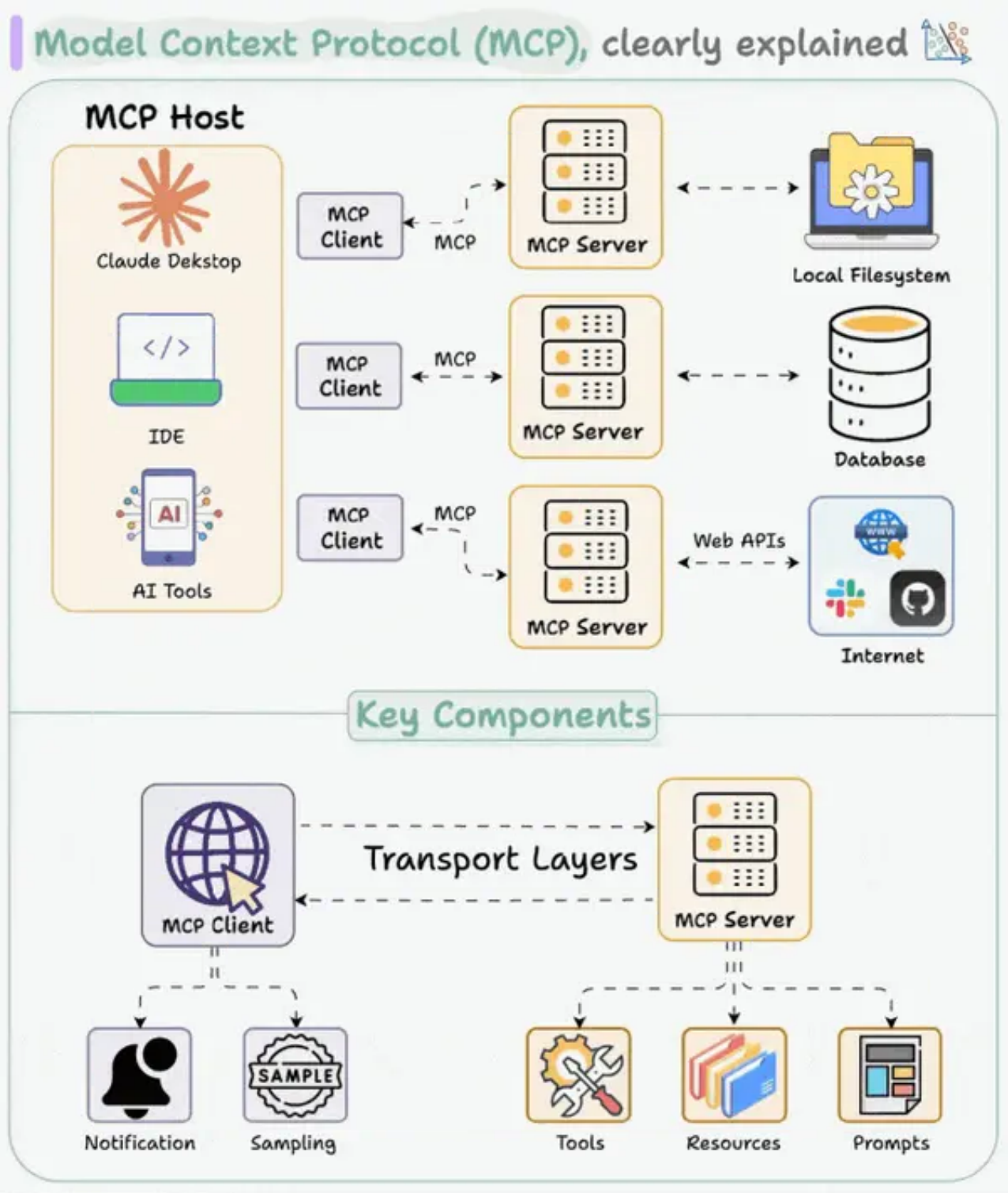
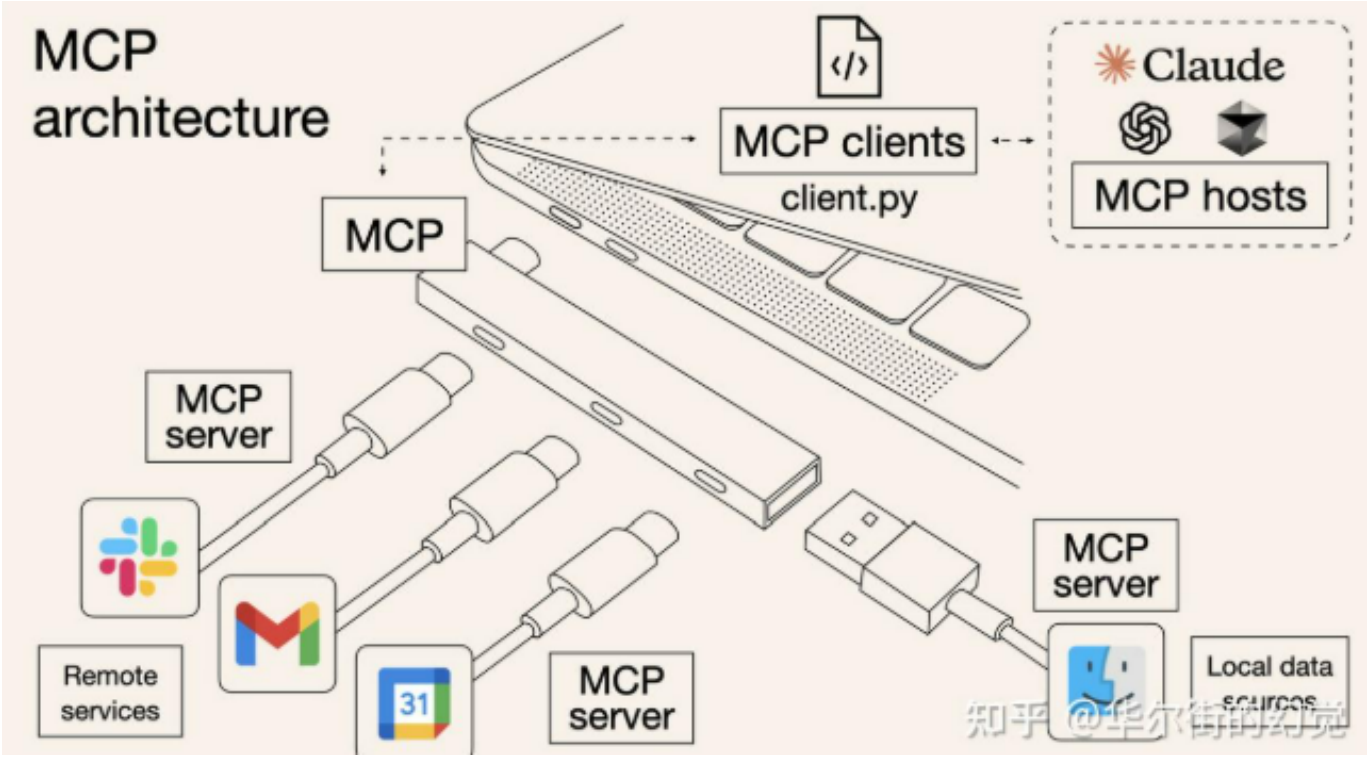
一、MCP基本介绍

