Model Context Protocol (MCP)

What Is a `Protocol` ?

> **A protocol is a set of rules that defines how two systems talk to each other .**

It decides:

- `What format the data should be in`
  → So both sides speak the *same language*
- `How requests are sent`
  → So messages reach the right place, the right way
- `How responses are returned`

  → So answers come back clear, expected, and usable

Protocol = rules for communication
MCP = rules for giving context to AI

`Context` Is the Key

The `core abilities of a Gen AI model` come from its pretraining —> `[ALT]` -> its architecture, training data, and learning process.

To make a pretrained model produce more relevant, coherent, and task-specific outputs, you must provide clear and well-structured context.

> **Context refers to the information the model uses to generate relevant and clear responses** .

Good context helps the model:

- Understand what you want
- Stay focused on the task
- Give more useful and accurate answers

In short, **better context → better results**.

`Ways Context Is Provided to AI Models`

1. Text-Based Models (GPT, LLaMA, DeepSeek)

Context comes from:

- **Prompt** – the input text that guides the response
- **Token Window** – how much information the model can remember at once (e.g., GPT-4-Turbo can handle ~128K tokens).
- **Conversation History** – previous messages in a chat
- **RAG** – external documents added dynamically for better answers

2. Image & Multimodal Models (DALL·E, Gemini)

Context comes from:

- **Text Descriptions** – prompts that guide generation
- **Visual Input** – images provided to the model
- **Cross-Modal Context** – combined text + image understanding

3. Code Generation Models (Codex, DeepSeek-Coder)

Context comes from:

- **Existing Code** – previous code, functions, and comments
- **Language Rules** – programming syntax and patterns
- **Documentation** – APIs or reference material

4. Speech & Audio Models (Whisper, AudioPaLM)

Context comes from:

- **Audio History** – previous speech or sound
- **Audio Features** – tone, speed, and intonation

MCP

`Model Context Protocol (MCP)` is an open-source protocol that defines how AI applications `provide context to Large Language Models (LLMs).`

Purpose

MCP standardizes the connection between AI models and external systems, enabling seamless access to:

- `Data sources`
- `Tools`
- `Workflows`

What MCP Enables

Using MCP, AI applications (such as ChatGPT or Claude) can connect to:

- **Data Sources**
  - Local files
  - Databases
  - External APIs
- **Tools**
  - Search engines
  - Calculators
  - Custom utilities

- **Workflows**
    - Specialized prompts
    - Automated task flows
    - Contextual pipelines

