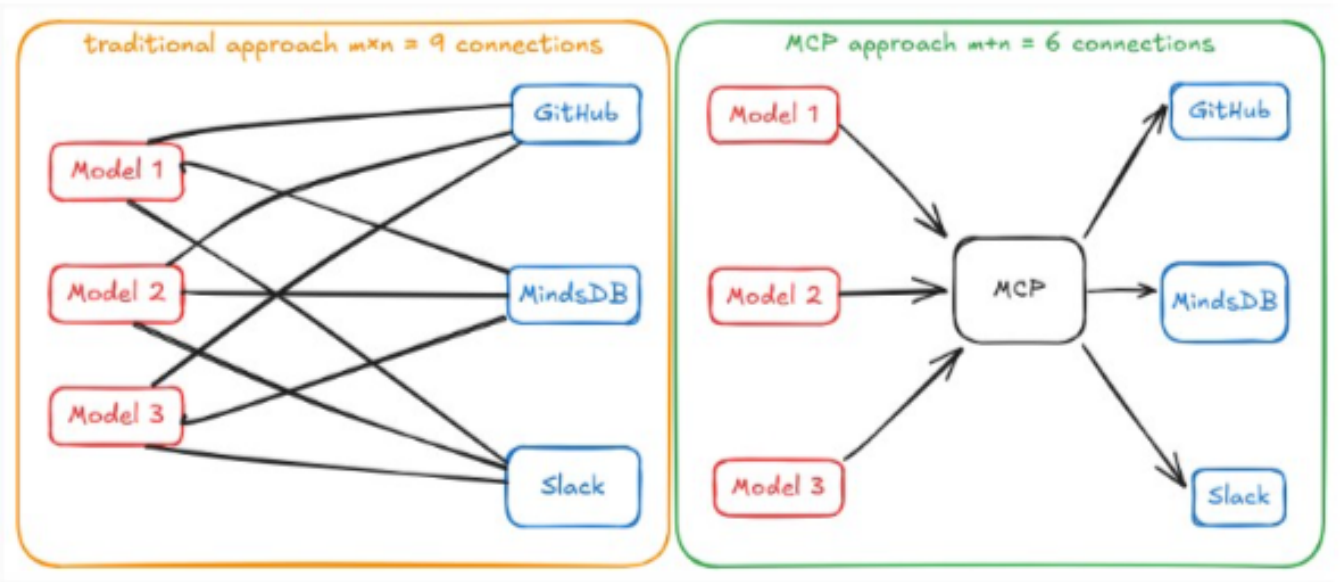
MCP solves this by placing a universal interface between AI models and tools.
Instead of building M × N direct links, you only need:

- M adapters for the AI models, and
- N adapters for the tools.

How MCP Simplifies Integration

In the traditional setup, every AI model had to create a direct integration for every external tool—resulting in a complex and tangled system of connections.
With MCP:

- Each AI model only needs to implement the MCP client once.
- Each external tool (like GitHub, Slack, MindsDB, etc.) only needs to implement the MCP server once.



Visual Comparison:

Left (Traditional Approach):

- Every model connects individually to every tool (M × N connections = 9 in this example).

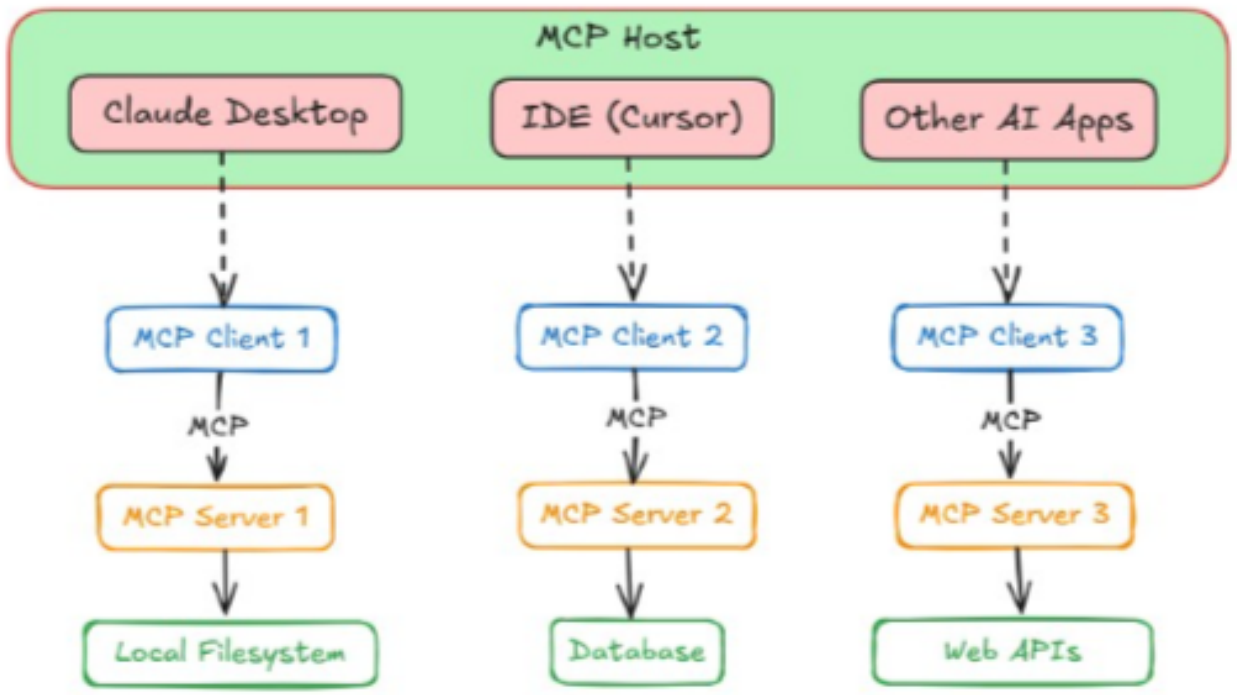
Right (MCP Approach):

- All models and tools connect to a shared MCP layer—drastically reducing integration effort (M + N connections = 6 here).