定义和定位

定义：`MCP`全称 **Model Context Protocol**（中文名：模型上下文协议），是由 Anthropic 于 2024 年 11 月推出的开源协议。

定位：**旨在解决不同大语言模型（LLM）与不同外部工具集成的标准化问题**。通过MCP，开发者能够以一种**统一的方式**将各种数据源和工具连接到 AI 大模型，从而提升大模型的实用性和灵活性。（简而言之：让开发者不用重复造轮子，通过`MCP`协议，以期构建更强大的 AI Agent 生态。





用一张更常见的图来说明

> `MCP` 就像是一个“通用插头”或者“USB接口”，制定了统一的规范，不管是连接数据库、第三方API，还是本地文件等各种外部资源，都可以通过这个“通用接口”来完成，让AI模型与外部工具或数据源之间的交互更加标准化、可复用。

目前，MCP 生态已经得到了广泛的支持，包括 Anthropic 的 Claude 系列、OpenAI 的 GPT 系列、Meta 的 Llama 系列、DeepSeek、阿里的通义系列以及 Anysphere 的 Cursor 等主流模型均已接入 MCP 生态。

架构设计

> [!TIP]
>
> **调用流程：用户 → Host（界面） → MCP Client（编码/传输） → MCP Server（执行/调用资源） → 返回结果 → Host → 用户**

MCP 采用了`客户端-服务器`架构，主要包括以下三个核心组件和数据资源：

- **MCP Host（主机）**：可以理解为用户可操作的界面，例如Cursor、Trae、CherryStuido
- **MCP Client（客户端）**：一般内置于Host中，负责链接Host和Server端，发送请求并接收响应，进行消息的相互传递。
- **MCP Server（服务器）**：负责接收Client传来的指令，和周边系统（网络、数据库等等）进行互动，完成特定的任务，并将结果反馈给Client端。
- **数据资源**：MCP 服务器可访问两类数据源：
  - 一类是**本地**数据源，即用户计算机上存储和管理的数据，使用的是`STDIO协议`。
  - 一类是**远程**服务，例如通过 Web API 获取的外部数据，这使得 AI 应用能够直接访问互联网数据。使用的是`SEE协议`。

> [!NOTE]
>
> 本地数据源（Local）

- **内容**：用户设备上的文件、数据库、环境变量等。
- 通信协议：

  STDIO（标准输入/输出）

  - Server 作为子进程启动，Host 通过 stdin/stdout 与其通信。
  - 适合**本地工具集成**（如读取项目文件、执行 shell 命令）。
- **优势**：低延迟、高安全性（数据不出本地）。

