Introduction to Large Language Models

Spring 2026

Model Context Protocol (MCP)

(Some slides adapted from Ralph Grishman at NYU,
Yejin Choi at UWashington, N. Tomura at UDepaul, Jurafsky and Martin, CS224N,
CS224, CME295 at Stanford and other resourses on the web)

Model Context Protocol (MCP)

The **Model Context Protocol (MCP)** is an **open, vendor-neutral standard** that defines how **large language model (LLM) applications connect to external tools, data sources, and workflows** in a consistent and secure way. It was **introduced by Anthropic on November 25, 2024** and has since been adopted across the AI ecosystem. [anthropic.com], [en.wikipedia.org]

You'll often hear it described as **"USB-C for AI"**—one universal connector that replaces dozens of custom integrations. [modelconte…rotocol.io], [sureprompts.com]

Why MCP Exists

Before MCP, developers faced an **N×M integration problem**:

- Every AI model required **custom connectors** for each tool or data source
- Switching models (e.g., Claude ↔ GPT ↔ Gemini) meant rewriting integrations
- Security and governance logic was duplicated and inconsistent

MCP solves this by standardizing the integration layer **above** model-specific function calling, so tools are built **once** and reused everywhere. [en.wikipedia.org], [paloaltonetworks.com]

What MCP Enables

With MCP, an AI assistant can:

- Read **live data** from files, databases, APIs, and SaaS tools
- Execute **actions** (e.g., create Jira tickets, query Postgres, trigger CI/CD)
- Use **reusable prompts and workflows**
- Work inside IDEs, chat apps, and enterprise systems with consistent behavior

This turns static chatbots into **agent-like systems** that can both reason *and* act. [modelconte…rotocol.io], [cloud.google.com]

MCP Architecture (Conceptual)

MCP uses a **client–server model** built on **JSON-RPC 2.0**, inspired by the Language Server Protocol (LSP). [en.wikipedia.org], [modelconte...rotocol.io]

Core Components

1. **MCP Host**
   - The AI application (e.g., Claude Desktop, an IDE, Copilot-style agent)
2. **MCP Client**
   - Lives inside the host; negotiates capabilities and sends requests
3. **MCP Server**
   - Exposes tools, data, or workflows (local or remote)
4. **Transport**
   - stdio for local servers
   - HTTP + Server-Sent Events (SSE) for remote servers

The Three Core MCP Primitives

MCP intentionally limits itself to **three primitives**: [sureprompts.com], [modelconte...rotocol.io]

1. Tools

- Executable functions the model can invoke
- Example: create_issue, run_sql_query, commit_changes