MCP Architecture Overview

At its core, MCP uses a client-server architecture, similar to how websites or networks operate. But here, the components are tailored for the AI ecosystem. There are three key roles in this architecture:

- Host
- Client
- Server



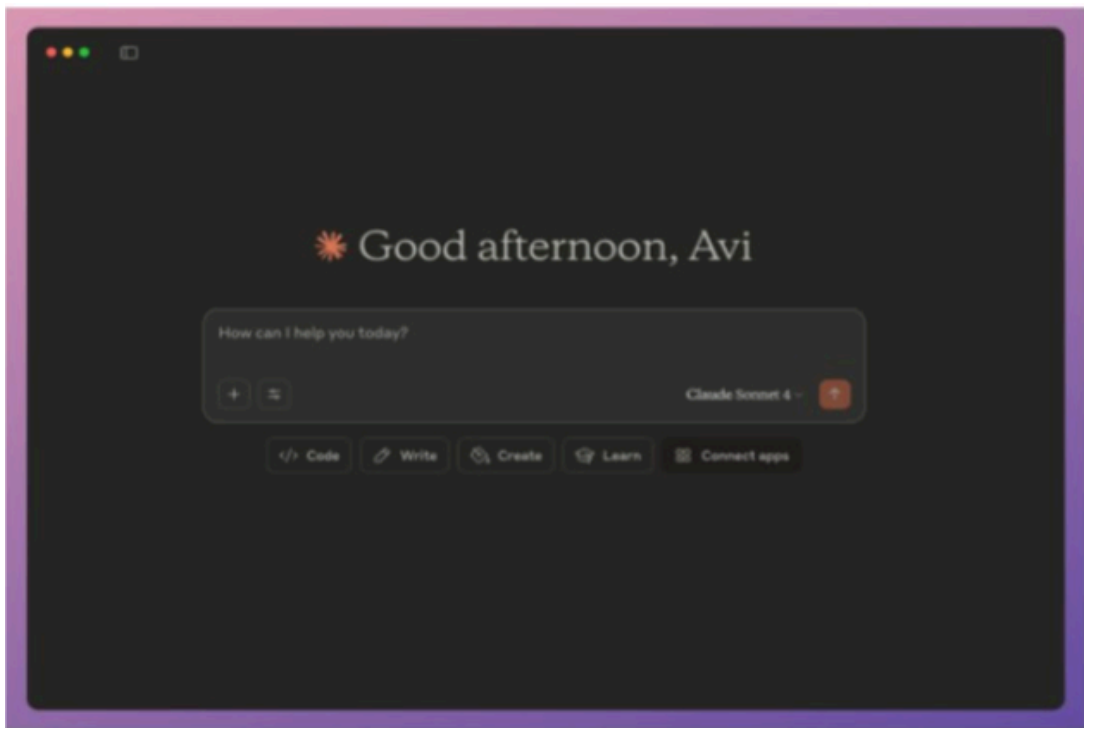
The Host is the interface where the AI model lives and interacts with the user. It's essentially the “face” of the AI system.

Examples include:

- Chat apps like ChatGPT or Claude desktop.
- AI-enabled IDEs such as Cursor.
- Custom-built apps that embed AI assistants like Chainlit.

The Host is responsible for:

- Initiating communication with MCP Servers.
- Capturing and processing user input.
- Managing conversation context.
- Displaying responses from the model.

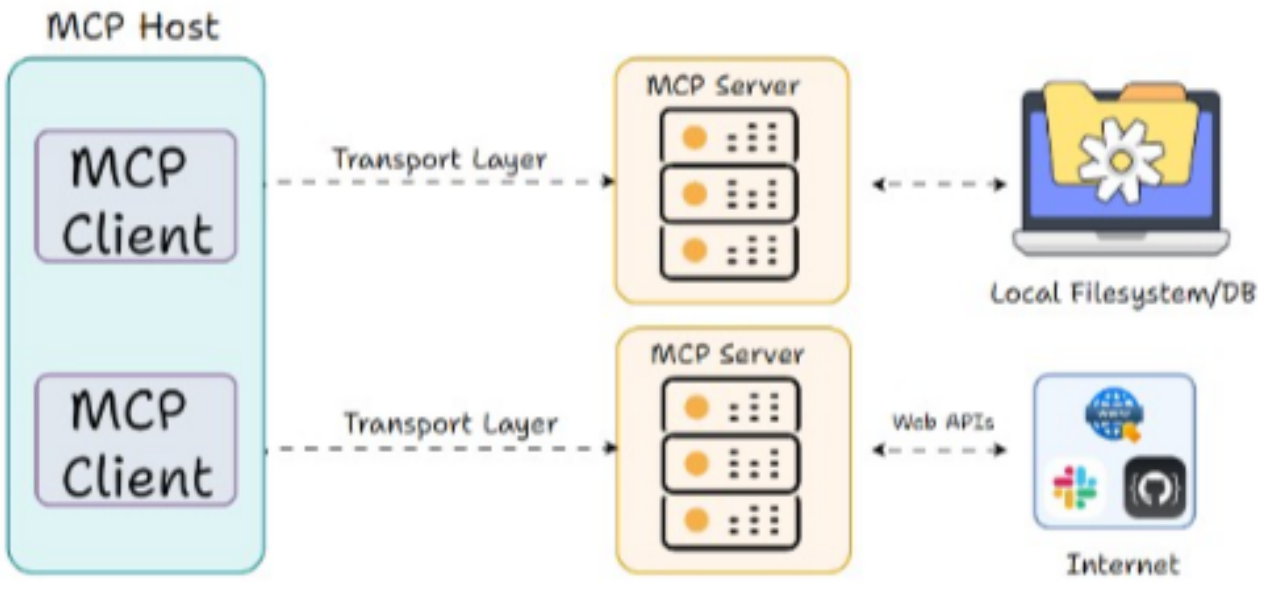


Client

The MCP Client is a crucial component within the Host. Its job is to handle the actual communication between the Host and the MCP Server.

Key Responsibilities of the Client:

- Establish a connection to the MCP Server through a defined transport layer.
- Send precise instructions from the Host to the right server (e.g., file request, API call).
- Receive responses and relay them back to the Host.

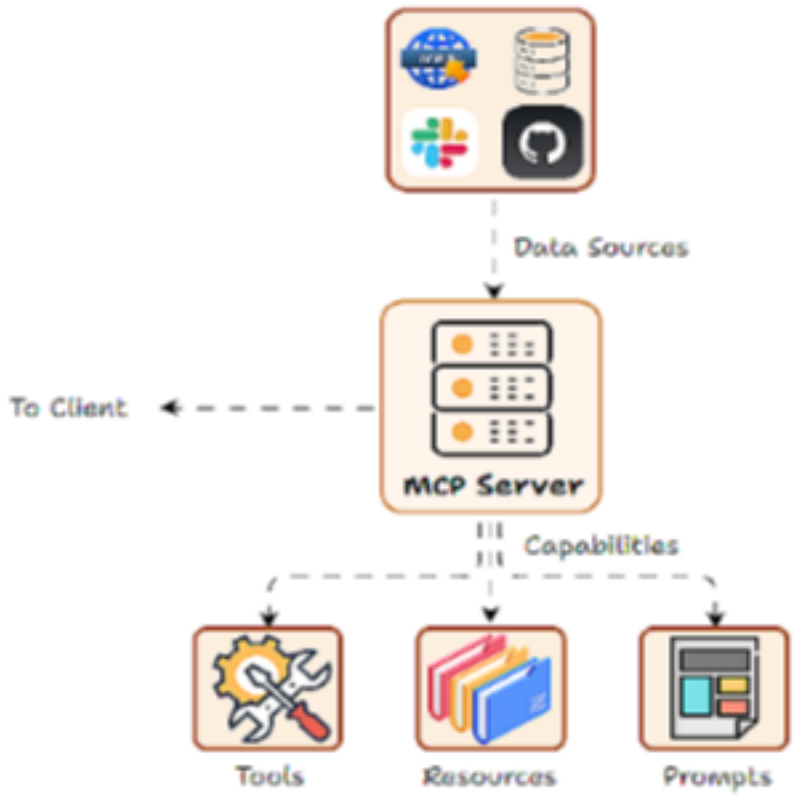


Server

The MCP Server is the engine behind the scenes. It provides the actual capabilities, such as access to tools, data sources, and resources that AI applications need to function effectively.