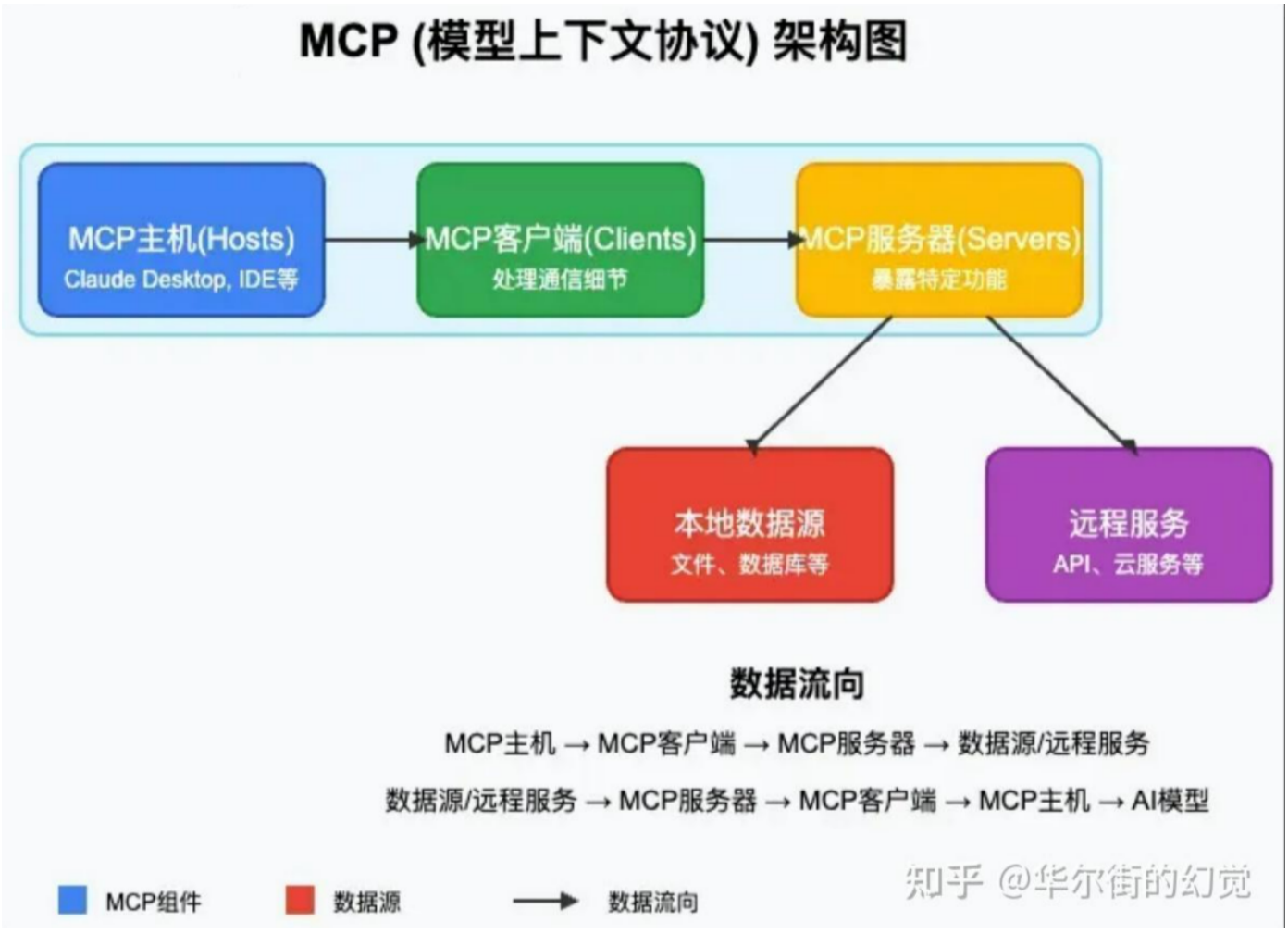
远程服务（Remote）

- **内容**：Web API（如天气服务、GitHub、数据库云服务等）。
- 通信协议：

  SEE（Secure External Execution）协议

  （注：SEE 是 MCP 中提出的一种安全远程调用机制，强调权限控制和沙箱执行）

  - 支持 HTTPS、OAuth、API Key 等标准认证。
  - 可限制访问范围（如只允许访问特定域名）。
- **优势**：扩展性强，可接入互联网生态。



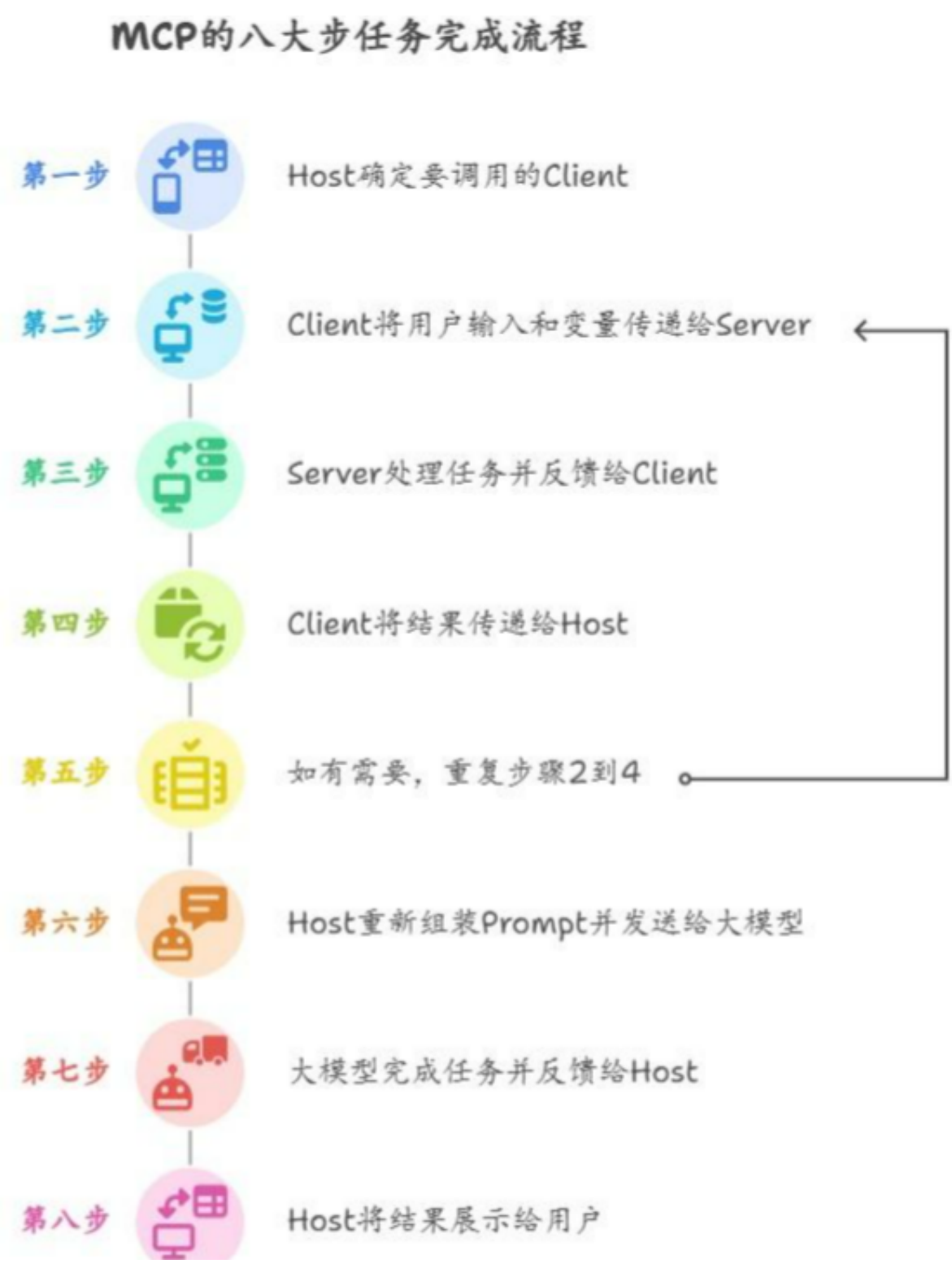
工作原理

MCP 核心是让我们能方便地调用多个工具，那随之而来的问题是 LLM（模型）是在什么时候确定使用哪些工具的呢？ `Anthropic` 为我们提供了详细的解释，当用户提出一个问题时：

- 客户端（Claude Desktop / Cursor）将问题发送给 LLM。
- LLM 分析可用的工具，并决定使用哪一个（或多个）。
- 客户端通过 MCP Server 执行所选的工具。
- 工具的执行结果被送回给 LLM。
- LLM 结合执行结果，归纳总结后生成自然语言展示给用户！

更细致的工作流程如下：用户输入要完成的任务或问题

1. 第一步：Host根据用户输入，确定要调用哪个Client；
2. 第二步：Client将用户输入和**一些环境变量**，传递给Server；
3. 第三步：Server根据Client传递的内容进行任务处理，并将结果反馈给Client；
4. 第四步：Client将结果传递给Host；
5. 第五步（可选）：如果有需要，继续重复2到4，调用不同的mcp完成不同的任务；
6. 第六步：Host将重新组装Prompt，传递给大模型；
7. 第七步：大模型完成任务之后,将结果反馈给Host；
