Function calling and Model Context Protocol (MCP) are not competing ideas.

They handle different parts of the same loop.

Function calling is how the model asks for a tool. MCP is one way your app finds tools, documents, and reusable prompts from servers.

If you read for 1 second:

The model picks a tool from a menu. Your app decides whether to run it. MCP can help fill that menu.

If you read for 10 seconds:

- the model cannot reach into your systems by itself
- function calling lets it request a named tool with structured arguments
- your app validates that request and runs the real code
- MCP lets your app discover tools from reusable servers instead of wiring every tool by hand

For example, if a user asks, "Where is order A100?", function calling lets the model request `get_order_status` with `order_id: "A100"`. MCP is one way your app can make `get_order_status` available from an orders server.

This walkthrough was prompted by a useful public ChatGPT discussion about function calling and MCP. The explanations below are grounded in the official OpenAI function calling guide, the OpenAI MCP and connectors guide, the MCP architecture specification, and the MCP tools specification.

If you read for 100 seconds

If you have about a minute, remember the five-part flow:

1. Your app gives the model a small tool menu.
2. The user asks a question.
3. The model requests one tool from that menu.
4. Your app checks the request and runs the real code.
5. Your app sends the result back to the model.

MCP fits before and during that flow. Before the model sees the menu, an MCP client can ask an MCP server what tools it offers. During execution, the host can

route the chosen call back to that server.

MCP server lists tools
Host shows selected tools to the model
Model asks for one tool
Host checks and routes the request
Tool result goes back into model context

That is the core idea. The rest of the article explains why those boundaries matter in production.

Start with what the model can see

A language model is not your database, browser, payment processor, filesystem, or warehouse system. It can reason over the context you send it, but it cannot inspect your production systems on its own.

Without tools, the model can only answer from its prompt, its current context, and its trained statistical patterns.

With function calling, the runtime gives the model a controlled menu of available operations. The model can then return a structured request like:

```
{
  "name": "get_order_status",
  "arguments": {
    "order_id": "A100"
  }
}
```

The model did not run `get_order_status`.

It selected a tool name and proposed arguments. Your app still has to validate the call, execute real code, and send the result back.

That separation matters because the model is not the authority. Your runtime is.

Function calling in one loop

A basic function calling loop works like this:

1. The developer defines a tool schema.
2. The app sends the user message and tool schema to the model.
3. The model decides whether it can answer directly or should call a tool.

4. If a tool is needed, the model returns a structured tool call.
5. The app validates the tool name and arguments.
6. The app executes real code or calls a real API.
7. The app sends the tool result back to the model for the final answer.

OpenAI's guide describes this as a multi-step conversation: the application gives tools to the model, the model returns tool calls, and the application returns tool outputs for the model to use.

The important point is simple: function calling does not put your functions inside the model. It lets the model request functions that your app made available.

Why schemas matter

The schema is the menu item the model reads.

For example:

```
{
  "type": "function",
  "name": "get_order_status",
  "description": "Look up the shipping and fulfillment status of one order.",
  "parameters": {
    "type": "object",
    "properties": {
      "order_id": {
        "type": "string",
        "description": "The internal order ID."
      }
    },
    "required": ["order_id"],
    "additionalProperties": false
  }
}
```

This schema tells the model three things:

- the operation is called `get_order_status`
- it should be used for shipping and fulfillment status
- it requires one string argument called `order_id`

The model can use that description even if it never saw your exact function during training.

Tool design is product design. Bad names, vague descriptions, broad arguments, or too many similar tools make the model's choice harder.

Your app owns safety

The model's structured call is only a proposal.

Your app should still ask:

- is this tool allowed for this user
- are the arguments valid
- is this read-only work or a real action
- should the user confirm it first
- should a human approve it
- should this call be rate-limited, logged, or blocked

For a read-only tool like `search_docs`, automatic execution may be acceptable. For an action tool like `refund_payment`, `publish_video`, or `delete_file`, automatic execution can be dangerous.

A practical default is:

Tool class	Example	Gate
Read-only	`search_docs`	Automatic after validation
Low-risk write	`create_draft`	Usually automatic with audit log
External side effect	`send_email`	User confirmation
High-risk action	`issue_refund`	Human approval or strict policy

This is the basic split:

- the model asks for work
- the runtime decides what is allowed
- real code does the work

Where MCP enters

MCP gives your app a standard way to bring in tools and context.