2. Resources

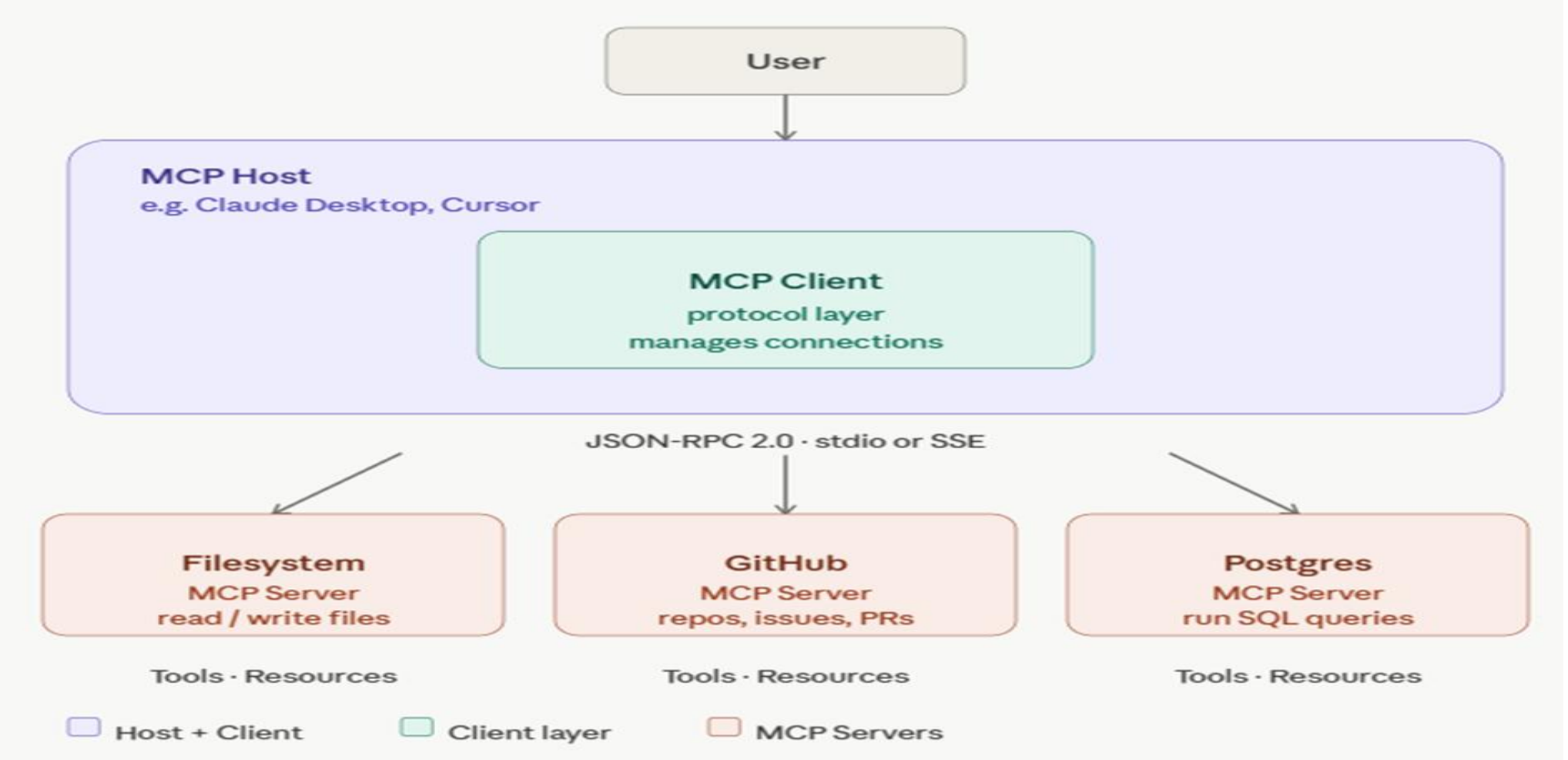
- Read-only contextual data
- Example: files, database rows, documentation, logs

3. Prompts

- Reusable instruction templates or flows
- Example: “Code review checklist”, “Incident response workflow”

This separation improves **control, safety, and clarity** in tool use.

MCP Architecture



The **User** talks to the **MCP Host** (like Claude Desktop), which contains an **MCP Client** as its internal protocol layer. The client speaks JSON-RPC 2.0 downward to one or more **MCP Servers** — each server is an independent program exposing its own tools and resources. The host can connect to many servers simultaneously; each gets its own dedicated client connection.

MCP Architecture

An **MCP Host** is the AI application that the user interacts with directly — it's the thing that *wants* to use external tools.

Examples: Claude.ai, Claude Desktop, Cursor, VS Code with Copilot. The host is responsible for:

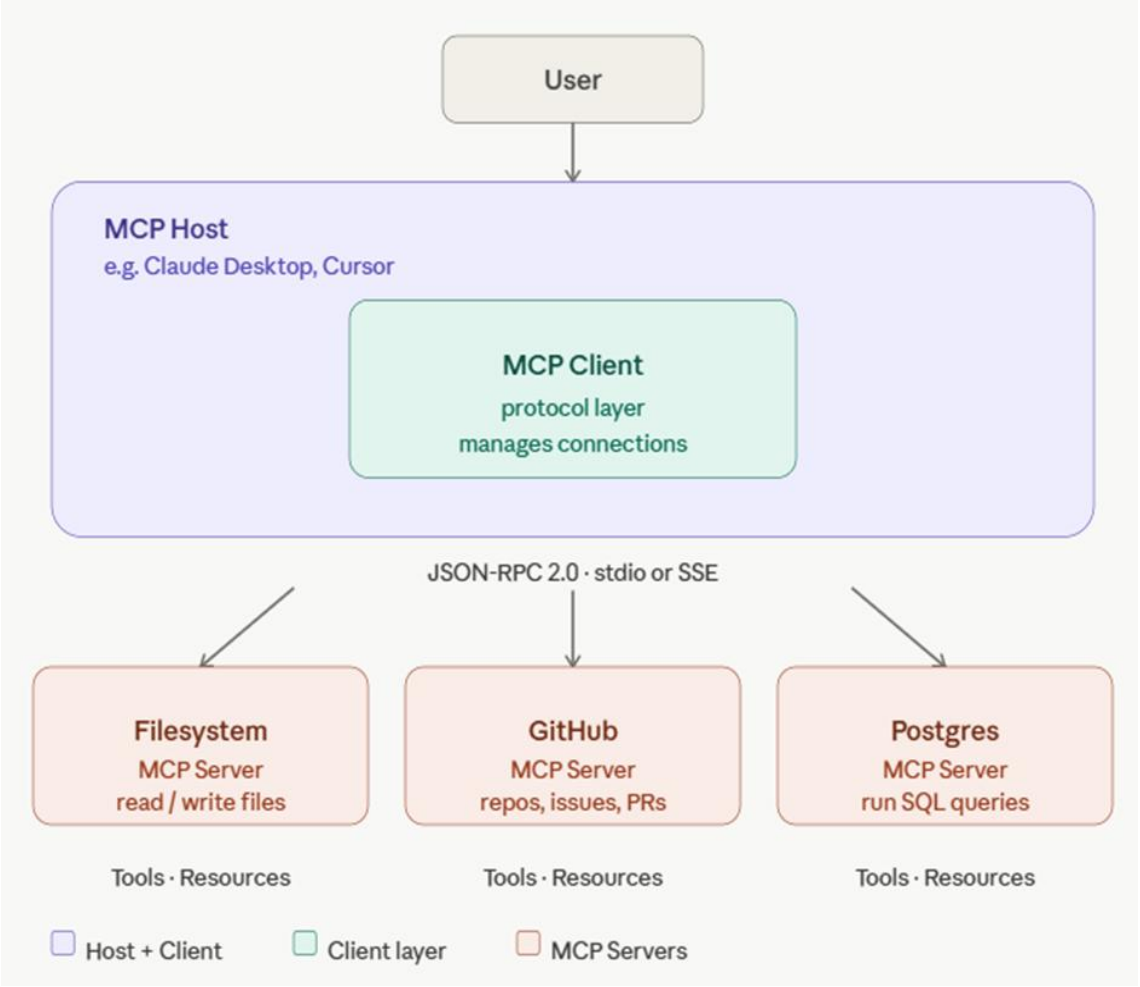
- **Managing connections** to one or more MCP servers
- **Showing tool results** to the user
- **Deciding which servers** to connect to based on configuration
- **Enforcing permissions** — the host controls what the model is allowed to do

In the architecture, the host sits at the top of the chain:

User → MCP Host (Claude) → MCP Client → MCP Server (GitHub, Slack...)

One subtle but important distinction: the **host** and the **client** are often part of the same application, but they're logically separate roles. The host is the user-facing app; the client is the protocol layer inside it that speaks JSON-RPC to servers. Claude Desktop, for example, acts as both host and client bundled together.

A single host can connect to **multiple MCP servers simultaneously** — so Claude could be talking to a GitHub server, a filesystem server, and a Postgres server all at once, with the host managing all those connections and routing tool calls to the right one.