8. 第八步：Host将结果展现给用户。



二、MCP基本使用

2.1 MCP主机和客户端（Host+Clent）

> **主机**是协调整个MCP系统的核心： - 创建和管理多个客户端实例 - 控制客户端的连接权限和生命周期 - 执行安全策略和权限管理 - 协调AI/LLM的集成和采样 实际上，**主机通常是你使用的AI应用程序**，如Claude Desktop、Cursor编辑器或其他支持MCP的应用。

在 MCP 官方文档中，我们看到已经支持了 **MCP** 协议的一些客户端/工具列表

Client	Resources	Prompts	Tools	Sampling	Roots	Notes
Claude Desktop App	✅	✅	✅	❌	❌	Full support for all MCP features
Claude Code	❌	✅	✅	❌	❌	Supports prompts and tools
5ire	❌	❌	✅	❌	❌	Supports tools.
BeeAI Framework	❌	❌	✅	❌	❌	Supports tools in agentic workflows.
Cline	✅	❌	✅	❌	❌	Supports tools and resources.
Continue	✅	✅	✅	❌	❌	Full support for all MCP features
Copilot-MCP	✅	❌	✅	❌	❌	Supports tools and resources.
Cursor	❌	❌	✅	❌	❌	Supports tools.
Emacs Mcp	❌	❌	✅	❌	❌	Supports tools in Emacs.
fast-agent	✅	✅	✅	✅	✅	Full multimodal MCP support with end-to-end tests