Instead of hardcoding every tool schema directly inside one application, an MCP host can connect to one or more MCP servers. Each connection is handled by an

MCP client inside the host.

The MCP architecture spec defines this as a host, client, and server model:

- the host coordinates the AI application and manages clients
- each client maintains a dedicated connection to one server
- each server exposes focused capabilities such as tools, resources, and prompts

This gives you a cleaner boundary:

```
AI host
  -> MCP client for GitHub server
  -> MCP client for Slack server
  -> MCP client for warehouse server
  -> MCP client for filesystem server
```

Each server can evolve separately. The host does not need to hand-write every integration from scratch.

MCP tools in one loop

For tools, MCP usually starts before the model sees anything.

At connection time:

1. The host creates an MCP client for a configured server.
2. The client initializes the session with the server.
3. The client asks the server for available tools with `tools/list`.
4. The server returns tool metadata, including names, descriptions, and `inputSchema`.
5. The host converts those discovered tools into model-visible tool definitions.

At runtime:

1. The user asks a question.
2. The host sends the model the user message and relevant tool definitions.
3. The model returns a tool call.
4. The host maps that tool call back to the owning MCP server.
5. The MCP client invokes the server with `tools/call`.
6. The server executes the operation and returns a result.
7. The host feeds the result back into the model context.

The MCP tools spec is explicit about the protocol messages: clients discover tools with `tools/list`, and invoke tools with `tools/call`.

So the model-facing surface can still look like function calling. MCP changes where the tool definitions came from and how execution is routed.

Tools are not the whole MCP story

Tools are only one MCP primitive.

MCP servers can also expose resources and prompts.

Resources are read-oriented context. A resource might be a file, document, database record, repository note, or API response that the host can choose to include in model context.

Prompts are reusable templates or workflows. They let a client fetch a structured prompt from a server instead of copying the same prompt by hand.

A useful shorthand is:

MCP primitive	First-principles role	Who usually controls it
Tool	Do something or fetch something on request	Model controlled through the host
Resource	Provide context to read	Application controlled
Prompt	Reuse a structured instruction template	User or application controlled

This is why MCP is named Model Context Protocol rather than Model Tool Protocol. Tools are the most visible primitive, but not the whole system.

Same model action, different tool source

The distinction becomes clearer if you compare the two loops:

Question	Plain app tool	Tool from MCP
Who defines the tool list	The application developer	MCP servers expose tools, host imports them

How does the model see tools	The app sends tool schemas	The host sends schemas from MCP
Who executes the call	The app's code	MCP server via host and client
Who owns permissions	The application runtime	The host, plus server and connector policy
Why use it	Simple custom app integration	Reusable connector and context boundary

MCP does not remove the need for function calling. In many systems, MCP feeds function calling.

The combined loop is:

MCP server exposes tool schema
MCP client discovers tool schema
Host makes tool schema model-visible
Model selects tool through function calling
Host routes call back through MCP
MCP server executes the operation
Host returns result to model
Model answers the user

Remote MCP through an API

Some platforms can connect to remote MCP servers through their API tool system.

OpenAI's MCP and connectors guide describes an `mcp` tool type for remote MCP servers and connectors in the Responses API. In that model, a developer can provide a server URL and let the API import tools, call them, and place the result in model context.

That shortens the code you write, but the same ideas still exist:

- tool discovery
- tool selection
- tool call execution
- result handling
- permission and approval policy

The responsibility does not disappear. Some of it moves into the provider's runtime.

If you read for 1000 seconds

For production agents, keep the layers separate:

Layer	Responsibility
Model	Decide whether a tool is useful and propose arguments
Host or runtime	Manage context, permissions, routing, retries, and stopping
MCP client	Maintain one protocol connection to one server
MCP server	Expose focused tools, resources, or prompts
Executor code	Touch the actual external system

When bugs happen, this layering helps you debug.

If the model chose the wrong tool, inspect tool names, descriptions, and the context it saw.

If the right tool failed, inspect runtime validation, MCP routing, server logs, and external API errors.

If an unsafe operation was attempted, inspect policy gates rather than blaming the schema alone.

What to remember

Function calling lets the model ask for a specific tool. MCP lets apps discover useful tools from reusable servers.

Function calling: the model asks.
Runtime: your app checks and runs.
MCP: servers can supply the tools.

If you remember one practical rule, make it this:

Let the model request work. Let the runtime decide what work is available, who owns it, whether it is safe, and how the result returns to context.