MCP Architecture

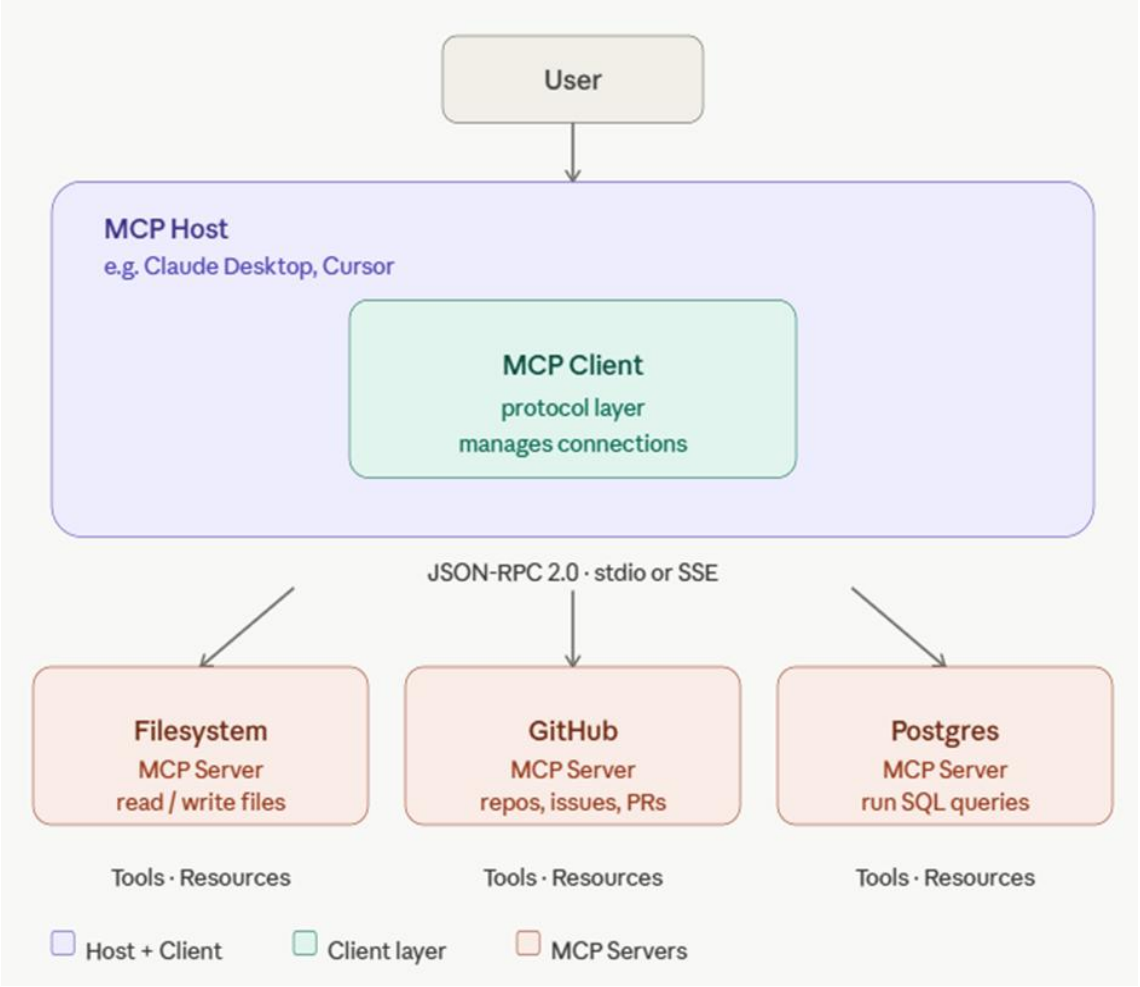
An **MCP Client** is the protocol layer that lives inside the host application and handles the actual communication with MCP servers.

It's the "translator" — it takes the model's intent and turns it into proper JSON-RPC messages that servers understand.

What the client does:

- Maintains a **1:1 connection** with each MCP server
- Sends the handshake (initialize) when a session starts
- Calls tools/list to discover what the server can do