Key Features:
Standardized Interface: The Server advertises its available actions and resources in a format all MCP Clients understand.
Flexible Deployment: It can run locally on the same machine as the Host or remotely in the cloud, depending on system design.
Execution Layer: When the Client sends a request (e.g., fetch data or call an API), the Server processes it and returns the result.

Capabilities Provided by MCP Server:
Tools (e.g., code execution, APIs)
Resources (e.g., files, databases)
Prompts (e.g., reusable instructions or templates)



Core Capabilities: Tools, Resources, and Prompts

- The MCP framework offers three main types of capabilities that an MCP Server can expose to AI applications:

- **Tools:** These are executable functions or actions the AI can trigger. They often interact with the outside world, such as calling APIs, sending emails, or running computations. Tools may have side effects.

- **Resources:** These are read-only data sources. The AI can query them to retrieve information (e.g., databases, files) without modifying anything—just fetch and use.

- **Prompts:** These are predefined templates or workflows designed to guide the AI's behavior. They help standardize how specific tasks are handled across tools and apps.

tool_example.py Copy

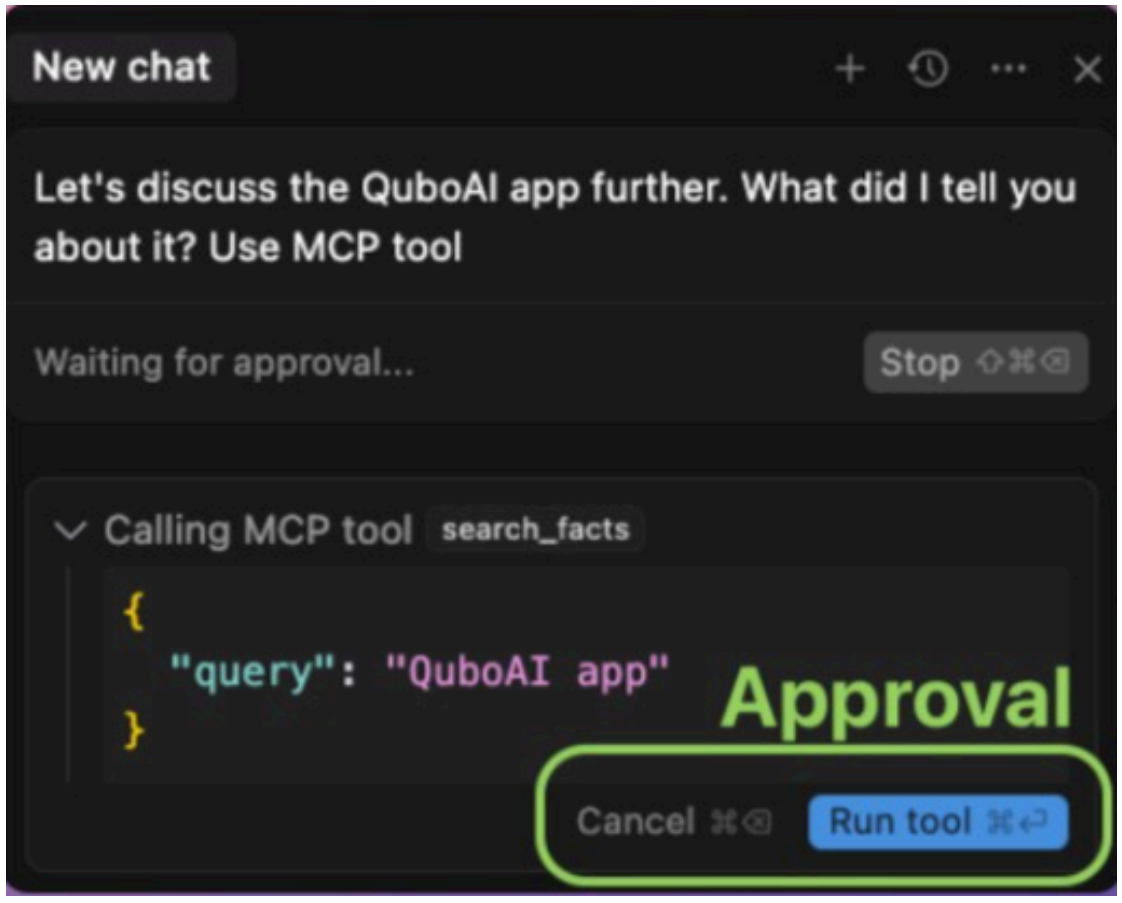
```python
@mcp.tool()
def get_weather(location: str) → dict:
    """Get the current weather for a specified location."""
    # (In a real implementation, call an external weather service)
    return {
        "temperature": 72,
        "conditions": "Sunny",
        "humidity": 45
    }
```

How it Works:

- The AI (via the Host) issues a tool call like:
- {"name": "get_weather", "args": {"location": "San Francisco"}}
- The MCP Server runs the get_weather("San Francisco") function.
- The output (a dictionary of structured weather data) is sent back to the Client.
- The Client forwards this data to the AI, which can then use it to craft a response.

Why It Matters:

- Tools return structured data like temperature and humidity.
- The AI can use this data directly or convert it into a natural language reply.
- Since tools may involve file access, API requests, or system-level actions, user permission is typically required before execution.



For instance, Claude's client interface might display a prompt like, "The AI is requesting access to the 'get_weather' tool—do you approve?" This mechanism prevents misuse and ensures that humans retain authority over critical actions.