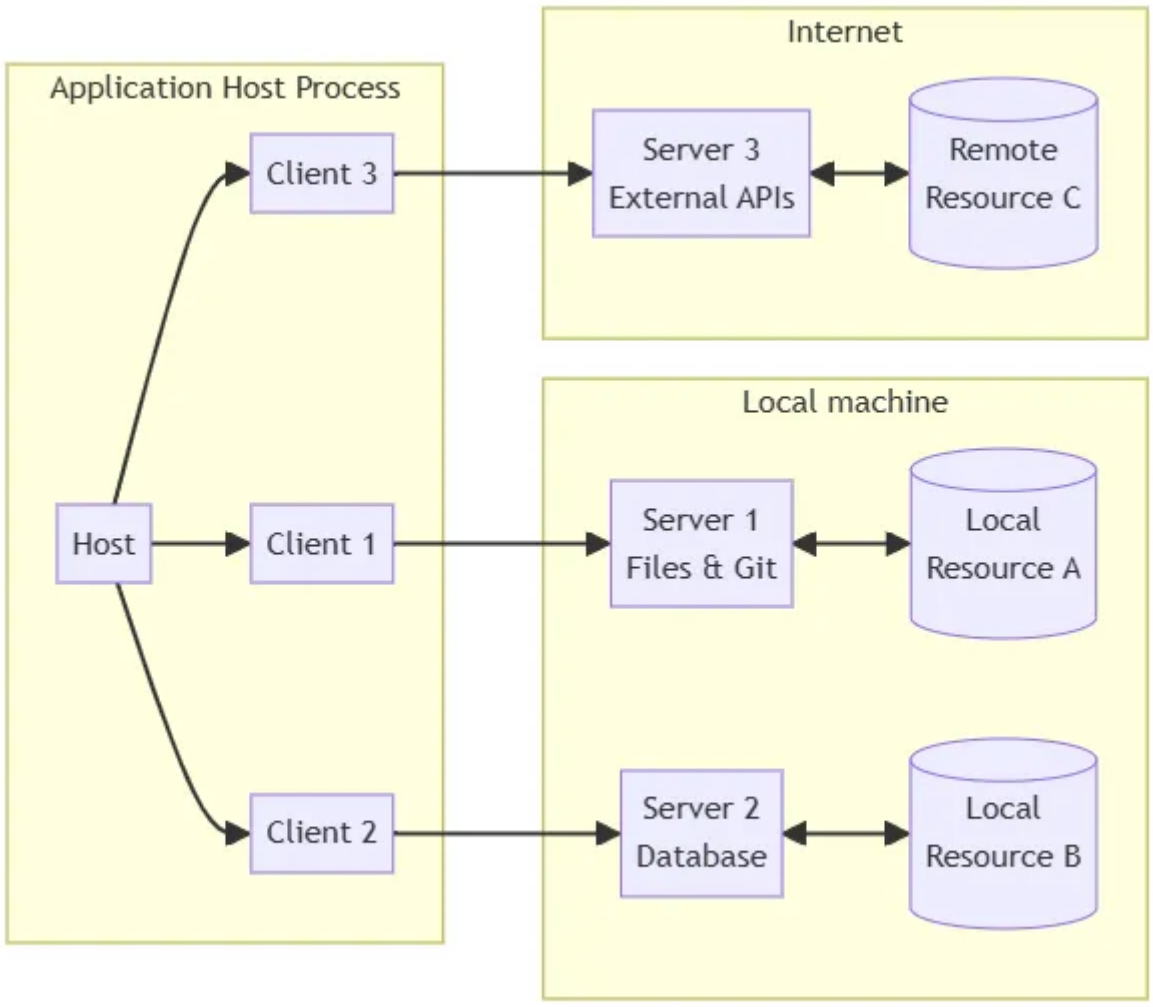
Simple Analogy

MCP is like a USB-C port for AI applications.

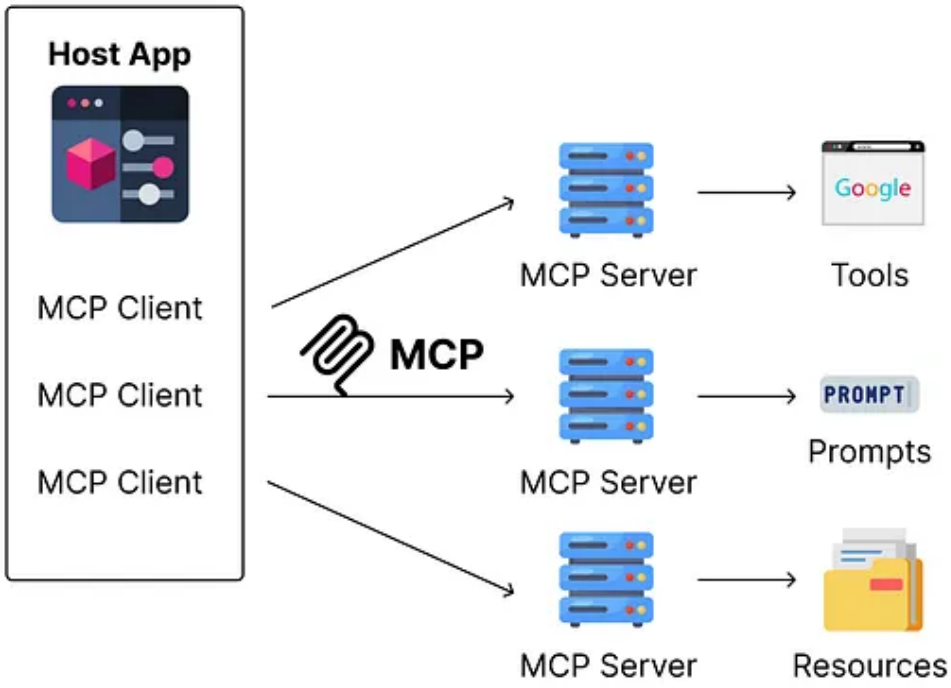
Just as USB-C provides a universal way to connect hardware devices, MCP provides a universal way to connect AI models to external systems and capabilities.