- Sends tools/call when the model wants to execute a tool

Receives results and passes them back to the host



MCP Architecture

Where the Client lives:

The client is embedded inside the host app — it's not a separate process you run. When you use Claude Desktop, the MCP client is just a component running inside it. The split looks like this:

Claude Desktop

├— Host layer ← talks to the user, sends messages to the model

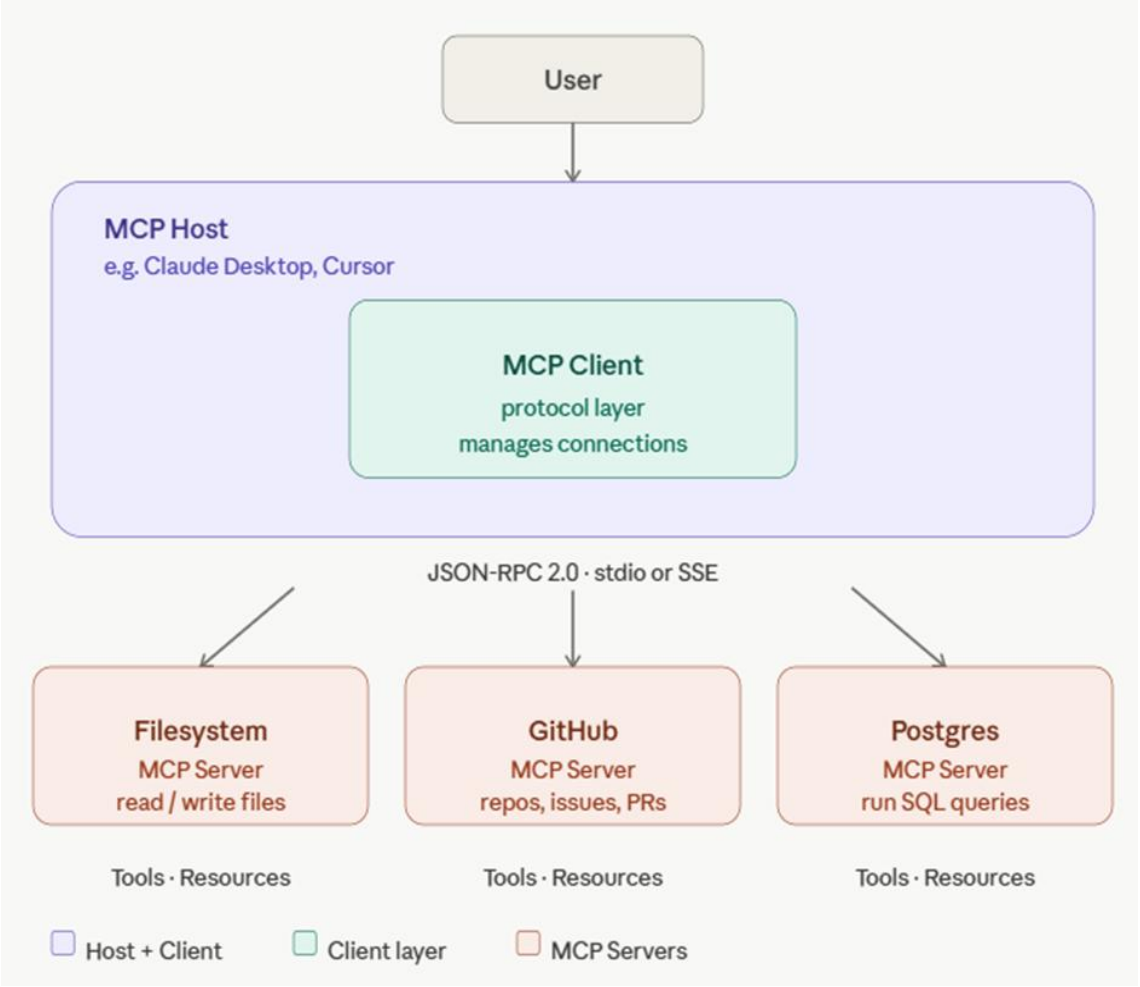
└— Client layer ← talks to MCP servers over stdio or SSE

