Model Context Protocol (MCP)

The **Model Context Protocol (MCP)** is a new open standard for connecting AI models to external data and tools. It was introduced by Anthropic in late 2024 to solve a big problem: advanced AI (like ChatGPT or Claude) can reason well, but they're "stuck" without data – their knowledge is isolated in silos. MCP acts like a *universal connector* that lets any AI assistant plug into many data sources (files, databases, calendars, web APIs, etc.) and tools (search engines, calculators, code editors, etc.) through one standard interface.

A helpful analogy: **think of MCP like a USB-C port for AI**. Just as USB-C gives a single, common way to plug in keyboards, drives, phones, and displays, MCP gives AI a single way to plug into your files, apps, databases, and online services. This means developers no longer need a custom "cable" or adapter for each AI-data connection – MCP provides a standard "plug" so any compatible AI or tool can connect and share information. By standardizing these connections, MCP helps AI systems get the context they need and produce more accurate, relevant responses.

Why MCP Exists

- **Breaking Down Data Silos.** Before MCP, every time an AI needed data from a new source (like Google Drive, Slack, or a company database), engineers had to write custom code or plugins. This quickly becomes an *N×M problem*: many AI models times many data sources, each with its own connector. MCP solves this by replacing that mess of custom adapters with one **open standard**. Any data source that speaks MCP can connect to any AI client that speaks MCP.

- **Keeping Context Across Tools.** Modern AI assistants often chain together multiple tools: e.g., planning a trip by checking flights, calendars, and maps. MCP lets the AI keep its "context" as it moves between different systems. Instead of starting fresh with each tool, the AI can stay aware of your data and tasks across tools. Over time this means AI agents can behave more *coherently* when using multiple data sources.

- **Open, Collaborative Ecosystem.** Anthropic made MCP **open-source** so anyone can build and share connectors (servers) and clients. Big companies (OpenAI, Google DeepMind) and startups alike are adopting it. For example, in early 2025 OpenAI announced that ChatGPT and its new Agents SDK support MCP, and Google's Gemini models will too. The hope is that MCP becomes a **public good** – a common "plug" that all AI and data tools support, rather than each company locking you into its own ecosystem.

How MCP Works (High-Level)

In MCP's architecture there are **MCP clients** and **MCP servers**.

- **MCP Client:** This is the AI application or chat interface (for example, Claude or a custom LLM-based chatbot). The client initiates conversations and connects to one or more MCP servers.
- **MCP Server:** This is a service that exposes data or actions through MCP. A server might connect to Google Drive, a database, a Git repository, a calendar, or any app. The server "speaks MCP" to share resources and tools with clients.

Under the hood, MCP uses a **JSON-RPC** message protocol: the client and server exchange structured JSON messages over a pipe or network. But you don't need to know the details of JSON-RPC to use MCP. Here's the simple flow:

1. **Connect:** The client (LLM app) connects to an MCP server (via HTTP or stdio). Upon connecting, the server tells the client what it can do – listing available *tools*, *resources*, or *prompts* (see below) with descriptions and input formats.

2. **AI Query:** A user asks the AI (client) a question or gives a task. The client sends this query to the LLM **along with** the list of available tools/resources (so the model knows what it can call).

3. **Tool Planning:** The LLM analyzes the question and decides if it needs to use any tools. If it does, it generates a `tool_use` request inside its response. This tells the client "I want to call tool XYZ with these inputs."

4. **Tool Call:** The client sees the `tool_use` request and invokes that tool on the server. For example, if the tool is "searchFlights(origin, destination)", the client calls the server and gets the flight data back.

5. **Return Results:** The client sends the tool's results back to the LLM (in a special `tool_result` message), and asks the model to continue. The LLM then incorporates the real data into its final answer.

6. **Respond:** The LLM produces the final response, now enriched by the tool data, and sends it back to the user.

In other words, the AI (client) and data source (server) have a conversation: **the model can ask to use tools on the server, get the answers, and continue the chat with that context**. This is how MCP **two-way** connections work. The diagram above illustrates multiple clients and servers – any combination of an AI app and a data source can connect as long as both support MCP.

Core Concepts: Tools, Resources, Prompts

MCP builds on a few core ideas, inspired by other developer protocols:

- **Tools (Functions):** These are actions an AI can command on the server. Each tool is defined with a name, a description, and a JSON input/output schema. For example, a "searchFlights" tool might take `{origin: "NYC", destination: "PAR", date:`

`"2025-12-01"}` and return flight options. MCP servers **list** their tools to the client (via a `tools/list` call). When the LLM requests a tool, the client calls that function (over the MCP protocol) and returns the result back to the model.

- **Resources:** These are data or context the AI can **read** but not change. Examples: documents, database rows, calendar events, code files. Each resource has a unique URI (like `file:///path/to/doc.txt` or `postgres://mydb/schema`) and a MIME type. The AI can retrieve resource contents if it needs them as context. Resources can be fixed (direct URIs) or templated with parameters (e.g. `db://users/{id}`) so the model can query them dynamically.