Architecture of MCP

MCP uses a `client-host-server architecture`, where:



- `Where each host can run multiple client instances`
- `Clients connect to servers to exchange context`



MCP operates on a client-server architecture, in which the clients live inside the host LLM applications and maintain 1-on-1 connections with servers, and the MCP servers act as middleman to external tools/data

This design:

- Makes it easy to integrate AI across different applications
- Keeps **security boundaries** clear
- Separates responsibilities for better isolation

MCP is built on **JSON-RPC** and uses a **stateful session protocol** to:

- Share context efficiently
- Coordinate requests and responses between clients and servers

Host

The **host** acts as the main controller and coordinator.

Responsibilities:

- Main controller of the system
- Creates and manages multiple clients
- Handles security, permissions, and user consent
- Coordinates AI/LLM usage
- Collects and manages context from all clients

Clients

Clients are created and managed by the host.
Each client connects to **one specific server (1:1)** and stays isolated from others.

Responsibilities:

- Establish a stateful session with a server
- Talks to the server using MCP rules
- Send and receive MCP messages
- Maintain security boundaries between servers
- Keeps servers isolated and secure

MCP clients are **AI applications or agents** that access external tools, data, or systems.

Key characteristics:

- MCP-compatible (supports prompts, tools, and resources)
- Can connect to any MCP server with minimal setup
- Invokes tools, queries resources, and applies prompts

Servers

Servers provide specialized capabilities and context. Responsibilities:

- Expose **resources, tools, and prompts**
- Focus on a specific task or system
- Execute tool requests and return results
- Respect security constraints
- Can be local or remote

`One-line summary:` Host manages → Clients connect → Servers provide capabilities

MCP Message Types

MCP uses **JSON-RPC 2.0** and defines three message types:

- **Requests** – Messages that ask for an action and expect a response
- **Responses** – Results or errors returned for a request
- **Notifications** – One-way messages with no response needed

All messages between MCP clients and servers **MUST** follow the JSON-RPC 2.0 specification. The protocol defines three fundamental types of messages:

Type	Description	Requirements
`Requests`	Messages sent to initiate an operation	Must include unique ID and method name
`Responses`	Messages sent in reply to requests	Must include same ID as request
`Notifications`	One-way messages with no reply	Must not include an ID

`JSON-RPC 2.0` in MCP

JSON-RPC 2.0 is a lightweight, stateless protocol that defines how two systems communicate using **JSON messages**. It is not tied to HTTP — it can work over **stdio, HTTP, WebSockets, SSE**, etc.

Message Structure

All JSON-RPC messages follow a standard structure:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "method_name",
  "params": {}
}
```

JSON-RPC Message Types in MCP

1. Request

Used when a response is required.

json Copy code

```
{
  "jsonrpc": "2.0",
  "id": 10,
  "method": "tools.call",
  "params": { "tool": "search" }
}
```

- Client or server sends it
- Receiver must send a response
- `id` is mandatory

2. Response

Returned for a request.

json Copy code

```
{
  "jsonrpc": "2.0",
  "id": 10,
  "result": { "status": "success" }
}
```

Or in case of error:

json Copy code