One client per server: if your host connects to three MCP servers, it spins up three client instances — one dedicated connection each. They don't share state.

Transport options the client can use to reach a server:

- stdio — for local servers running as a subprocess (most common)
- SSE (Server-Sent Events) — for remote servers over HTTP

The client is intentionally thin — it doesn't make decisions about *when* to call tools, it just faithfully executes what the host and model ask it to. All the intelligence stays in the host/model layer above it.



MCP Architecture

An **MCP Server** is the program that exposes tools, resources, and prompts to the AI model. It's the bridge between Claude and the outside world — your files, databases, APIs, or any external service.

The server's job is simple: wait for requests from the client, execute them, return results.

MCP Server

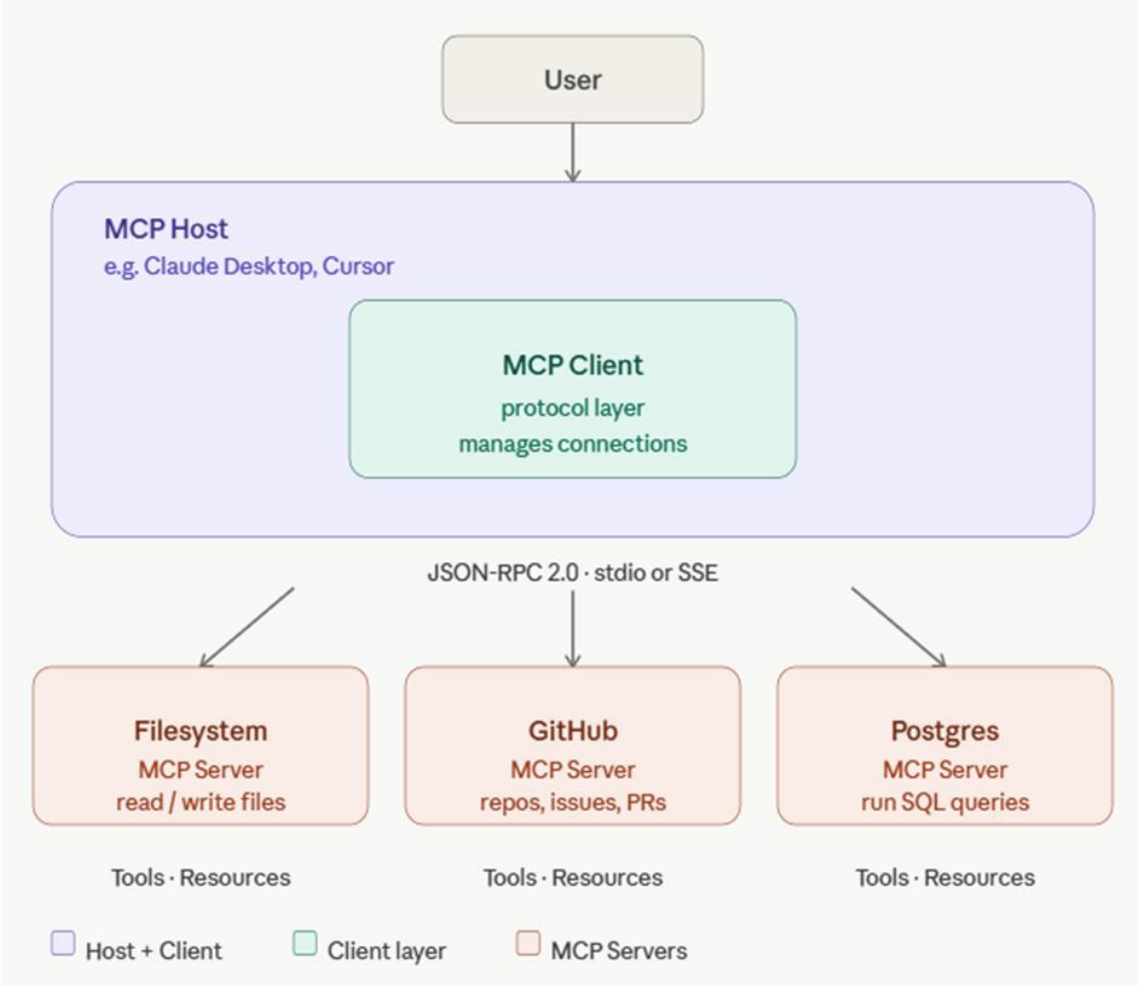
├— Exposes tools ← "here's what I can do"

