Why MCP?

AI development is often fragmented and brittle. MCP solves this by:

- Reducing fragmentation between models, tools, and data sources.
- Recognizing that LLMs are only as effective as the context they are given.
- Enabling smoother, consistent integration of tools, APIs, and resources into LLM workflows.

What is MCP?

MCP (Model Context Protocol) is a standard interface between language models and external context providers, such as:

- Tools (functions)
- Data sources (resources)
- Prompt templates

Just like:

- REST standardizes application communication
- LSP standardizes IDE-language tooling
- MCP standardizes LLM-context communication

Core Ideas:

- MCP is a standardization layer, not a reinvention.
- Everything possible with MCP can be done without it—just in a fragmented, error-prone way.
- Anyone can author MCP servers. Many reusable open-source servers are already available.

Isn't MCP just an API wrapper?

In some ways, yes—but it's more structured and LLM-native.

Key Benefits:

- Unified interface for tools, resources, and templates
- Strong typing, better error handling
- Easy integration with tools like LangChain or OpenDevin

MCP Architecture

MCP uses a Client-Server model built around three primitives:

1. Tools

Functions the model can invoke with parameters.

Example:

```
@mcp.tool()
def add(a: int, b: int) -> int:
    "Add two numbers"
    return a + b
```

Use Cases: Database updates, API calls, internal logic

2. Resources

Read-only content accessible by the model.

Static Resource Example:

```
@mcp.resource("config://version")
def static_greeting():
    return "Hello"
```

Dynamic Resource Example:

```
@mcp.resource("users://{name}")
def dynamic_greeting(name: str):
    return f"Hello {name}"
```

Use Cases: Config files, user profiles, DB records

3. Prompt Templates

Reusable templates for common LLM prompts.

Example:

```
@mcp.prompt()
def custom_prompt(name: str) -> str:
    "Generate a prompt asking for a summary."
    return f"Hello from prompt, {name}"
```

Use Cases: Q&A, summaries, structured JSON responses

Client-Server Initialization

To connect a client with an MCP server:

1. Client sends an initialization request
2. Server replies with an initialization response
3. Client sends an initialization notification

This ensures that the client understands what tools/resources are available.

MCP Transports

Depending on the setup, different transports are used:

Mode	Protocol	Use Case
Local	Stdio	CLI or test setups
Remote	http + sse	Streaming responses
Remote	streamable http	Stateless or stateful HTTP servers

Creating an MCP Server

Minimal server using fastmcp:

```
from fastmcp import FastMCP

mcp = FastMCP("Demo")

if __name__ == "__main__":
    mcp.run()
```

After running, tools, resources, and prompts can be accessed by any compatible MCP client.

Complete Server Implementation

```
# server.py
from fastmcp import FastMCP

mcp = FastMCP("Demo")

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

@mcp.resource("config://greeting")
def static_greeting():
    return "Hello"

@mcp.resource("users://{name}")
def dynamic_greeting(name: str):
    return f"Hello {name}"

@mcp.prompt()
def custom_prompt(name: str) -> str:
    """Generate a prompt asking for a summary."""
    return f"Hello from prompt, {name}"

if __name__ == "__main__":

    mcp.run()
```

Complete Client Implementation

```
import asyncio
from fastmcp import Client

async def main():
    async with Client("server.py") as client:
        tools = await client.list_tools()
        print(f"Available tools: {tools}\n")

        result = await client.call_tool("add", {"a": 5, "b": 3})
        print(f"Result: {result[0].text}\n")

        static_resources = await client.list_resources()
        print(f"Available static resources: {static_resources}\n")

        static_resource_response = await
client.read_resource("config://greeting")
        print(f"Result: {static_resource_response[0].text}\n")

        dynamic_resources = await client.list_resource_templates()
        print(f"Available dynamic resources: {dynamic_resources}\n")

        dynamic_resource_response = await
client.read_resource("users://jainil")
        print(f"Result: {dynamic_resource_response[0].text}\n")

        prompts = await client.list_prompts()
        print(f"Available Prompts: {prompts}\n")

        prompt = await client.get_prompt("custom_prompt",
                                arguments={"name": "Jainil"})
        print(f"Result: {prompt}")


if __name__ == "__main__":
    asyncio.run(main())
```

Example Use Case: AI Support Agent

Using MCP to build a support chatbot:

- Tools: fetch_ticket, escalate_issue
- Resources: faqs://product-x, files://docs/manual.pdf
- Prompts: summarize_transcript, reply_template

This architecture reduces prompt complexity and helps scale context-rich AI features with ease.

Summary

MCP Components	Purpose
Tool	MCP Server Primitive, Invokable function
Resource	MCP Server Primitive, Read-only data source
Prompt Template	MCP Server Primitive, Standardized prompt creation
Transport	Communication method
Server	Hosts all MCP primitives
Client	Connects and consumes server features