**Introduction of MCP:**

MCP is an open protocol that standardizes how applications provide context to LLMs. Think of MCP like a USB-C port for AI applications. Just as USB-C provides a standardized way to connect your devices to various peripherals and accessories, MCP provides a standardized way to connect AI models to different data sources and tools.

* **Why MCP:**

Modern applications increasingly rely on **Large Language Models (LLMs)** that must dynamically interact with **multiple tools, databases, APIs, and files**. Traditional RESTful APIs struggle with this because they:

* Require manual integration and boilerplate per tool.
* Lack context awareness and persistent state.
* Don’t support dynamic tool discovery or bidirectional communication.

**Model Context Protocol (MCP)** solves this by standardizing how LLM-based apps **discover**, **invoke**, and **interact** with tools in a structured, contextual, and modular way.

**It enables:**

* **Dynamic, pluggable tools** with self-describing schemas.
* **Context sharing** (memory/state) across tools.
* **Asynchronous interactions** like notifications or background tasks.
* **Reduced N×M complexity** for tool integrations.
* **Learn Few Terminologies:**

Learn few terminologies like: **MCP Host**, **MCP Client**, **MCP Server**, **MCP Protocol**, etc.

1. **MCP Host:**

The **MCP Host** is the LLM-powered application or environment (e.g., Claude Desktop, a chat UI, or IDE) that wants to access external information or tools—but cannot talk to them directly.

* Acts as the “**chef**” who needs ingredients (data), but doesn’t fetch them.
* Example: Claude Desktop wanting to query a Postgres database.
* Hosts orchestrate the conversation and manage multiple tool interactions.
* Uses **MCP Clients** internally to connect to tools.

1. **MCP Client:**

The **MCP Client** is the connector within a Host that translates LLM requests into MCP messages and sends them to servers.

* Like the “waiter” taking orders from the chef.
* Example: MCP client inside an IDE that calls GitHub or file-system MCP servers.
* Maintains one-to-one connections to each MCP Server.
* Handles JSON-RPC communication (requests, responses, notifications).

1. **MCP Server:**

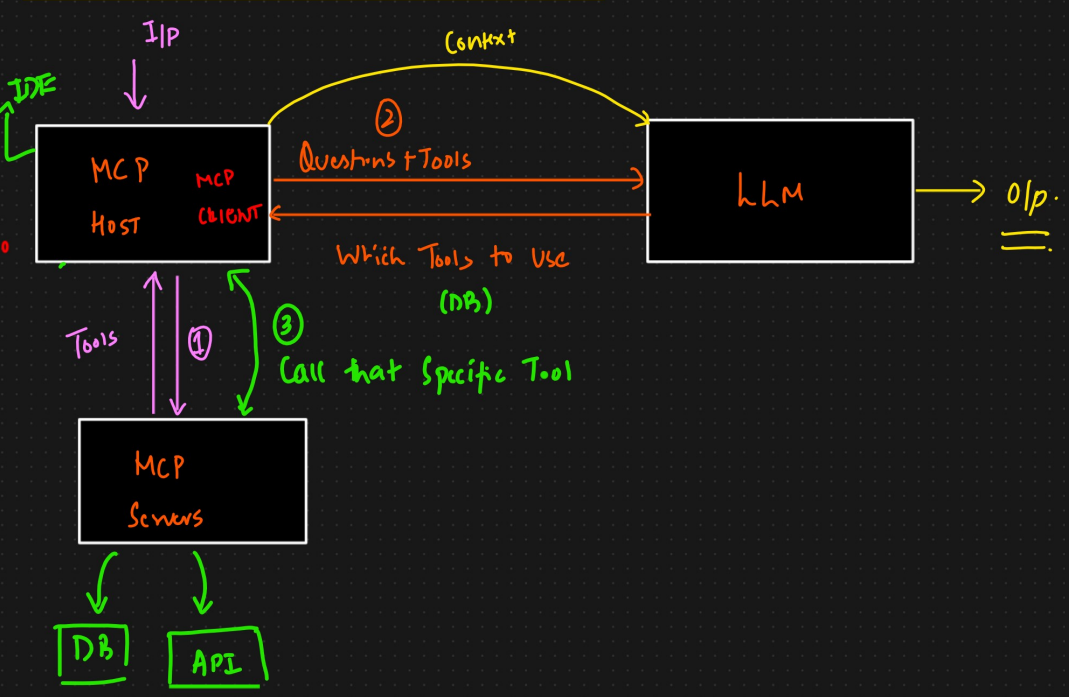
The **MCP Server** is a lightweight service exposing specific capabilities—such as databases, files systems, and web APIs— via MCP.

* Acts like the “supplier” who delivers ingredients/tools.
* Example: PostgreSQL MCP Server that runs SQL queries and returns results.
* Servers are **self-describing**, sharing schema, tools, annotations, and capabilities.
* Can wrap local resources (e.g., files) or remote services (e.g., Slack, Stripe).

1. **MCP Protocol:**

The **Model Context Protocol** standardizes how Clients and Servers communicate via **JSON-RPC 2.0** over **stdio or HTTP+SSE**, enabling dynamic tool use and context exchange.

* Defines messages: requests (tool calls), notifications, and responses.
* Supports tool discovery, structured inputs/outputs, and **persistent context.**
* Enables bidirectional communications: Servers can notify Clients or request model samples.
* Example types:
  + **Resources**: read-only context like database views.
  + **Tools**: actions like “run\_sql” or “create\_chart.”
  + **Sampling**: server asks model to generate summary of data.
* **Communication between Components:**



* **Real World Example:**

**Scenario: K12 Chatbot with Agentic Capabilities**

You're building an AI-powered chatbot that helps students with:

* Answering subject-specific questions (Math, Science, etc.)
* Generating MCQs or quizzes.
* Fetching summaries from textbooks.
* Displaying charts or solving equations.