知乎 @华尔街的幻觉

从表格里，我们可以看到，MCP 对支持的客户端划分了五大能力，这里我们先简单了解即可：

图像中的文字如下：

- **Tools**：服务器暴露可执行功能，供 LLM 调用与外部系统交互。
- **Resources**：服务器暴露数据和内容，供客户端读取并作为 LLM 上下文。
- **Prompts**：服务器定义可复用的提示模板，引导 LLM 交互。
- **Sampling**：让服务器借助客户端向 LLM 发起完成请求，实现复杂的智能行为。
- **Roots**：客户端给服务器指定的一些地址，用来告诉服务器该关注哪些资源和去哪里找这些资源。

很显然，**Tools** 是最最最重要、最广泛的功能，其他功能可以没有，`Tools`功能必须有。 才能称之为“**支持MCP的客户端**”。

> [!NOTE]
>
> **客户端（Client）** 由主机创建，维护与一个服务器的连接： - 与一个特定服务器建立一对一的会话 - 处理协议协商和能力交换 - 在服务器和主机之间双向传递消息

2.2 MCP 服务器（Server）

> **服务器**提供专门的功能和数据： - 通过MCP原语公开资源、工具和提示模板 - 独立运行，专注于特定责任 - 可以是本地进程或远程服务 **这是你将要构建的部分，用于让AI访问你的特定数据或功能。**

常见的 MCP Server 有： - **文件和数据访问类**：让大模型能够操作、访问本地文件或数据库，如 File System MCP Server； - **Web 自动化类**：让大模型能够操作浏览器，如 Puppeteer MCP Server； - **三方工具集成类**：让大模型能够调用三方平台暴露的 API，如 高德地图 MCP Server；

三、Function Calling（函数调用）

允许大语言模型（LLM）通过自然语言指令与外部工具和服务进行交互，从而将自然语言转换为具体的 **API 调用**。这一技术解决了大语言模型在训练完成后知识更新停滞的问题，使大模型能够获取实时信息，比如：当前的天气、股市收盘点数等。

3.1工作原理

Function Calling 的工作原理可以通过以下4个步骤来理解：

1、**识别需求**：大模型识别出用户的问题需要调用外部 API 来获取实时信息。比如：用户询问“今天北京的天气如何？”大模型会识别出这是一个关于实时天气的问题。

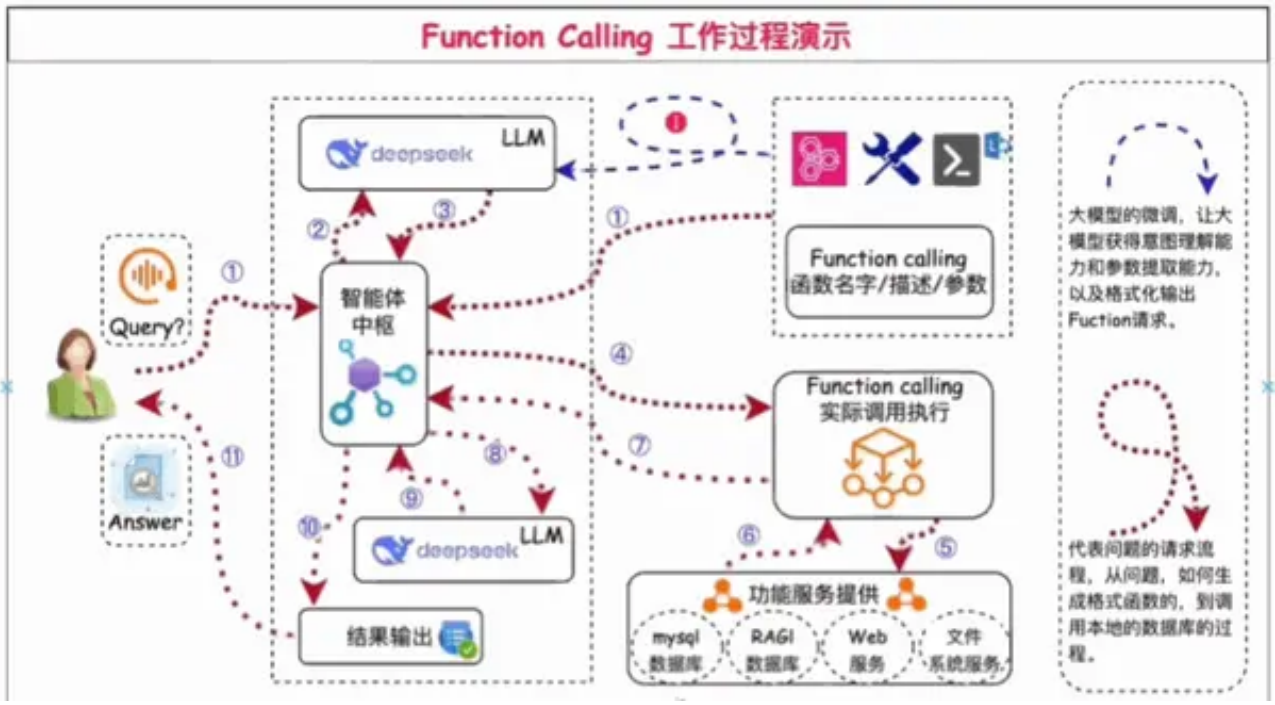
2、**选择函数**：大模型从可用的函数库中选择合适的函数。在这个例子中，大模型会选择 get_current_weather 函数。