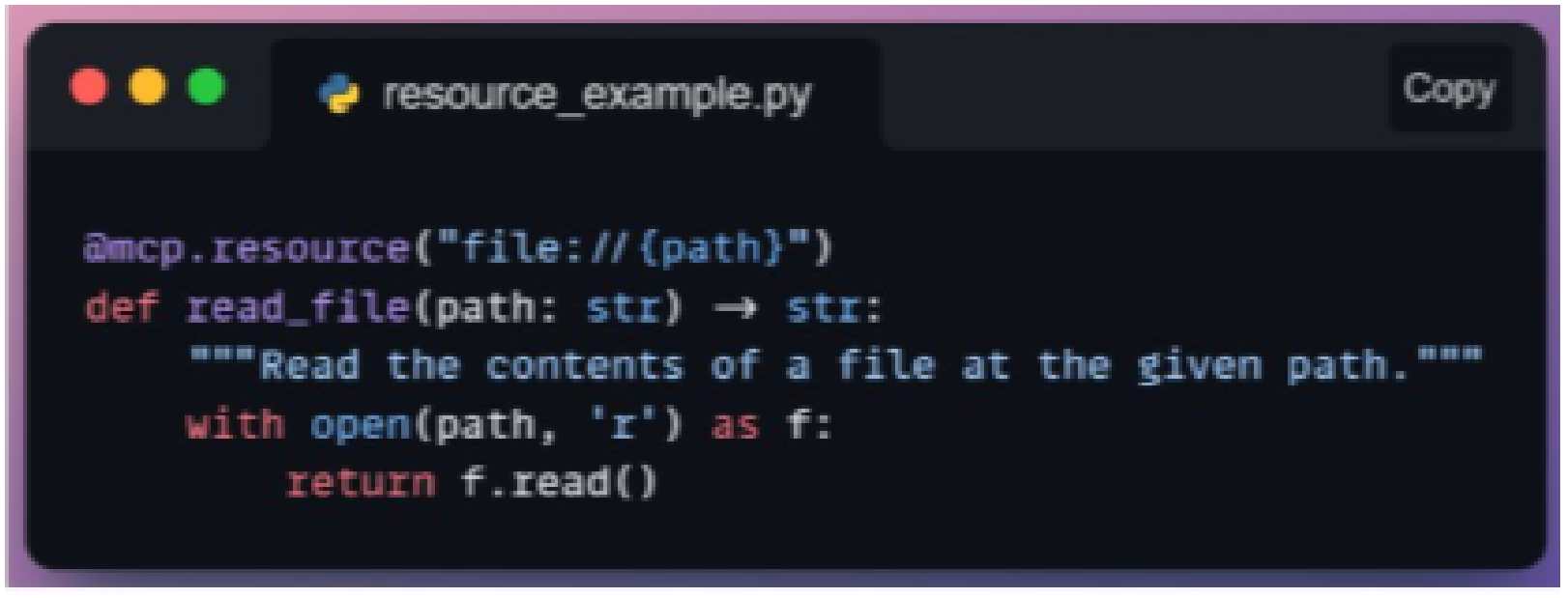
Resources

- Resources offer read-only access to information for the AI system.
- Think of them as structured data sources—like knowledge bases or databases—that the AI can retrieve facts from but cannot alter.
- Unlike tools, resources are lightweight in terms of processing and have no side effects, making them ideal for simple lookups.

- Suppose a user says, “Refer to the company handbook to respond to my question.” In this case, the Host could fetch a resource that extracts relevant sections from the handbook and provides them to the AI model.

- Resources can include various types of read-only data such as the contents of local files, excerpts from documentation or knowledge bases, query results from databases, or static configuration settings.

- Essentially, any contextual information that helps the model reason or respond more effectively qualifies as a resource. For example, an AI research assistant might query an “ArXiv papers database” to fetch abstracts or citations on request.



Notably, resources are referenced using standardized identifiers—typically URIs or predefined names—rather than arbitrary function calls.

- Resources are often controlled by the application itself, meaning the app determines when and how to fetch them—preventing the AI model from freely accessing everything at will.
- Since resources are read-only, they pose minimal risk in terms of system integrity. However, privacy and access permissions must still be carefully managed to prevent unauthorized reading of sensitive files.
- The Host or server can enforce rules around which resource URIs the AI can access, effectively gating access to particular types of data.

Prompts

- Within the MCP architecture, prompts are a unique and essential concept. These are predesigned templates or structured conversation flows that help guide how the AI responds or behaves.

- Prompts function like reusable, preconfigured instruction sets—such as example dialogues or task-specific cues—that help the AI stay on track during a given task.

- Why are these treated as capabilities?
- Consider common usage patterns: for example, a prompt that automatically configures the AI with a system role like “You are a code reviewer,” and then inserts a user’s code for evaluation.

- Instead of embedding such logic directly in the host app, the MCP server can dynamically inject these prompts when needed.

- These advanced prompts can be triggered on demand to tailor the model's responses to specific scenarios.
- When it comes to control, prompts are generally governed either by the user or the developer.
- For example, a user might choose a predefined prompt from a UI—such as a "Summarize this document" option. The host then fetches the corresponding prompt definition from the server.
- Unlike tools, prompts are not invoked spontaneously by the model. They are explicitly selected to define the context or intention before the AI begins its generation process.
- Typically, prompts are loaded at the start of an interaction or when the user selects a specific mode of operation.

prompt_example.py

```python
@mcp.prompt()
def code_review(language: str) -> list:
    """Provide a structured prompt for reviewing code in the given language."""
    return [
        {"role": "system", "content": f"You are a meticulous {language} code reviewer..."},
        {"role": "user", "content": f"Please review the following {language} code:"}
    ]
```

- This function returns a series of message objects formatted for AI input (compatible with OpenAI's structure), simulating a code review context.

- When the host triggers this prompt, it retrieves the predefined message structure and dynamically inserts the user's actual code snippet into it.

- The AI then receives these messages before generating its response—allowing the server to frame the conversation more effectively.

- Hosting prompts on the server allows them to be updated or enhanced without requiring any changes to the client application. This also enables different servers to provide their own tailored sets of prompts.

- A key insight is that prompts—as a capability—effectively blur the boundary between data and instructions, acting as both informational input and behavioral guidance.

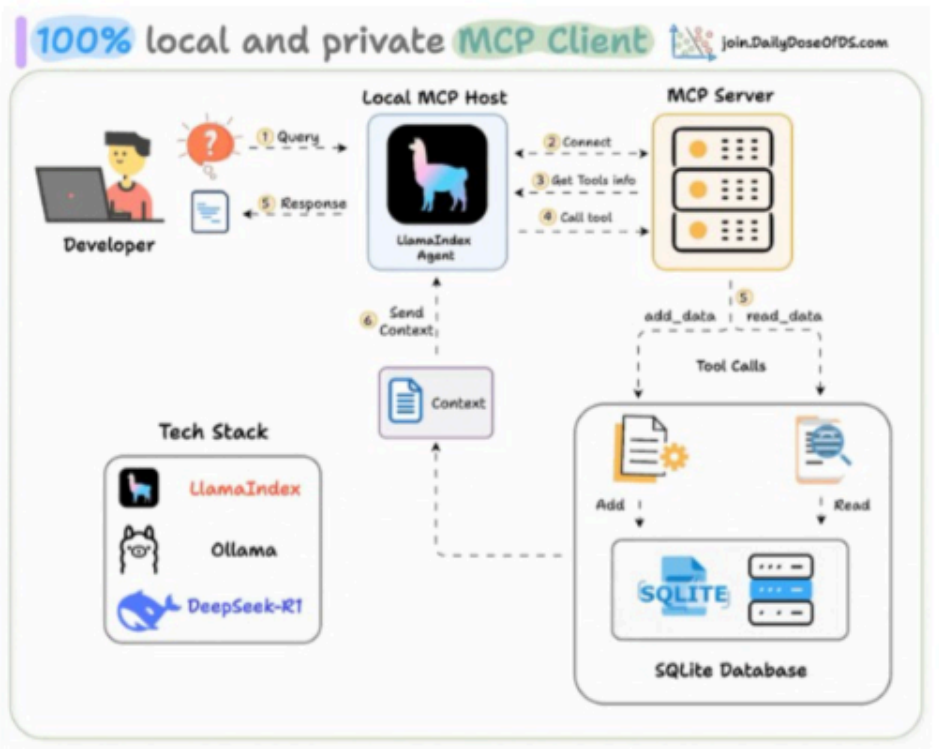
- They embody standardized strategies and best practices that the AI can follow.