```
{
  "jsonrpc": "2.0",
  "id": 10,
  "error": {
    "code": -32601,
    "message": "Method not found"
  }
}
```

- `id` matches the request
- Exactly **one response per request**

---

3. Notification

Used when **no response is needed.**

json Copy code

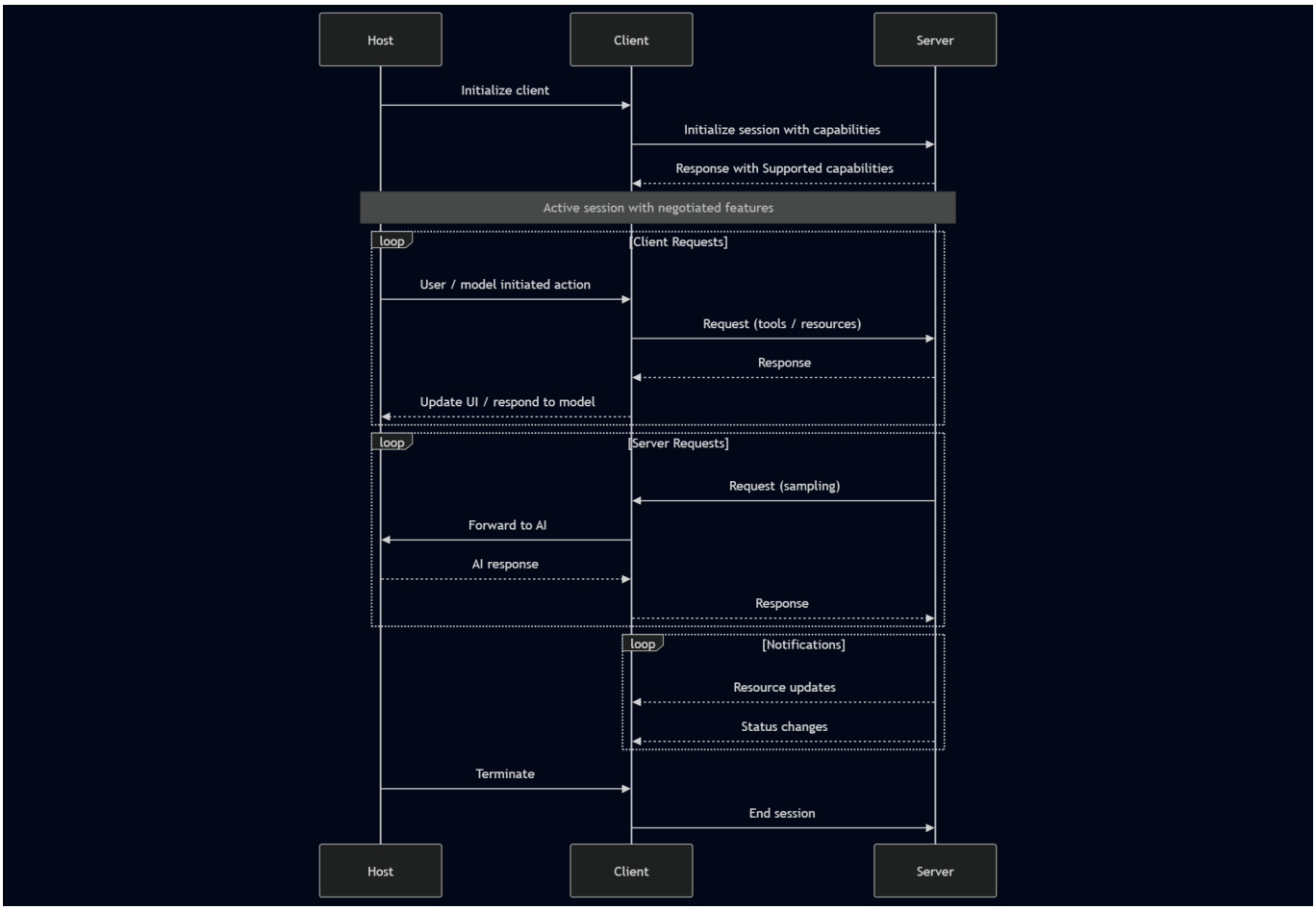
```
{
  "jsonrpc": "2.0",
  "method": "context.updated",
  "params": { "source": "files" }
}
```

- No `id`
- Fire-and-forget message

Capability Negotiation in MCP

> **MCP uses a capability-based system during initialization.**

- Clients and servers **declare what they support**
- Only shared capabilities are enabled in the session
- This decides which features, tools, and primitives can be used



Capability-Based Flow Explained (Sequence Diagram)

This sequence diagram shows how **MCP's capability-based system** controls all communication between the **Host**, **Client**, and **Server**.

---

1. Initialization = Capability Agreement

- The **Host** creates the **Client**
- The **Client** starts a session by sending its **supported capabilities**
- The **Server** responds with the capabilities it provides
- MCP enables **only the overlapping capabilities**

➡ Result: both sides agree *in advance* on what actions are allowed.

---

2. Active Session = Rules Are Locked

- Once capabilities are negotiated, the session becomes **active**
- No new features can be used unless negotiated again
- All communication must respect the agreed capabilities

➡ This ensures safety, predictability, and clear boundaries.

---

3. Client Requests (Tools & Resources)

- A user or model action triggers a request
- The **Client** calls only the **tools/resources declared by the Server**
- The **Server** processes the request and sends a response
- The **Host** updates the UI or responds to the model

➡ Client **cannot access undeclared tools**.

---

4. Server Requests (Sampling)

- The **Server** may request sampling from the model
- This happens **only if the Client declared sampling support**
- The **Client** forwards the request to the AI