3、**准备参数**：大模型准备调用函数所需的参数。例如：

```
{
    "location": "北京",
    "unit": "celsius"
}
```

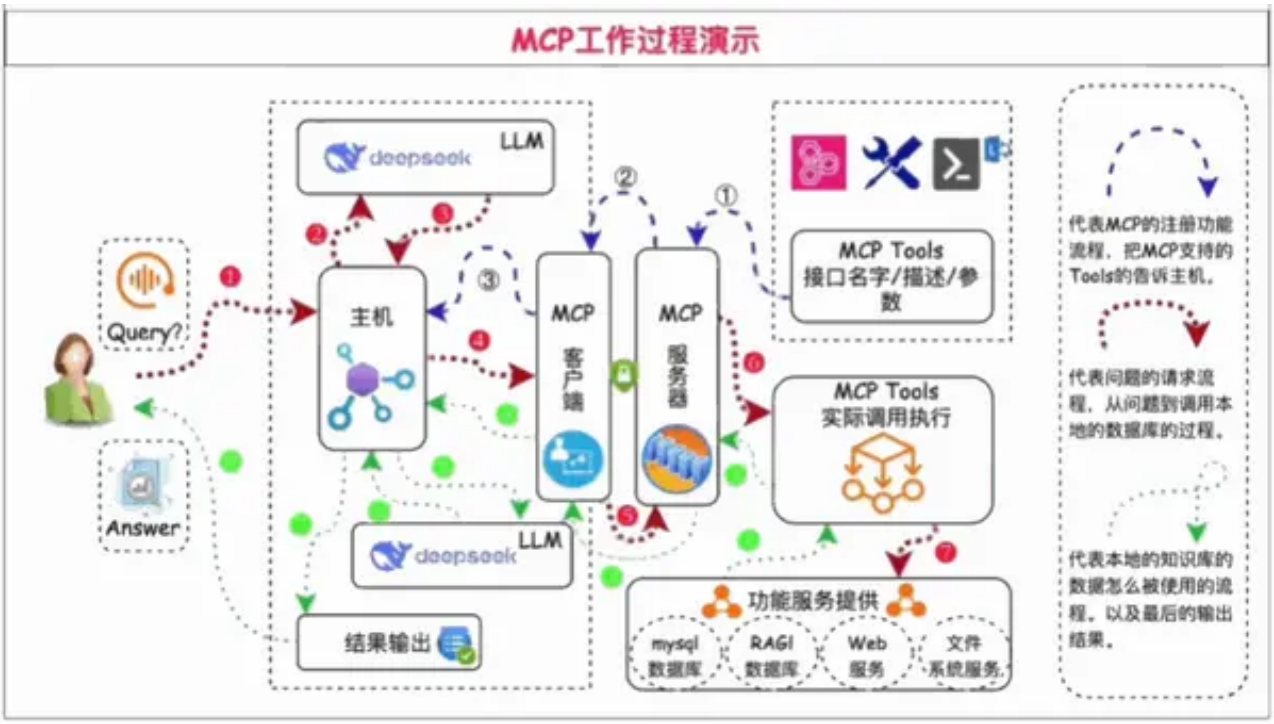
3、**调用函数**：AI 应用使用这些参数调用实际的天气 API，获取北京的实时天气数据。

4、**整合回答**：大模型将获取的数据整合成一个完整的回答，比如：“根据最新数据，北京今天的天气晴朗，当前温度23°C，湿度45%，微风。今天的最高温度预计为26°C，最低温度为18°C。”



Function Calling工作过程演示

1. **用户输入问题**： 用户通过界面或语音输入问题（Query）。
2. **LLM处理**： 问题被发送到DeepSeek LLM进行分析和理解。
3. **智能体中枢**： LLM将问题传递给智能体中枢，中枢解析问题意图并生成函数调用请求。
4. **Function Calling注册**： 大模型通过微调获得意图理解和参数提取能力，生成格式化的Function请求。
5. **工具执行**： Function Calling实际调用本地数据库或其他服务（如MySQL、RAGI、Web服务等）获取数据。
6. **结果返回**： 处理后的结果返回给LLM，并最终输出答案。