- In essence, MCP prompts work similarly to how ChatGPT plugins guide query formatting, but with the added benefit of being standardized, discoverable, and integrated into the communication protocol.

"MCP Projects"

Fully Local MCP Client Setup

An MCP client acts as a bridge within an AI application (like Cursor), enabling connections to external tools. Here's how to set up a fully local version for privacy and control.



Tech Stack Used:

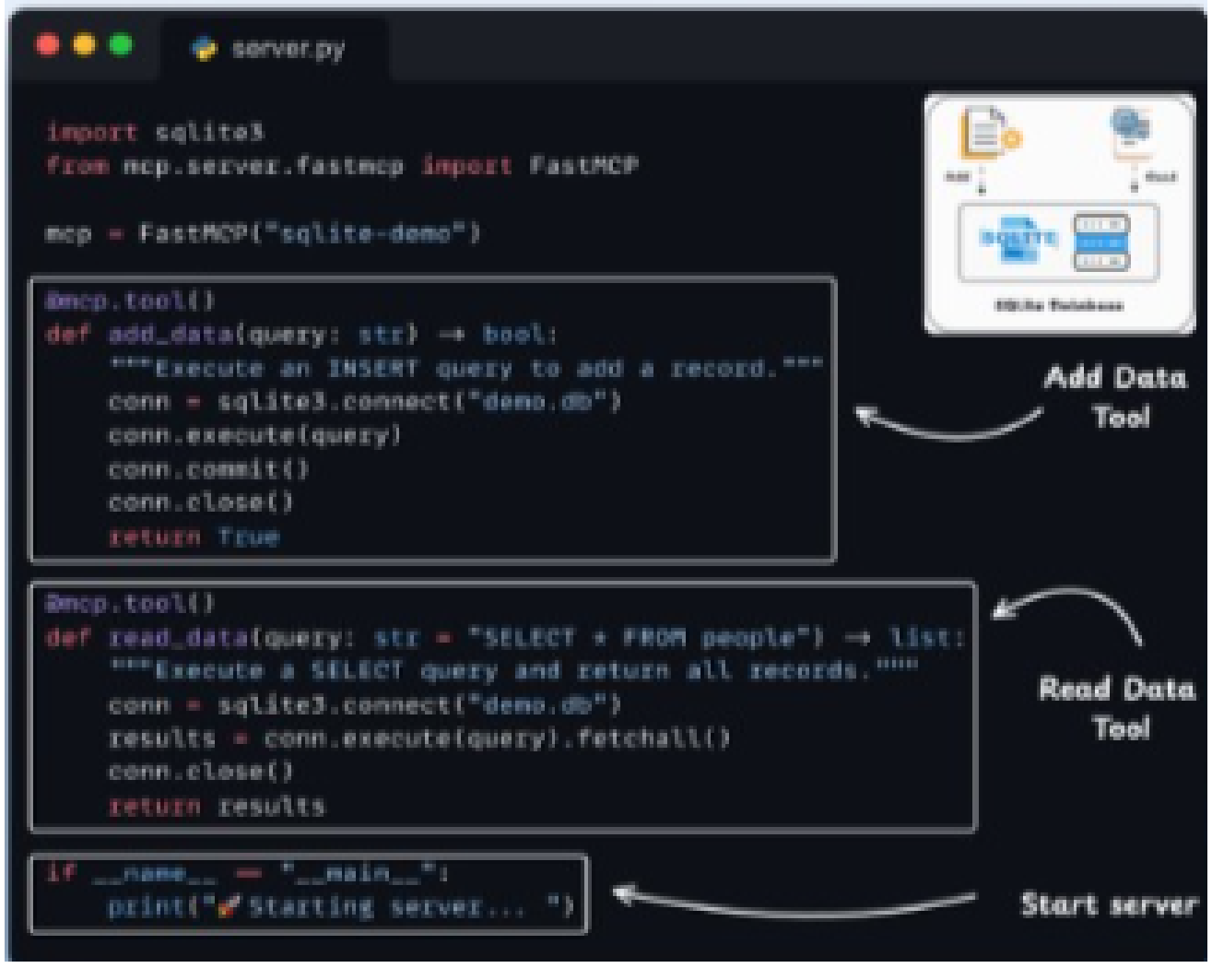
- LlamaIndex – For building the MCP-compatible intelligent agent.
- Ollama – To locally run and serve the Deepseek-R1 language model.
- LightningAI – For managing development and hosting environments.

Operational Flow:

- The user submits a query to the AI system.
- The MCP agent connects with the MCP server to identify available tools.
- The agent selects and runs the appropriate tool based on the query, retrieving the necessary context.
- The agent formulates and returns a context-enriched response.

Step 1: Create an SQLite-Based MCP Server

- In this demonstration, we've implemented a lightweight MCP server powered by SQLite. It offers two essential tools:
- Add Data – to insert new entries into the database
- Fetch Data – to retrieve stored records
- While we use SQLite for simplicity, this architecture allows the client to connect to any compliant MCP server.



Step 2: Set Up the Language Model (LLM)

- We'll use Deepseek-R1, hosted locally via Ollama, as the large language model (LLM) powering our MCP agent.
- This ensures full local control over model inference without relying on external APIs or cloud infrastructure.

ollama-client.py

Deepseek-R1

```python
from llama_index.llms.ollama import Ollama
from llama_index.core import Settings

llm = Ollama(model="deepseek-r1", request_timeout=120.0)
Settings.llm = llm
```