- The response flows back through the Client to the Server

➡ The server never talks to the model directly.

---

5. Notifications (One-Way Updates)

- The **Server** sends updates (resource or status changes)
- Notifications are allowed **only if both sides agreed**
- No response is required

➡ Lightweight communication within capability limits.

---

6. Termination

- The **Host** ends the session
- The **Client** cleanly disconnects from the **Server**

➡ All interactions stop under defined rules.

Key Takeaways (Easy to Remember)

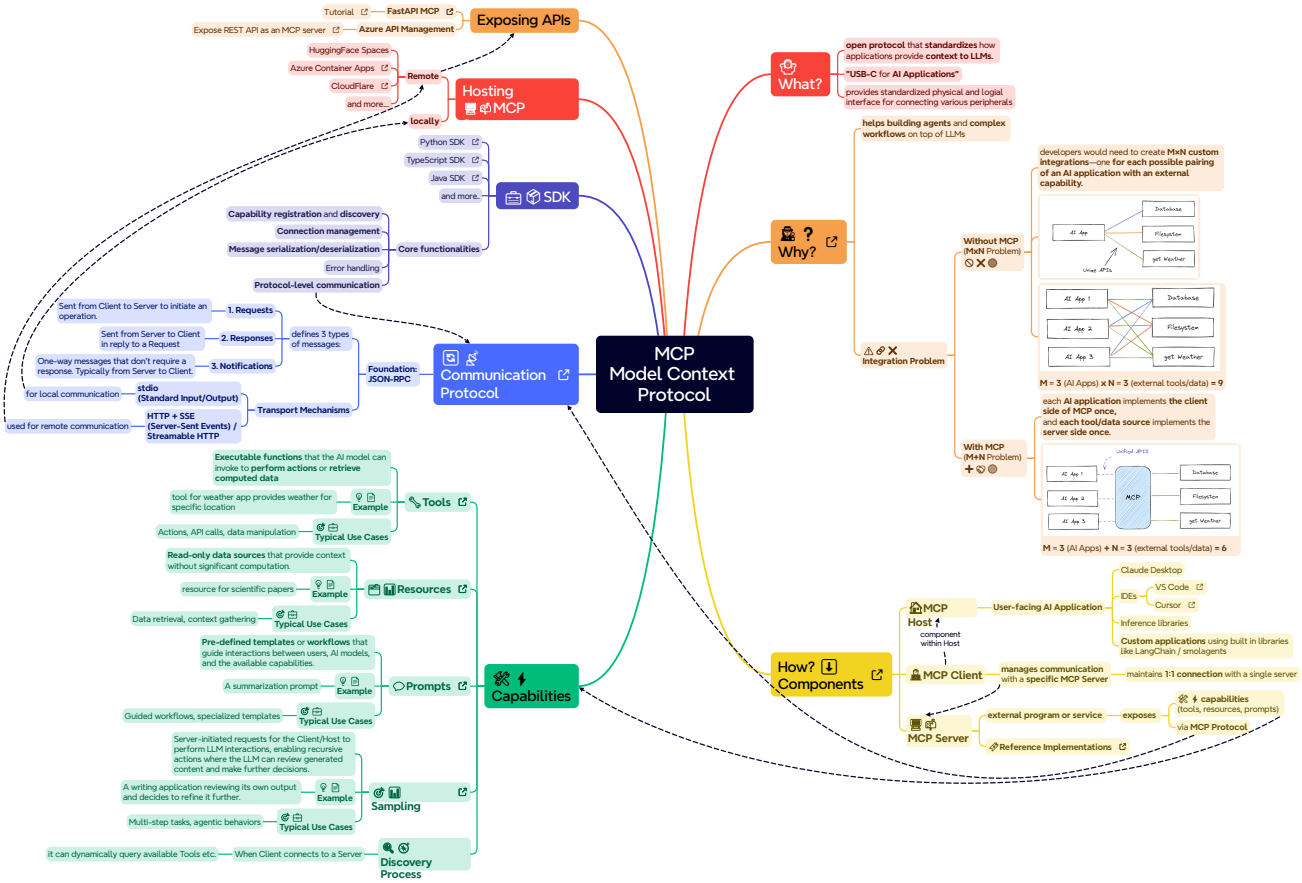
- **Handshake first** → declare capabilities
- **Only shared features work**
- **Clients call tools, servers execute them**
- **Servers request sampling via clients**
- **Notifications are optional and one-way**
- **Everything follows the agreed capabilities**

One-Line Mental Model

Agree → Lock rules → Communicate safely → End cleanly

MCP MIND MAP

Model Context Protocol Mind Map illustrates basic concepts :


MCP
Model Context
Protocol
What?
open protocol that standardizes how applications provide context to LLMs.
"USB-C for AI Applications"
provides standardized physical and logial interface for connecting various peripherals
Why?
helps building agents and complex workflows on top of LLMs
Integration Problem
Without MCP (M×N Problem)
developers would need to create M×N custom integrations—one for each possible pairing of an AI application with an external capability.
M = 3 (AI Apps) x N = 3 (external tools/data) = 9
With MCP (M+N Problem)
each AI application implements the client side of MCP once,
and each tool/data source implements the server side once.
M = 3 (AI Apps) + N = 3 (external tools/data) = 6
How? Components
MCP Host
component within Host
User-facing AI Application
Claude Desktop
IDEs
VS Code
Cursor
Inference libraries
Custom applications using built in libraries like LangChain / smolagents
MCP Client
manages communication with a specific MCP Server
maintains 1:1 connection with a single server
