Chapter 2 - MCP Fundamentals

Recap

In the last video, we talked about why AI needs tools, how to communicate between AI and Tool and why a standard like MCP is necessary.

Today, we're diving deeper into the *MCP fundamentals*— what it's made of, how it works, and why it's such a game-changer.

Core Components — Host, Client, Server

"Let's start with the **core building blocks of MCP**.

MCP is based on three key roles:

- **Host** → This is the environment where the model lives. Think of it as the platform that manages the conversation.
- **Client** → A component within the host application that manages communication with a specific MCP Server.
- **Server** → This is where the tools live — calculators, search engines, databases, whatever you want.

Now here's the cool part:

- The LLM doesn't need to know how to format for tool calling.
- The *Server* doesn't care which LLM is calling it.
- The Host and Client mediates between LLM and MCP server.

This separation makes MCP scalable and reusable across different AIs and tools."

For more information:

https://modelcontextprotocol.io/specification/2025-06-18/architecture

---

2.2 Tool Discovery & Tool Schemas

"Okay, so how does a model even know what tools are available?

To discover available tools, clients send a `tools/list` request.

Then the server responses about tools information. It includes,

- name: tool name
- description: what the tool does
- inputSchema: what inputs it needs.
- required: which arguments must be set

For example, a get_weather tool might look like this in MCP:

```
...
   "tools": [
    {
     "name": "get_weather",
     "description": "Get current weather information for a location",
     "inputSchema": {
      "type": "object",
      "properties": {
       "location": {
        "type": "string",
        "description": "City name or zip code"
       }
      },
      "required": ["location"]
     }
...
```

This means the model knows *exactly* how to call the tool and what kind of answer to expect back.

No guesswork, no messy parsing — just clear contracts.”

For more information:

https://modelcontextprotocol.io/specification/2025-03-26/server/tools

---

2.3 Calling tools

So, let’s say the client wants to call a specific tool. In that case, the client sends a `tools/call` request.

When making a tool call, two things need to be specified: the tool name and the arguments.

Request Example:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    }
  }
}
```

The server then receives this request, checks if it's valid, and if so, calls the tool. After that, it sends back a response with the result, like this:

Response Example:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nCondition
s: Partly cloudy"
      }
    ],
    "isError": false
  }
}
```

This way, the client and server can communicate with each other smoothly.

2.4 Protocol Mechanics — JSON-RPC 2.0

"Now let's talk about **how MCP actually works under the hood**.

Instead of custom APIs or imports, MCP uses **JSON-RPC 2.0**.

The MCP client and server communicate through request and response using JSON format and lightweight JSON-RPC 2.0 protocol. This design choice makes MCP incredibly flexible.

1. **Tool calling works like a client-server system**

   In the previous video, we just talked about local python function.

   But it doesn't need to be local python function. Tool can be external API.

   The AI sends a structured request to the tool,

   Tool makes the result following by the LLM's request

   LLM gets the tool's result and then continues generating its response.

2. **We want to use existing tools easily**

   Like I mentioned right before, Tool doesn't need to be local function. It can be the any API whchi we can connect to through internet. In this way, standard tool like MCP should support network communication not just standard input/ouput in local network.

3. **We want streaming service.**

   Many users want real-time results. Most LLMs generate text token by token, so we don't want to wait until the full response is ready.

4. **Tool developers don't want to make an complex API server for making the tools.**

   Even though creating a web server is easier today than it used to be, it's still a bit of a hassle. Most developers just want to define their tool's input and output formats, run it, and let it work with AI. They don't want to deal with full-blown HTTP endpoints, routing, or server management.

   MCP solves this by letting tools speak a **common protocol**—JSON-RPC 2.0. With this, any tool can communicate with AI without needing to become a web server. All it needs is a way to accept structured requests and return structured responses.

   This makes it much easier for developers to create and share tools, while AI can still interact with them reliably—whether the tool runs locally, on another machine, or in the cloud.

For supporting these needs, MCP selects **JSON-RPC 2.0** protocol,

and supports multiple transports, like:

- **stdio** (standard input/output) → great for local tools.
- **Streamable HTTP** → perfect for networked tools.
    - Previously, streamable HTTP wasn't available. To work with internet protocols, the model context protocol used **stdio + SSE (Server-Sent Events)**. However, this approach was later replaced with streamable HTTP. If you're interested in the reasoning behind this change, you can check GitHub issue #206.

This design choice makes MCP more universal.

For more information:

https://modelcontextprotocol.io/specification/2025-03-26/basic/transports

https://www.jsonrpc.org/specification

https://chatgpt.com/share/68b186bd-c728-8010-8ed4-128e7454f0d9

> https://github.com/modelcontextprotocol/modelcontextprotocol/pull/206

https://mangkyu.tistory.com/442

2.5 Relationship with Function Calling Feature

"Now you might be thinking — wait, isn't this just *function calling*?

Well... not exactly! Let me explain why MCP is actually a game-changer.

Function calling in models like GPT is great, but it's tied to the platform.

For example, OpenAI defines its own function format, Anthropic defines another one, and so on.

MCP goes **further** because it's a **protocol, not a feature**.

It standardises tool definitions, discovery, and communication in a way that works across all models and hosts.