├— Executes calls ← "ok, doing it now"

└— Returns results ← "here's what I got"

What a server can expose:

- **Tools** — callable functions (search_github, read_file, run_sql)
- **Resources** — readable data at a URI (file:///project/main.py, db://users/42)
- **Prompts** — reusable templates (summarize_pr, explain_error)

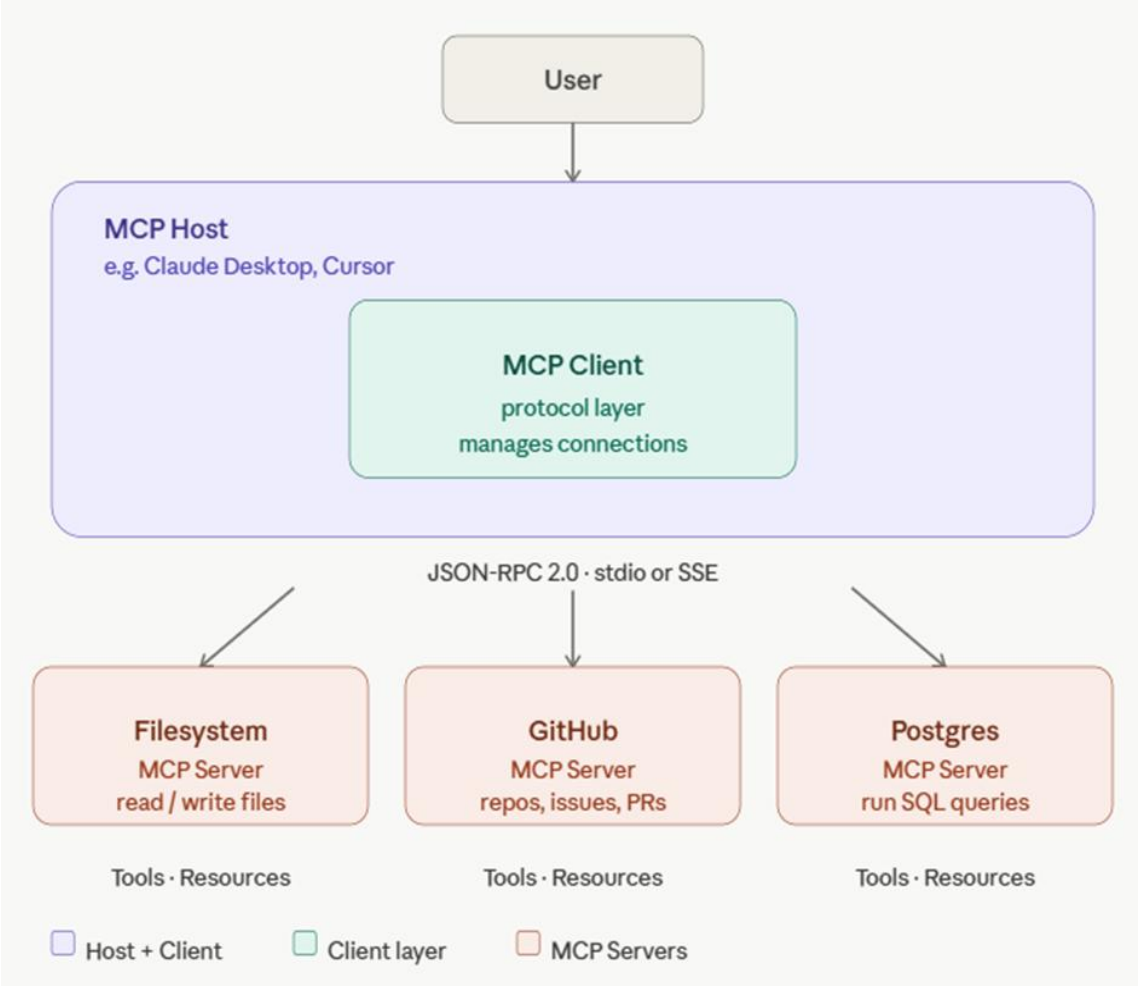


MCP Architecture

Examples of real MCP servers:

- filesystem — read/write local files
- github — search repos, open issues, read PRs
- postgres — run SQL queries against a database
- slack — post messages, read channels
- puppeteer — control a browser

The server is where you, as a developer, spend most of your time when building MCP integrations — the host and client are usually handled by existing frameworks like Claude Desktop or the MCP SDK.



Security Model (Important)

MCP itself does **not** grant permissions.

Security is enforced by the **host application**, which controls:

- Authentication (API keys, OAuth tokens)
- Tool allow-lists
- Data access scope
- User consent & audit logging

This design makes MCP suitable for **enterprise and regulated environments**. [paloaltonetworks.com], [modelconte…rotocol.io]

Comparison of MCP vs Function Calling

Below is a **clear, practical comparison of MCP (Model Context Protocol) vs Function Calling**, focusing on *what problem each solves*, *how they work*, and *when to use which*.

Concept	Core Problem Solved
Function Calling	"How can a model call this specific function with structured arguments?"
MCP (Model Context Protocol)	"How can a model discover, connect to, and use many tools/services in a standardized way?"

Definition (short)

Function Calling

A **model-level capability** where the LLM emits structured JSON arguments for a *predefined function* that the application executes.

Tight, local, app-controlled tool invocation.

MCP (Model Context Protocol)

A **tool/service protocol** that standardizes how models access tools, files, APIs, and services via *tool servers*.

Broad, networked, tool ecosystem integration.

Comparison of MCP vs Function Calling

Architectural difference:

Function Calling (inside your app)

User → LLM

→ "Call function X with args"

→ Your app executes function

→ Result sent back to LLM

Key properties

- Functions are **defined by the developer**
- Functions exist **inside the host application**

No discovery, no standard interface beyond JSON schema

MCP (tool ecosystem layer)

User → LLM

→ MCP Client

→ MCP Tool Server(s)

- filesystem
- databases
- CI tools
- cloud services

Key properties

- Tools are **external, discoverable services**
- Standardized protocol for:
  - Tool discovery
  - Invocation
  - Capability description
- Tool servers can be reused across apps and models

Comparison of MCP vs Function Calling

Aspect	Function Calling	MCP
Scope	Model-specific API	Open, model-agnostic protocol
Integrations	Per-model schemas	Build once, reuse everywhere
Tool discovery	Hard-coded	Dynamic capability negotiation
Enterprise fit	Limited	Strong governance support

Tool discovery

Key insight

Function calling assumes *you already know the tool*.

MCP allows the model to *find and understand tools at runtime*.

Feature	Function Calling	MCP
Tool discovery	❌ None	✅ Built-in
Dynamic tools	❌ Mostly static	✅ Yes
Tool reuse across apps	❌	✅
Namespace collisions avoided	❌	✅

Adoption and Ecosystem (2026)

MCP is now supported or integrated by:

- **Anthropic (Claude)**
- **OpenAI**
- **Google**
- Developer tools (VS Code, Visual Studio, Cursor, Replit)
- Enterprise platforms (e.g., Jama Connect MCP Server)

In **December 2025**, MCP was donated to the **Agentic AI Foundation under the Linux Foundation**, reinforcing its long-term neutrality. [en.wikipedia.org]

MCP Alternatives / Competing Standards

1. OpenAI Function Calling / Tools API

The closest conceptual competitor.