- **Prompts:** These are pre-built message templates or workflows on the server. A prompt might be something like "/vacationPlan" that the model can call to get started on a multi-step task (plan trip, summarize meetings, etc.). If a server provides prompts, the model can invoke them to use a chain of tools or multi-step logic easily. (Think of prompts as "macros" for the AI to follow a script or template.)

By defining tools, resources, and prompts in a standard way, MCP lets any AI client discover what an MCP server can do. Notably, MCP was **inspired by the Language Server Protocol (LSP)** that editors use to talk to language tools. In the same spirit, MCP standardizes how AI models talk to context sources.

Design Principles

Anthropic designed MCP around a few key principles:

- **Standardized Discovery:** Every MCP server and client follows the same protocol, so tools and data can be listed and described in a uniform way. This means an AI can automatically know what tools are available on any connected server, instead of needing custom code per server.

- **Two-Way, Secure Connections:** MCP connections are **bi-directional** and respect security. The protocol supports OAuth or other authentication for connecting to servers, and encourages user consent for data access or actions. For example, before a model uses a tool on your behalf (like sending an email), the user can be prompted to approve it.

- **Openness and Reusability:** MCP is **open-source**, with SDKs in Python, TypeScript, Java, C#, etc. There's a library of reference server implementations (e.g. for GitHub, Slack, Google Drive) and developers can build custom servers for any new system. Because it's a public standard (like OpenAPI is for HTTP APIs), different AI platforms can interoperate. In fact, OpenAI and Google DeepMind have already said they will support MCP.

- **Simplicity and Modularity:** MCP deliberately keeps the architecture straightforward. You either build an **MCP server** to expose your data, or build an

MCP client (AI app) that connects to servers. There's no heavy middleware or new database – just standard messages over HTTP or stdio. This makes it easier to maintain and evolve. As one Anthropic exec put it, MCP is meant to be a "universal translator" so **"AI connects to any data source"**.

- **User Control & Safety:** The MCP spec emphasizes user consent and data privacy. For example, the protocol requires that users **explicitly approve** any data access or tool use. The client should clearly show what data it will send and what actions it will perform. This helps prevent hidden behavior (like a model secretly sending your files to the cloud) and keeps the user in the loop.

Analogies and Intuition

- **USB-C / Universal Adapter:** As mentioned, MCP is like giving AI a universal port. Without MCP, each tool or data source is a different plug; MCP makes them all fit one socket.

- **Language Server for AI:** In programming, the Language Server Protocol (LSP) lets any code editor use language-specific tools (linters, compilers) through a standard interface. MCP does the same for AI: it's the **LSP for AI context**. Any AI "editor" can browse and use any "server" of data through the same protocol.

- **Swiss Army Knife:** Think of the AI as a person, and MCP servers as pockets of tools (pliers, scissors, knives). MCP is like having a Swiss Army Knife toolbox: the AI can pull out exactly the tool it needs (flight search, database query, etc.) without having to carry separate toolboxes.

- **Multi-lingual Translator:** One Anthropic engineer called MCP a **"universal translator"** for data. In a meeting of people speaking different languages, you'd normally need a translator for each pair. MCP is like a single common language so everyone (all AI and data tools) understands each other.

MCP vs. Other Approaches

Before MCP, developers used things like OpenAI's **Function Calling** or ChatGPT **Plugins** to connect AI to tools. MCP is similar in spirit but has some key differences:

- **OpenAI Function Calling:** This feature lets you define specific functions (with JSON schemas) that an OpenAI model can call. It works well for simple cases, but it's tied to OpenAI's API and you have to declare each function in your prompt ahead of time. MCP, by contrast, is an open protocol not tied to one vendor. It allows the model to **dynamically** discover and call tools on many servers in a conversation. Function calling is like a fixed function library for one AI, whereas MCP is a dynamic ecosystem of servers and tools.

- **ChatGPT Plugins (OpenAI):** ChatGPT plugins let ChatGPT access web services (like a browser or database), but each plugin is specific to ChatGPT and requires building a manifest. MCP aims to be broader: any LLM (Claude, ChatGPT, GeminAI, etc.) can be a client, and any data system can be a server, using the same protocol. In fact, major AI providers (OpenAI, Google) are adopting MCP to make their tools interoperate.
- **LangChain / Custom Code:** Many AI devs currently write custom code or use frameworks (like LangChain) to connect to data sources. MCP can replace much of that boilerplate. Instead of hand-writing API calls or embeddings for each source, developers write or install an MCP server once, and any MCP-compatible AI can use it. This saves development time and reduces duplication.

In short, MCP is *vendor-neutral and more flexible* than single-provider features. It provides **standard discovery, bidirectional calls, and multi-tool workflows**, whereas older approaches are typically single-tool or single-vendor.