Introduction to MCP

What the Model Context Protocol (MCP) is, the N x M integration problem it solves, the host/client/server roles, and the tools/resources/prompts primitives a server can expose.

Intermediate 18 min

Last updated: August 22, 2026
Read online: https://www.learnforgex.com/en/topics/introduction-to-mcp

---

Introduction to MCP

"Tools and Function Calling" described the mechanism: a model requests a function call, and an application executes it. What that lesson left open is *how* an application actually offers a model a consistent set of tools, data, and prompts -- especially when the tools and data live in many different places (a filesystem, a company database, a third-party API) and the application talks to more than one AI model or vendor. The **Model Context Protocol (MCP)** is the open, standardized answer to that question.

What Is MCP?

MCP is an open protocol that standardizes how an AI application connects to external tools, data sources, and prompt templates. Rather than every application inventing its own way to describe and call tools (as "Tools and Function Calling" left implementation-specific), MCP defines one consistent format for exposing them, one consistent way to discover what's available, and one consistent way to invoke it -- regardless of which AI model is on the other end, or which programming language the tool happens to be written in. It doesn't replace the tool-calling loop from the previous lesson; it standardizes the *server side* of it, so the same tool implementation can be reused by any MCP-compatible application.

Why Does It Exist?

Without a shared standard, connecting *M* different AI applications to *N* different tools and data sources requires a large number of separate, custom integrations -- an application built with one vendor's tool-calling format has to be rewritten to work with another's, and every new data source needs bespoke glue code for every application that wants to use it. MCP's purpose is to standardize that integration architecture, reducing the need to write a separate, custom integration between every AI application and every tool or data source: a tool or data source is implemented once, as an MCP server, and any MCP-compatible

application can connect to it, unmodified. This is often summarized as "$M \times N \rightarrow M + N$" -- keep in mind that's not a precise mathematical cost formula, just a simplified mental model for holding onto the idea quickly. This is a direct, practical answer to what "LLM Capabilities and Limitations" and "Tools and Function Calling" both established: models need a reliable way to reach current information and take real actions, and that access has to be reusable rather than rebuilt for every application and every tool.

Host, Client, and Server

MCP defines three roles that stay the same across every setup:

- **Host** -- the AI application the person actually interacts with (a chat app, an IDE, a custom agent). The host is responsible for talking to the LLM and for deciding when to use MCP.
- **Client** -- lives inside the host and maintains a single, one-to-one connection to exactly one server. A host that needs to reach three different servers runs three clients internally, one per connection.
- **Server** -- a separate program that exposes tools, data, or prompt templates to any client that connects to it. A server doesn't know or care which host it's talking to, or which underlying LLM the host uses -- it only speaks the protocol.

This separation is what makes the "write once, use anywhere" property work: a server built to expose, say, a company's internal ticketing system, doesn't need to know anything about the specific chat application, IDE, or LLM that will eventually call it.

What a Server Exposes: Tools, Resources, and Prompts

An MCP server can expose three kinds of primitives, though a given server is free to offer just one:

- **Tools** -- executable functions, in exactly the sense described in "Tools and Function Calling": a name, a description, and a parameter schema, invoked to perform an action or computation and return a result.
- **Resources** -- readable data the host application can pull in, like a specific file's contents or a database record, identified by a URI. A resource is read, not executed -- it supplies information rather than performing an action.
- **Prompts** -- reusable prompt templates a server can offer, so common or complex prompting patterns don't have to be duplicated inside every host application that wants to use them.

This course's hands-on example in "Building an MCP Server" focuses on tools specifically, since they connect most directly to "Tools and Function Calling" -- but a real server is free to combine all three kinds of primitives.

MCP and the Tool-Calling Loop

MCP doesn't change the tool-calling loop from the previous lesson; it standardizes two of its steps. Step 1 of that loop -- the application learning what tools are available -- becomes, in MCP terms, the client asking a connected server to list its tools, and the server responding with each tool's name, description, and parameter schema, all following one shared format. Step 4 -- the application actually executing a function -- becomes the client sending a structured call to the server over that same connection, and the server running the real code and returning the result. Everything in between (the model deciding whether and how to call a tool) is unchanged and still happens exactly as described in "The Tool-Calling Loop" -- MCP standardizes how tools are discovered and invoked, not how a model decides to use them.

Best Practices

- Think of an MCP server as reusable infrastructure, not a one-off integration -- design it around what the underlying data source or system can genuinely do, not around one specific host application's needs (see "Why Does It Exist?").
- Keep the host/client/server separation in mind when debugging: a client only ever talks to one server, so a host connected to multiple servers is really running multiple independent client connections in parallel (see "Host, Client, and Server").
- Choose the right primitive for the job -- data the model should simply read belongs in a resource, not a tool that returns the same data, since a tool implies an executable action (see "What a Server Exposes: Tools, Resources, and Prompts").

Common Mistakes

- **Assuming MCP is a specific AI model or a new tool-calling format.** As "What Is MCP?" explained, MCP is a protocol that standardizes how existing tool-calling concepts (from "Tools and Function Calling") get discovered and invoked -- it doesn't replace or compete with the underlying idea of tool use.
- **Treating "server" as meaning a client-facing web server.** In MCP terms, as "Host, Client, and Server" defined, a server is whatever program exposes tools, resources, or prompts to a client -- it can just as easily be a small local process as a remote web service.
- **Building one server per host application instead of one server per data source.** "Why Does It Exist?" showed that MCP's whole value comes from a server being reusable across hosts -- designing around a single host's quirks throws that reusability away.

Summary, Cheat Sheet, and Glossary

Summary

- MCP (Model Context Protocol) is an open protocol that standardizes how AI applications connect to external tools, data, and prompt templates.
- It exists to standardize integration architecture, reducing the need for every application to be custom-wired to every tool -- often summarized as "$M \times N \rightarrow M + N$" (a simplified mental model, not a precise formula): implement a data source once as a server, use it from any compatible host.
- Three roles stay constant: the **host** (the AI application), the **client** (inside the host, one per server connection), and the **server** (exposes functionality, independent of any specific host).
- A server can expose **tools** (executable actions, as in "Tools and Function Calling"), **resources** (readable data), and **prompts** (reusable templates).
- MCP standardizes tool discovery and invocation -- the two application-side steps of the tool-calling loop -- without changing how a model decides to use a tool.

Cheat Sheet

- MCP = open protocol standardizing tool/data/prompt access for AI applications.
- Problem solved: M x N custom integrations -> M + N (write once, use anywhere).
- Host = AI application. Client = inside host, one per server. Server = exposes functionality.
- Primitives: tools (actions), resources (readable data), prompts (reusable templates).

Glossary

- **Model Context Protocol (MCP):** an open protocol standardizing how AI applications connect to external tools, data sources, and prompt templates.
- **Host:** the AI application a person interacts with; responsible for talking to the LLM and deciding when to use MCP.
- **Client:** the component inside a host maintaining a one-to-one connection to a single MCP server.
- **Server:** an independent program exposing tools, resources, or prompts to any connecting client.
- **Resource (MCP):** readable data a server exposes, identified by a URI, supplying information rather than performing an action.
- **Prompt (MCP):** a reusable prompt template a server can offer to connecting clients.