Step 3: Define system prompt

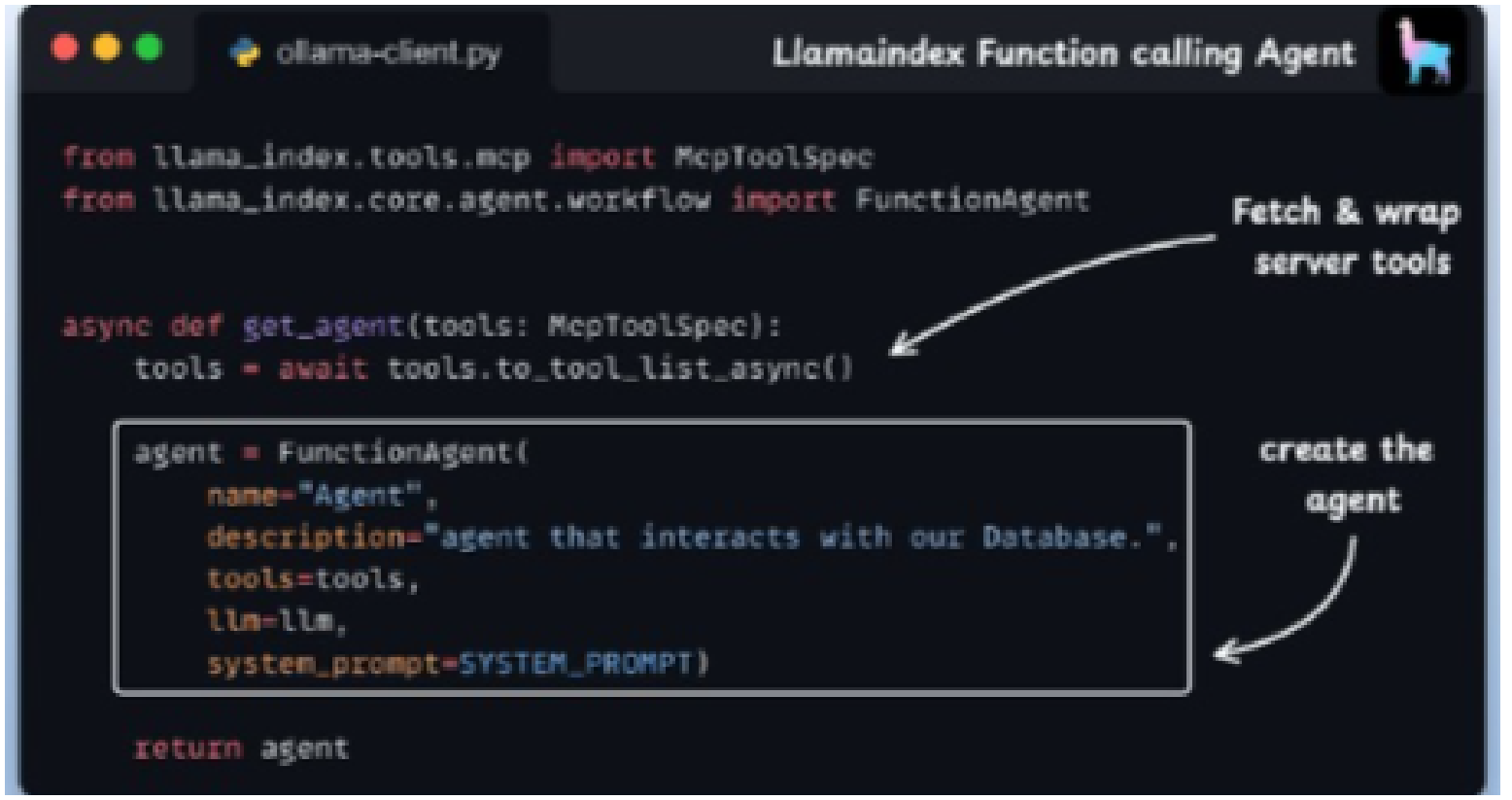
- We define our agent's guiding instructions to use tools before answering user queries.
- Feel free to tweak this on a need basis.

ollama-client.py System Prompt

```
SYSTEM_PROMPT = """\

You are an AI assistant for Tool Calling.
Before helping, work with our tools to interact
with our database.

"""
```

Step 4: Define the Agent

- Next, we define a function to create a LlamaIndex-based agent, passing all required parameters including tools, model, and the system prompt.
- The tools are MCP-compatible and are wrapped by llama_index so they can be seamlessly used by the FunctionAgent.



Step 5: Define How the Agent Handles Interaction

- This step sets up the logic for how user input is processed.
- We route messages to our FunctionAgent, providing it with shared memory context to handle state, manage chat history, invoke necessary tools, and return a final response.
- The agent runs with the given context and interprets streaming events—distinguishing tool invocations and results along the way.

ollama-client.py

Agent Interaction

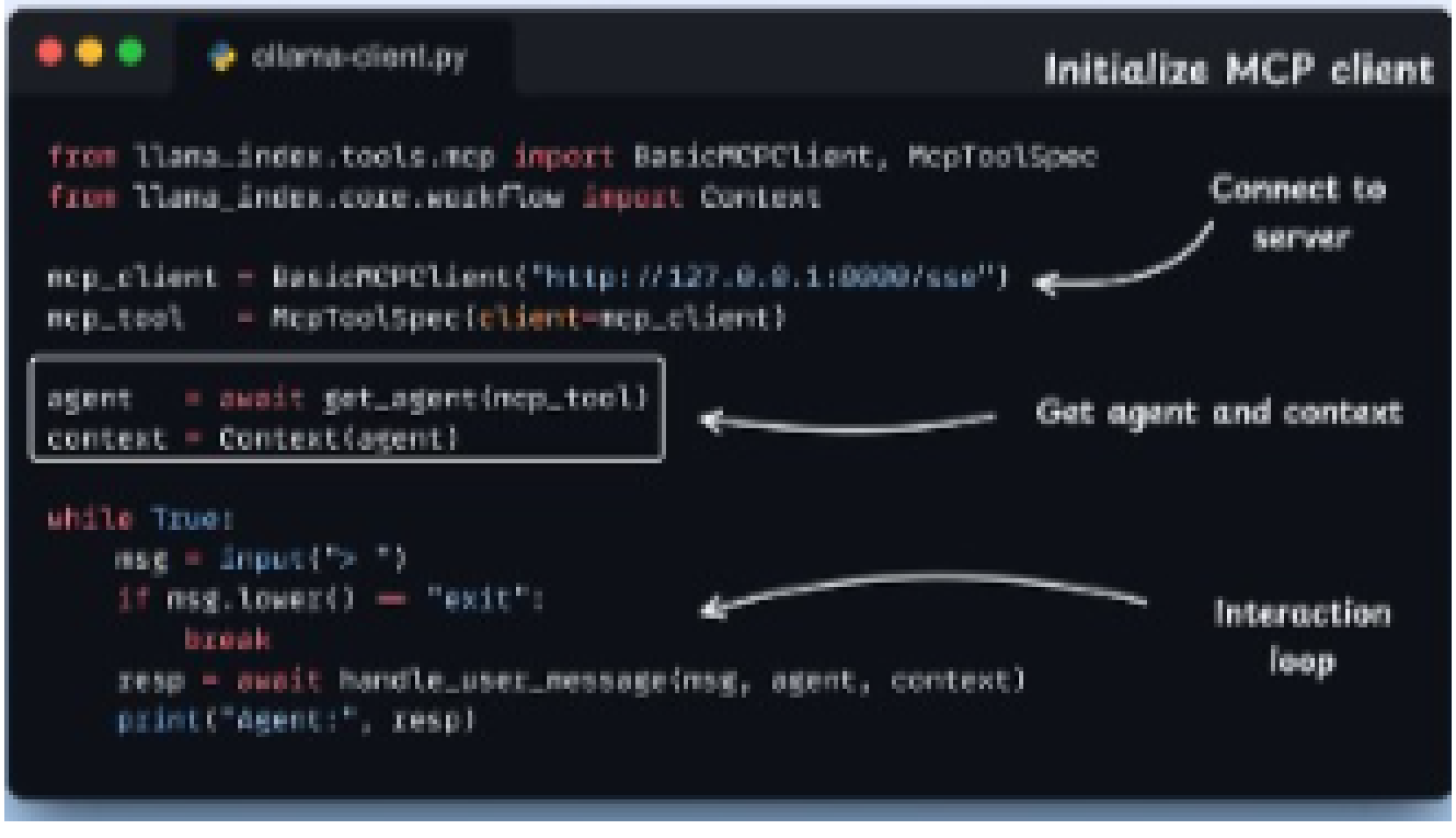
```python
from llama_index.core.agent.workflow import (
    FunctionAgent,
    ToolCallResult,
    ToolCall)

from llama_index.core.workflow import Context

async def handle_user_message(
    message_content: str,
    agent: FunctionAgent,
    agent_context: Context,
    verbose: bool = False,
):
    handler = agent.run(message_content, ctx=agent_context)
    async for event in handler.stream_events():
        if verbose and type(event) == ToolCall:
            print(f"Calling tool {event.tool_name}")
        elif verbose and type(event) == ToolCallResult:
            print(f"Tool {event.tool_name} returned {event.tool_output}")

    response = await handler
    return str(response)
```