- Model emits structured tool calls
- Host application executes tools
- Results are fed back to the model

Main difference:

- MCP standardizes the transport + discovery layer
- OpenAI tools are primarily API-schema driven inside a model runtime

Relevant platforms:

- OpenAI Platform
- OpenAI Responses API docs

2. LangChain Tool Interface

Framework-centric orchestration rather than a universal protocol.

Characteristics:

- Tool abstraction
- Agent routing
- Memory + retrieval integration
- Multi-provider support

Good for:

- rapid prototyping
- agent pipelines
- complex orchestration

Less ideal for:

- interoperable ecosystem standards

Resources:

- LangChain
- LangGraph

MCP Alternatives / Competing Standards

3. Agent-to-Agent Protocols (Complementary, Not Replacements)

- These **do not replace MCP**, but solve a different problem.
  - **A2A (Google)** – Agent-to-agent coordination across vendors
  - **ACP (IBM)** – REST-native agent messaging in enterprises
  - **UCP** – Commerce & payment semantics for agents
- **Key insight:**
  Modern systems often combine:
  - **MCP** → agent ↔ tool
  - **A2A / ACP** → agent ↔ agent

Google A2A (Agent-to-Agent)

Focused more on agent interoperability than tool interoperability.

Concept:

- agents communicate capabilities
- delegation between agents
- distributed autonomous workflows

Closer to:

- multi-agent systems
  than:
- simple tool invocation

MCP Alternatives / Competing Standards

4. Semantic Kernel Plugins

Microsoft's plugin abstraction.

Features:

- planners
- connectors
- memory
- function orchestration

Very enterprise-oriented.

Resources:

- Microsoft Semantic Kernel

5. AutoGen

Conversation-oriented multi-agent orchestration.

Patterns:

- agent handoff
- role specialization
- collaborative tool use

More autonomous than MCP.

Resources:

- Microsoft AutoGen

MCP Alternatives / Competing Standards

Related Architectural Patterns

These are not direct MCP replacements but solve adjacent problems.

JSON-RPC Tooling

MCP itself resembles modernized JSON-RPC patterns:

- request/response
- schema-defined capabilities
- transport abstraction

Often used internally in:

- IDE integrations
- local AI tooling
- editor extensions

OpenAPI-Based Tool Exposure

Instead of MCP:

- expose REST APIs
- generate tool schemas automatically
- let models consume OpenAPI specs

Common in:

- enterprise integrations
- SaaS automation

Used by:

- Swagger/OpenAPI Initiative

MCP Alternatives / Competing Standards

GraphQL + LLM Middleware

Alternative strategy:

- expose typed graph
- AI middleware maps intents → queries/mutations

Advantages:

- strong typing
- introspection
- efficient querying

Weakness:

- less native agent semantics

MCP Alternatives / Competing Standards

Workflow Automation Alternatives

Zapier AI Actions

Tool interoperability through workflow automation.

Resources:

- Zapier AI Actions

n8n AI Agent Tools

Visual orchestration for AI tools.

Resources:

- n8n

IDE / Local Tool Ecosystem Alternatives

Language Server Protocol (LSP)

Historically important comparison.

MCP is often described as:

“LSP for AI tools.”

Similarities:

- capability negotiation
- standard protocol
- editor/client interoperability

Resources:

Language Server Protocol

MCP Alternatives / Competing Standards

Conceptual Comparison

System	Primary Focus	Standardized	Multi-Agent	Tool Discovery
MCP	Tool interoperability	Yes	Partial	Yes
OpenAI Tools	Model tool calling	Partial	No	Limited
LangChain	Orchestration framework	No	Partial	Framework-defined
A2A	Agent interoperability	Emerging	Yes	Yes
Semantic Kernel	Enterprise orchestration	Partial	Partial	Plugin-based
AutoGen	Autonomous agents	No	Yes	Dynamic

MCP Alternatives / Competing Standards

Where MCP Is Strongest

MCP excels when you want:

- portable tool ecosystems
- IDE integration
- local + remote tools
- provider-agnostic tooling
- standardized capability discovery

Especially valuable for:

- AI IDEs
- desktop assistants
- secure local tool access
- enterprise agent platforms

Current Industry Direction

The ecosystem appears to be converging toward:

- MCP-like transport standards
- OpenAPI-compatible schemas
- agent-to-agent protocols
- capability discovery
- secure sandboxed tool execution

Likely future stack:

- MCP for tools
- A2A for agents
- OpenAPI/JSON Schema for structure
- OAuth/scoped permissions for security

THANK YOU FOR YOUR ATTENTION