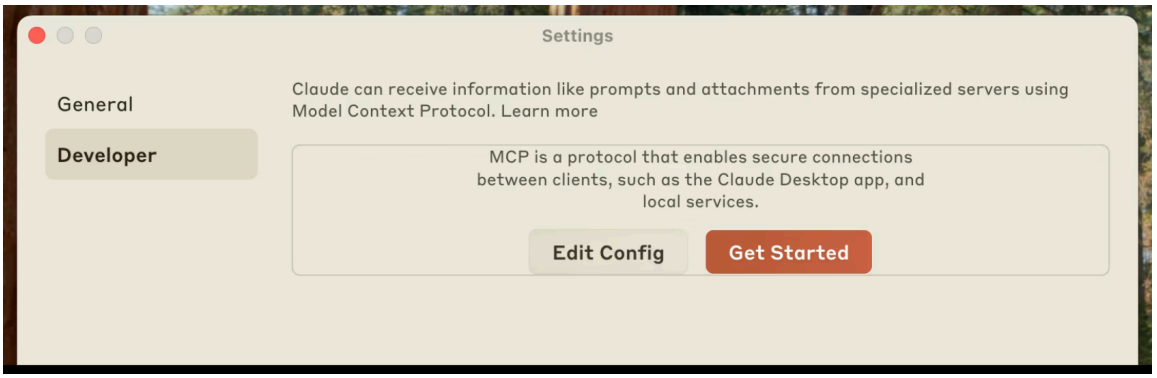
So instead of re-writing the same function definitions for each platform, MCP gives us a universal layer.

But, here's where it gets interesting — the adoption story has two sides, and it really depends on whether you're an end user or a developer building apps.

For end users and desktop tools, MCP is incredibly smooth:

- Claude Desktop? Just copy-paste a JSON config file and you're done

  Click the “Edit Config” button to open the configuration file:




```
{
 "mcpServers": {
  "filesystem": {
   "command": "npx",
   "args": [
    "-y",
    "@modelcontextprotocol/server-filesystem",
    "C:\\Users\\username\\Desktop",
    "C:\\Users\\username\\Downloads"
   ]
  }
 }
}
```

- Claude Code? Same thing - super simple setup

```
# claude code MCP connector for sse transport

# Basic syntax
claude mcp add --transport sse <name> <url>

# Real example: Connect to Linear
```

```
claude mcp add --transport sse linear https://mcp.linear.app/sse

# Example with authentication header
claude mcp add --transport sse private-api https://api.company.com/mcp \
  --header "X-API-Key: your-key-here"
```

- Cursor IDE? same thing

  For more information:

  https://modelcontextprotocol.io/quickstart/user
  https://docs.anthropic.com/en/docs/claude-code/mcp#option-2%3A-add-a-remote-sse-server

  https://docs.cursor.com/en/context/mcp#protocol-support

  https://techcommunity.microsoft.com/blog/azure-ai-foundry-blog/model-context-protocol-mcp-integrating-azure-openai-for-enhanced-tool-integratio/4393788

But for developers building apps with APIs? That's where it gets more complex.

Many other AI providers support MCP now. But it has a slightly different format yet.

OpenAI API MCP connector:

```
from openai import OpenAI

client = OpenAI()

resp = client.responses.create(
    model="gpt-5",
    tools=[
        {
            "type": "mcp",
            "server_label": "dmcp",
            "server_description": "A Dungeons and Dragons MCP server to assist with dice rolling.",
            "server_url": "https://dmcp-server.deno.dev/sse",
            "require_approval": "never",
```

```
        },
    ],
    input="Roll 2d4+1",
)

print(resp.output_text)
```

Claude API MCP connector:

```
import anthropic

client = anthropic.Anthropic()

response = client.beta.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=1000,
    messages=[{
        "role": "user",
        "content": "What tools do you have available?"
    }],
    mcp_servers=[{
        "type": "url",
        "url": "https://mcp.example.com/sse",
        "name": "example-mcp",
        "authorization_token": "YOUR_TOKEN"
    }],
    betas=["mcp-client-2025-04-04"]
)
```

So here's the current state: if you're just using desktop apps like Claude or Cursor, setting up MCP is very easy - literally just copy and paste a config file and you're done.

But if you're a developer building your own AI app? You still need to write code to handle the MCP integration - setting up server connections, managing authentication, handling responses, and all that technical stuff.

The good news is that popular developer tools are stepping in to bridge this gap. For example, LangChain now has MCP adapters that handle all the complex integration work for you - so instead of writing dozens of lines of

setup code, you can just use their pre-built connectors and focus on building your actual app.

For more information:

https://platform.openai.com/docs/guides/tools-connectors-mcp?quickstart-panels=remote-mcp

https://openai.github.io/openai-agents-python/mcp/

https://docs.anthropic.com/ko/docs/agents-and-tools/mcp-connector

> https://github.com/langchain-ai/langchain-mcp-adapters

Recap

So to recap:

we have learned

- MCP is built on Host, Client, and Server roles.
- MCP Client sends `tools/list` request to know which tools exist and the spec of tools
- Client sends `tools/call` request to call the tool in the MCP server
- MCP runs on JSON-RPC with stdio/Streamable HTTP for maximum flexibility.
- And it goes beyond function calling by standardising everything.

In the next video, we'll look at MCP *in action* with real code examples.