Pass message to agent

Invoke agent

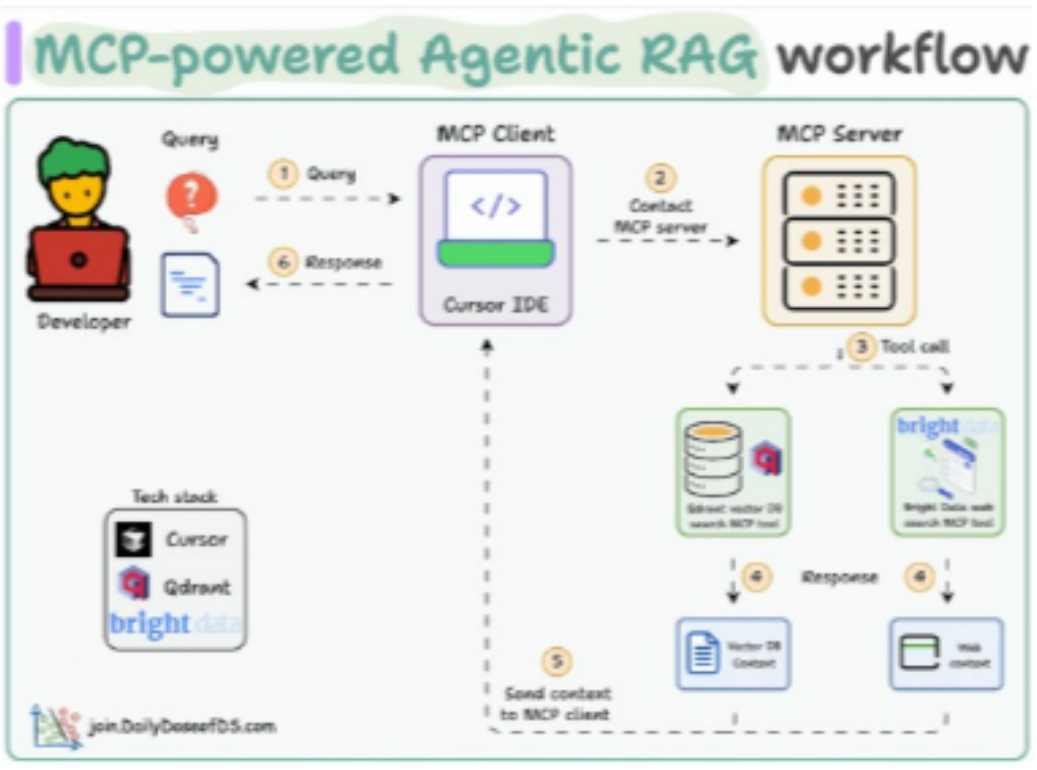
Step 6: Launch the MCP Client and Initialize the Agent

- Start the MCP client, connect it to the appropriate server, retrieve the available tools, and wrap them as callable functions.
- These tools are passed into the LlamaIndex agent, and a context manager is created to handle ongoing state during interactions.



#2) Building an MCP-powered Agentic RAG System

In this section, we'll walk through creating an Agentic Retrieval-Augmented Generation (RAG) system using MCP (Multi-Component Protocol). This setup utilizes a vector database for context retrieval and intelligently falls back to web search when needed.



Technology Stack:

- **Bright Data** – For large-scale web scraping when information is not found in the vector DB.
- **Qdrant** – Used as the vector database to store and retrieve contextual embeddings.
- **Cursor** – Serves as the MCP client interface where the query is initiated.

Step 1: Launch the MCP Server

Start by defining an MCP server using the FastMCP class. Specify the host, port, and timeout settings.

server.py — Define MCP server

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

# Create an MCP server
mcp = FastMCP("MCP-RAG-app",
              host="127.0.0.1",
              port=8000,
              timeout=30)
```

Step 2: Create a Vector DB Tool

- To expose a tool via the MCP server, ensure:
- It is decorated with the @tool decorator.
- It includes a clear and structured docstring explaining inputs and outputs.

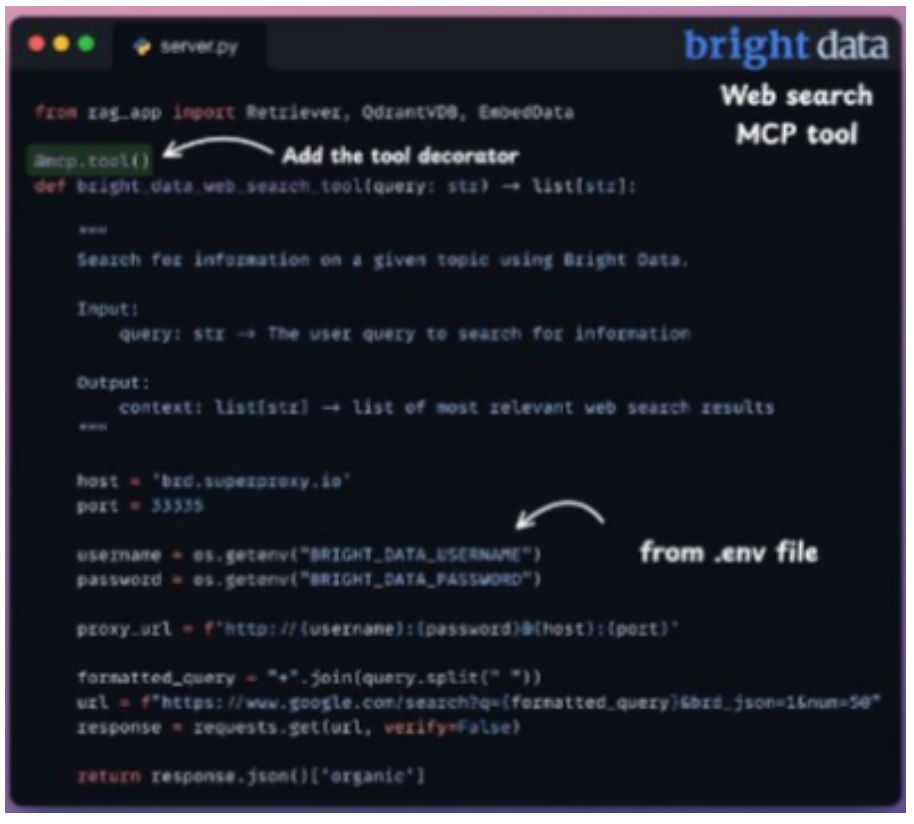
server.py — Vector DB MCP tool

```python
from rag_app import Retriever, QdrantVDB, EmbedData

@mcp.tool()   # Add the tool decorator
def machine_learning_faq_retrieval_tool(query: str) -> str:

    """Retrieve the most relevant documents from the machine learning
       FAQ collection. Use this tool when the user asks about ML.