**Key Components in these Applications:**

1. **MCP Host (K12 Chatbot Application):**

* This is your main chatbot interface (e.g., built with **LangChain** or **LangGraph**).
* It includes the **LLM agent** that interacts with students.
* It needs access to textbooks, quiz generation tools, image/video explainers, etc., but doesn’t connect directly to them.
* **Example**: A LangChain app with memory and state management

1. **MCP Client (Connector Layer):**

* Resides within the host (chatbot) and connects to different MCP Servers.
* Translates AI’s internal requests into structured MCP tool calls.
* Maintains **persistent state and tool discovery.**

1. **MCP Servers (Your Tool Providers):**

You can create various servers depending on the tools you want your chatbot to access:

* **Textbook Reader Server:** Exposes subject-specific PDFs or markdown notes as a resource.
* **Quiz Generator Server:** A tool that takes a topic and returns 5 MCQs.
* **Math Solver Server:** Accepts math expressions, returns LaTeX-rendered solution or graph.
* **Video/Animation Server:** Calls external APIs to return animated video explanations.

Each of these servers self-describes its tools (e.g., generate\_quiz, read\_topic, solve\_equation).

1. **MCP Protocol:**

* The standard that **connects your client and servers**.
* Uses **JSON-RPC 2.0** over:
  + **HTTP + Server-Sent Events (SSE)** for web-hosted tools.
  + **stdio** for local resources or offline apps
* This enables your chatbot to:
  + Discover which tools are available.
  + Understand each tool’s input/output schema.
  + Maintain session context (like memory of what topic is being discussed).
  + Receive **notifications** (e.g., “quiz ready!”).

**Why MCP is perfect for this:**

* **Dynamic tool discovery**: Add new subjects/tools without coding changes in the chatbot.
* **Persistent state**: Student topic context and progress retained across multiple questions.
* **Security and isolation**: Each tool runs in its own **MCP** **Server** (e.g., quiz generator can't access DB directly).
* **Scalable & modular**: You can scale servers independently (e.g., separate quiz service per grade).

**Note:**

If you have **50 different tools**, you do **not necessarily need 50 servers**, but you can, depending on your architecture and design goals. Here’s a breakdown based on **Security**, **Isolation**, and **Scalability** using **MCP:**

1. **Security & Isolation:**

**Why separation helps:**

* Each **MCP Server** runs in its **own process or container**.
* If a tool is compromised, the damage is isolated.
* Sensitive tools (e.g., **student\_db\_tool**) can run in a **secure environment** with access control.

**Example:**

* **Quiz\_generator** cannot access **student\_grades\_tool** or **admin\_tools** unless explicitly allowed.
* You can sandbox tools that run untrusted code (e.g., **code\_executor\_tool**).

**Conclusion**: Creating separate servers per sensitive or critical tool enhance **security and isolation**.

1. **Scalability & Modularity:**

**Why modular servers help:**

* Tools with **heavy load** (e.g., **video\_generator**, **text\_summarizer**) can scale independently.
* You can deploy tool servers on different machines or even in the cloud (e.g., AWS Lambda, containers).
* Developers can work on tools independently and deploy updates without affecting the rest of the system.

**Example:**

* **math\_solver**, **textbook\_parser**, and **speech\_to\_text\_tool** are hosted separately and scaled based on demand.

**Conclusion:** Splitting tools into modular MCP servers allows **horizontal scaling** and better resource management.

* **Comparisons between APIs vs. MCP:**

**Similarities between APIs vs. MCP:**

|  |  |
| --- | --- |
| **Aspect** | **Descriptions** |
| **Communication** | Both are mechanisms for communication between clients and services/tools. |
| **Data Exchange** | Both use structured data (usually JSON) to send and receive information. |
| **Request/Response** | Both follow a pattern where a client sends a request and gets a response. |
| **Interoperability** | Both allow integration of multiple services and tools, often over HTTP. |
| **Can be Secured** | Both support secure communication (e.g., via HTTPS or internal auth). |

**Difference between APIs and MCP:**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Traditional APIs** | **Model Context Protocol (MCP)** |
| **Designed For** | Human/frontend clients (browsers, apps). | LLMs and autonomous agents. |
| **Interface Type** | RESTful (HTTP verbs, endpoints like /**getData**). | JSON-RPC method calls like **run\_tool**, **generate\_quiz**. |
| **Tool Discovery** | Manual (**Swagger**/**OpenAPI**) or hardcoded. | Automatic, self-describing tools (metadata sent by each MCP server). |
| **Context Awareness** | Stateless — no memory between requests. | Contextual — LLMs retain memory across multiple tool calls. |
| **Protocol Structure** | URI-based, with route handlers (/**user**/**5**). | RPC-like calls with tool names as methods and parameter dicts. |
| **Dynamic Tool Calling** | Not native, requires orchestration logic. | Built-in — LLMs can reason and call new tools in real-time. |
| **Latency Handling** | Synchronous by default. | Supports **async** patterns (like server push, background jobs via notify, etc.) |
| **Scalability Model** | One endpoint per functionality or microservice. | One MCP Server per tool or group of tools — more modular, LLM-aware. |
| **Use Case Fit** | CRUD apps, frontend/backend separation. | LLM agents, tool chaining, AI orchestration (e.g., agents solving tasks using many tools). |