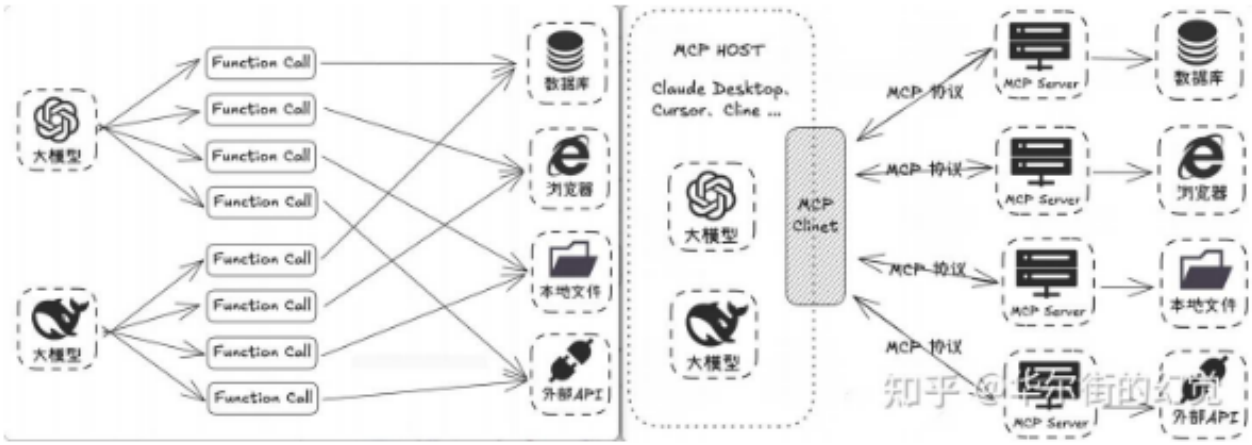
MCP工作过程演示：

1. **用户输入问题**： 用户通过界面或语音输入问题（Query）。
2. **LLM处理**： 问题被发送到DeepSeek LLM进行初步处理。
3. **MCP注册流程**： MCP工具的接口信息（名称、描述、参数）被注册并告知主机。
4. **客户端与服务器交互**： 客户端将问题传递给MCP服务器，服务器根据问题调用相应的工具。
5. **工具执行**： MCP Tools实际调用本地数据库或其他服务（如MySQL、RAGI、Web服务等）获取数据。
6. **结果返回**： 处理后的结果返回给LLM，并最终输出答案。

MCP相较于**Function Calling**，核心区别在于**引入了客户端-服务器架构**：

1. **客户端发起请求**： 用户问题经LLM处理后，由客户端传递至MCP服务器；
2. **服务器协调工具**： MCP服务器统一调用本地工具（如数据库、Web服务），并返回结果；
3. **工具注册机制**： MCP需预先注册工具接口信息至主机，Function Calling则直接通过智能体解析意图生成函数调用，无需显式注册。

> 这一架构使MCP具备更清晰的职责划分与扩展性，而Function Calling侧重大模型自主解析与执行。



（左图Function Call，右图MCP）

> [!NOTE]
>
> **Function Calling**
>
> **核心特点：**
>
> - **模型专属**：不同模型(GPT/Claude/DeepSeek)的调用规则不同
> - **即时触发**：模型解析用户意图后直接调用工具
> - **简单直接**：适合单一功能调用(如"查北京温度"->调用天气API)
>
> **痛点：**
>
> - **协议碎片化**：需为每个模型单独开发适配层
> - **功能扩展难**：新增工具需重新训练模型或调整接口
>
> **类比**：不同品牌手机的充电接口(Lightning/USB-C)，设备间无法通用
>
> ---
>
> **MCP**
>
> **核心特点：**
>
> - **协议标准化**：统一工具调用格式(请求/响应/错误处理)
> - **生态兼容性**：一次开发即可对接所有兼容MCP的模型
> - **动态扩展**：新增工具无需修改模型代码，即插即用
>
> **核心价值，解决三大问题**
>
> - **数据孤岛** →打通本地/云端数据源
> - **重复开发**→工具开发者只需适配MCP协议
> - **生态割裂**→形成统一工具市场

3.2对比小结

> **MCP 不是 Function Calling 的替代，而是基于 Function Calling 的工具箱。**

很多人误认为，MCP 是对传统 Function Calling 的一种替代。

而实际上，两者并非替代关系，而是紧密合作的关系。

Function Calling 是大语言模型（LLM）与外部工具或 API 交互的核心机制。它是大模型的一个基础能力，就是识别什么时候要工具，可能需要啥类型的工具的能力。

而 MCP 则是工具分类的箱子。因此 MCP 不是要取代 Function Calling，而是在 Function Calling 基础上，联合 Agent 一起去完成复杂任务。

如果把整个工具调用的流程剖析开来，实际是"Function Calling+ Agent + MCP 系统"的组合。

最后用一句话总结：大模型通过 FunctionCalling 表达，我要调用什么工具，Agent 遵循指令执行工具的调用，而 MCP 则是提供了一种统一的工具调用规范。 MCP 并不负责决定使用哪个工具，也不进行任务规划或理解用户意图。这些是 Agent 层面的工作。MCP 只是提供了一个统一的工具接口，成为了产业内认可的工具调用标准协议。