MCP Server
external program or service
exposes
capabilities (tools, resources, prompts)
via MCP Protocol
Reference Implementations
Capabilities
Tools
Executable functions that the AI model can invoke to perform actions or retrieve computed data
Example
tool for weather app provides weather for specific location
Typical Use Cases
Actions, API calls, data manipulation
Resources
Read-only data sources that provide context without significant computation.
Example
resource for scientific papers
Typical Use Cases
Data retrieval, context gathering
Prompts
Pre-defined templates or workflows that guide interactions between users, AI models, and the available capabilities.
Example
A summarization prompt
Typical Use Cases
Guided workflows, specialized templates
Sampling
Server-initiated requests for the Client/Host to perform LLM interactions, enabling recursive actions where the LLM can review generated content and make further decisions.
Example
A writing application reviewing its own output and decides to refine it further.
Typical Use Cases
Multi-step tasks, agentic behaviors
Discovery Process
When Client connects to a Server
it can dynamically query available Tools etc.
Communication Protocol
Foundation: JSON-RPC
defines 3 types of messages:
1. Requests
Sent from Client to Server to initiate an operation.
2. Responses
Sent from Server to Client in reply to a Request
3. Notifications
One-way messages that don't require a response. Typically from Server to Client.
Transport Mechanisms
for local communication
stdio (Standard Input/Output)
used for remote communication
HTTP + SSE (Server-Sent Events) / Streamable HTTP
Hosting MCP
Remote
HuggingFace Spaces
Azure Container Apps
CloudFlare
and more...
locally
Exposing APIs
Tutorial
FastAPI MCP
Expose REST API as an MCP server
Azure API Management
SDK
Python SDK
TypeScript SDK
Java SDK
and more...
Core functionalities
Capability registration and discovery
Connection management
Message serialization/deserialization
Error handling
Protocol-level communication