    Input:
        query: str → The user query to retrieve the most relevant documents

    Output:
        response: str → most relevant documents retrieved from a vector DB
    """

    # check type of text
    if not isinstance(query, str):
        raise ValueError("query must be a string")

    retriever = Retriever(QdrantVDB("ml_faq_collection"), EmbedData())

    return retriever.search(query)
```

Step 3: Implement Web Search Tool

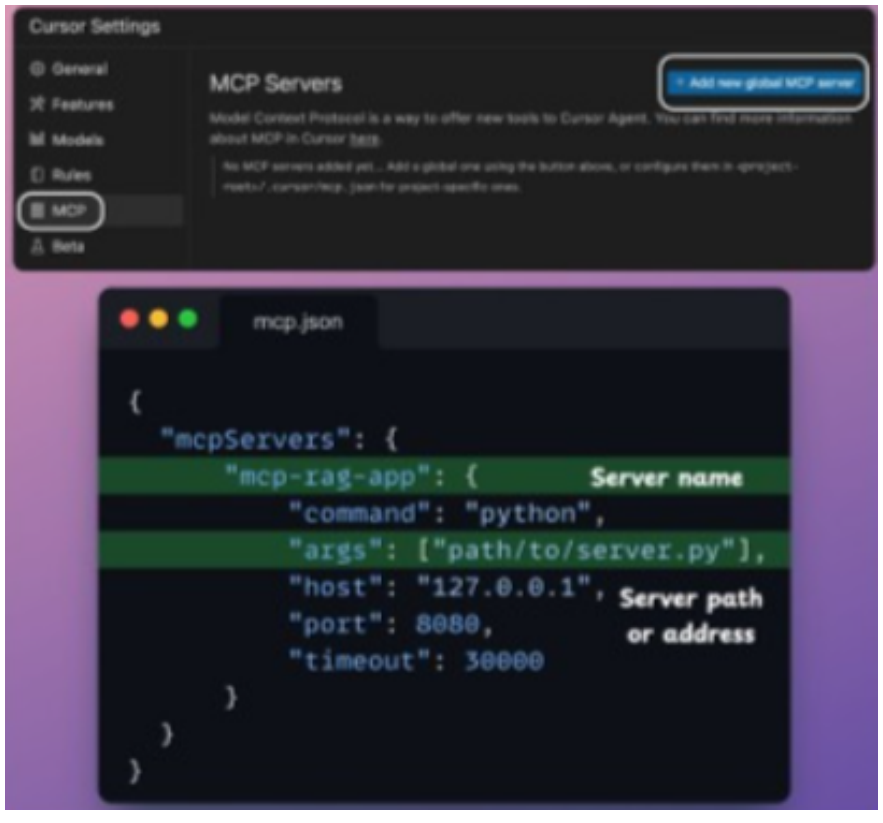
If the user query isn't related to machine learning or doesn't yield results from the vector DB, fall back to web search using Bright Data's SERP API. This allows scalable and real-time scraping from the web.

Here's the MCP tool for that:



Step 4: Connect MCP Server to Cursor

- To allow Cursor to interact with your MCP server, follow these steps:
- Open Cursor → Settings → MCP.
- Click Add new global MCP server.
- Update your MCP config file (mcp.json) with the following:
- json
- CopyEdit



You're All Set!

- Your MCP server is now running and fully integrated with the Cursor IDE. It supports two intelligent tools:
- **Bright Data Web Search Tool** – For wide-scale data collection from the internet.
- **Vector DB Search Tool** – For fast retrieval of relevant documents from a vector database.

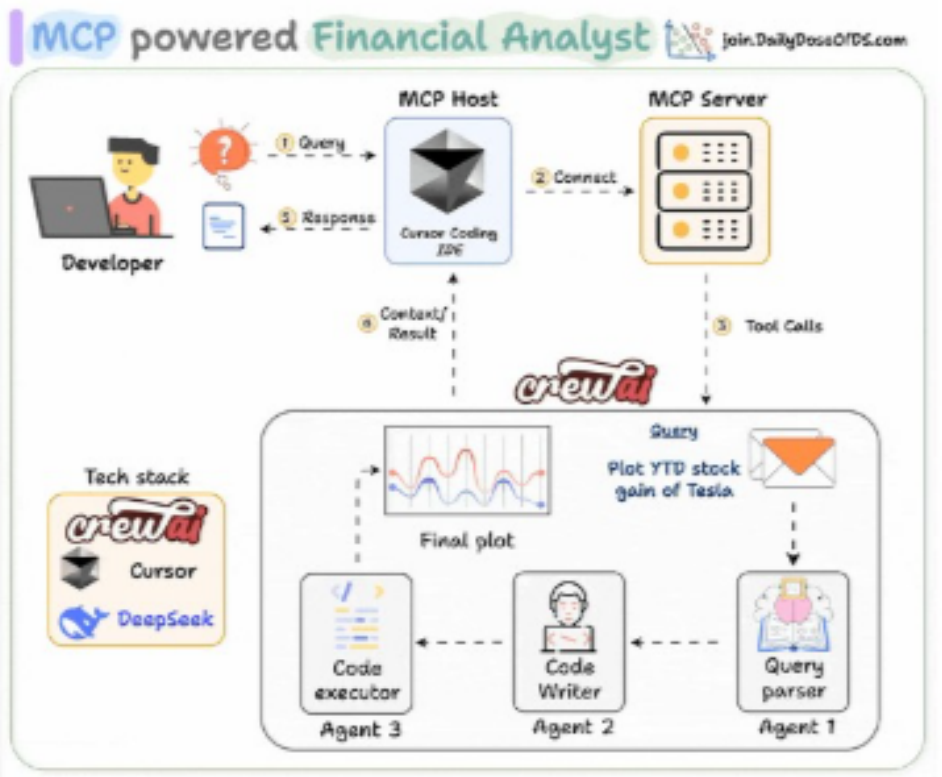
How It Works

When you issue a query:

- If it’s machine learning-related, the system routes it to the Vector DB tool.
- If it’s a general question, it falls back to the Bright Data tool to scrape real-time information from the web.

#3) MCP-powered Financial Analyst Agent

Create an intelligent financial analyst powered by MCP that can autonomously retrieve, analyze, and visualize stock market trends — directly from Cursor or Claude Desktop.



Tech Stack:

CrewAI – Enables multi-agent orchestration and delegation of specialized tasks.
Ollama + DeepSeek-R1 – Runs the large language model locally for processing financial queries.
Cursor – Acts as the MCP host, providing the interface for agent execution.