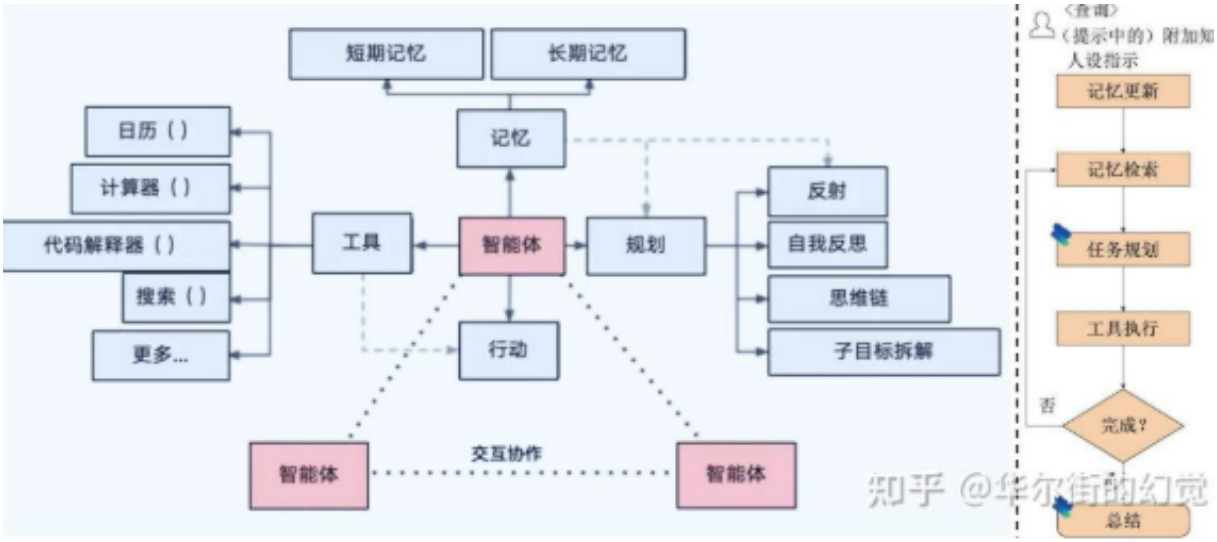
3.3 MCP和Agent的区别

流行最广的是前 OpenAI 研究与安全副总裁 Lilian Weng 对 Agent 的**定义**：

Agent = LLM + Planning + Tools + Memory。

完整定义：Agent（智能体）是具备自主行动能力的**系统**，包含规划(Planning)、记忆(Memory)、工具(Tools)、执行(Action)四大要素（或称组件），能够执行一系列复杂的任务。

- LLM 充当代理的大脑 具有**规划**能力：如思维链(CoT)、任务拆解、反思和改进
- 具有**记忆**：短期和长期记忆
- 具备**工具**使用能力：通过API、FunctionCall、**MCP**协议等渠道**获取**外部信息，甚至进行**交互**。



Agent的推理流程：Agent规划并执行子任务，利用工具来增强功能，并通过反思行为来学习和改进。通过多个模块的紧密协作，Agent能够有效地分解复杂任务，进行规划和执行，并最终生成解决方案。这种结构化和分步的方法使得复杂问题的处理变得更加清晰和可管理。

从 Agent定义 中就能知道，Agent是一个系统，MCP是Agent调用工具能力的其中一种方式。

3.4 小结

MCP、Function Calling、Agent 区别

从表面上看，这三者似乎功能相似，目的也相近，都是为了弥补大模型的短板。

第一、MCP： MCP（Model Context Protocol）是一种开放协议，旨在提供一个通用的开放标准，用于连接大语言模型（LLM）和外部数据及行为。MCP 运行在本地（至少目前是），由像 Cursor 这样的主机（或统称为终端）调用，可以访问本地资源、个性化的 API 等。

第二、Function Calling： Function Calling 是 AI 模型与外部函数或服务交互的一种机制。在这种模式下，模型生成一个函数调用请求，宿主应用解析该请求并执行相应的操作，然后将结果返回给模型。

Function Calling 通常具有以下特点：

- **同步执行**：调用函数后，程序会等待函数执行完毕并返回结果，才继续执行后续代码。
- **紧耦合**：模型与函数或服务之间的关系较为紧密，需要在代码中明确指定。
- **特定实现**：函数调用的实现方式可能因平台或服务提供商而异，缺乏统一标准。

第三、Agent： Agent（智能体）是具备自主行动能力的**系统**，能够执行一系列复杂的任务，比如曾经一码难求Manus和flowith。

Agent 通常具备以下特征：

- **自主性**：能够根据环境变化和目标自主做出决策。
- **任务执行**：能够执行多步骤、多环节的任务，往往需要调用多个工具或服务。
- **集成性**：通常集成多种功能模块，如 **MCP** 和 **Function Calling**，以实现复杂的任务处理。

第四、主要区别

MCP：作为一种**协议**，主要解决模型与外部工具和数据源之间的交互问题，提供标准化的接口和通信方式。它非常灵活，只要遵循标准，几乎能实现任何功能。

Function Calling：是模型与特定函数或服务交互的具体**实现方式**，关注如何在代码层面实现功能调用。它需要大模型和特定客户端紧密绑定，灵活性较低。

Agent：是一个复杂的**系统**，能够自主执行任务，通常需要**结合 MCP** 和 **Function Calling** 等机制，以实现复杂的功能。

> [!NOTE]
>
> **Function Calling**：
>
> - 发生在 **模型推理内部**。
> - 由模型“决定”调用哪个函数，宿主程序执行后把结果“喂回”模型。
> - 通常是一次性、单步的（虽然可循环，但逻辑在宿主控制）。
>
> **MCP**：
>
> - 发生在 **应用层与工具层之间**。
> - Host（如 Cursor）通过 MCP Client 与 MCP Server 通信，**不依赖模型本身实现调用逻辑**。
> - 支持更丰富的交互类型：工具调用（Tools）、上下文资源（Resources）、提示模板（Prompts）等。
>
> **Agent**：
>
> - 是 **顶层架构**，可能内部使用 Function Calling 或 MCP 作为其“手脚”。
> - 能自主规划任务、拆解子目标、循环调用多个工具、处理错误、记忆状态等。