This demo implementation demonstrates how to build and run a FastAPI while exposing its routes as MCP-SERVER tools:

It shows how to:

Define custom endpoints (/sum, /greet, /translate, /alerts)

Expose them as MCP tools via FastApiMCP

Connect and test the server with MCP-compatible clients (like Claude Desktop)

Validate that LLMs (in vscode plugins) can dynamically invoke these tools to extend their capabilities.

What is MCP?

The Model Context Protocol (MCP) is an open standard protocol designed to enable seamless communication between AI models (particularly large language models) and external tools, services, and data sources. MCP provides a structured, standardized way for AI systems to:

- Discover available tools and capabilities in their environment
- Request specific actions or data from external systems
- Receive structured responses that can be easily processed and incorporated into AI responses
- Maintain context across multiple interactions with external systems

Using MCP, AI applications like Claude or ChatGPT can connect to data sources (e.g. local files, databases), tools (e.g. search engines, calculators) and workflows (e.g. specialized prompts)—enabling them to access key information and perform tasks.

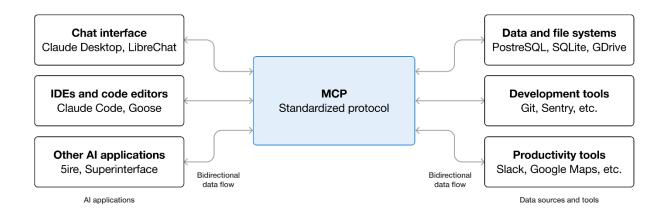


Figure 1- MCP

What can MCP enable?

- Agents can access your Google Calendar and Notion, acting as a more personalized AI assistant.
- Claude Code can generate an entire web app using a Figma design.

- Enterprise chatbots can connect to multiple databases across an organization, empowering users to analyze data using chat.
- Al models can create 3D designs on Blender and print them out using a 3D printer.

Core MCP Concepts

MCP servers can provide three main types of capabilities:

- 1. **Resources**: File-like data that can be read by clients (like API responses or file contents)
- 2. **Tools**: Functions that can be called by the LLM (with user approval)
- 3. **Prompts**: Pre-written templates that help users accomplish specific tasks

Concepts of MCP

Participants

MCP follows a client-server architecture where an MCP host — an AI application like **Claude Code** or **Claude Desktop** — establishes connections to one or more MCP servers. The MCP host accomplishes this by creating one MCP client for each MCP server. Each MCP client maintains a dedicated one-to-one connection with its corresponding MCP server. The key participants in the MCP architecture are:

- MCP Host: The AI application that coordinates and manages one or multiple MCP clients
- MCP Client: A component that maintains a connection to an MCP server and obtains context from an MCP server for the MCP host to use
- MCP Server: A program that provides context to MCP clients

For example: Visual Studio Code acts as an MCP host. When Visual Studio Code establishes a connection to an MCP server, such as the **Sentry MCP server**, the Visual Studio Code runtime instantiates an MCP client object that maintains the connection to the Sentry MCP server. When Visual Studio Code subsequently connects to another MCP server, such as the **local filesystem server**, the Visual Studio Code runtime instantiates an additional MCP client object to maintain this connection, hence maintaining a one-to-one relationship of MCP clients to MCP servers.

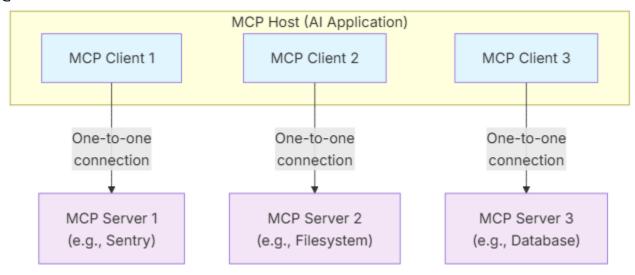


Figure 2 - MCP (HOST / SERVER)

Note that **MCP** server refers to the program that serves context data, regardless of where it runs. MCP servers can execute locally or remotely. For example, when Claude Desktop launches the <u>filesystem server</u>, the server runs locally on the same machine because it uses the STDIO transport. This is commonly referred to as a "local" MCP server. The official <u>Sentry MCP server</u> runs on the Sentry platform, and uses the Streamable HTTP transport. This is commonly referred to as a "remote" MCP server.

Layers

MCP consists of two layers:

- **Data layer**: Defines the JSON-RPC based protocol for client-server communication, including lifecycle management, and core primitives, such as tools, resources, prompts and notifications.
- **Transport layer**: Defines the communication mechanisms and channels that enable data exchange between clients and servers, including transport-specific connection establishment, message framing, and authorization.

Conceptually the data layer is the inner layer, while the transport layer is the outer layer.

Data layer

The data layer implements a <u>JSON-RPC 2.0</u> based exchange protocol that defines the message structure and semantics.

MCP uses JSON-RPC 2.0 as its underlying RPC protocol. Client and servers send requests to each other and respond accordingly. Notifications can be used when no response is required.

Basic JSON-RPC Structure

```
{
  "jsonrpc": "2.0",
  "id": "unique-request-id",
  "method": "method_name",
  "params": {
    // method-specific parameters
  }
}
```

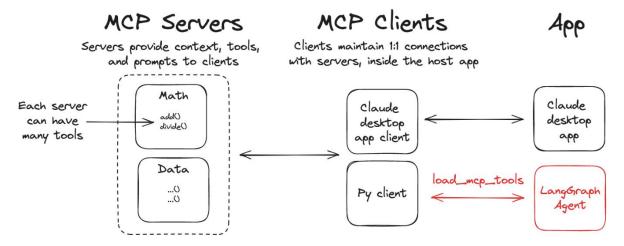


Figure 3 – mcp (client / server / app)

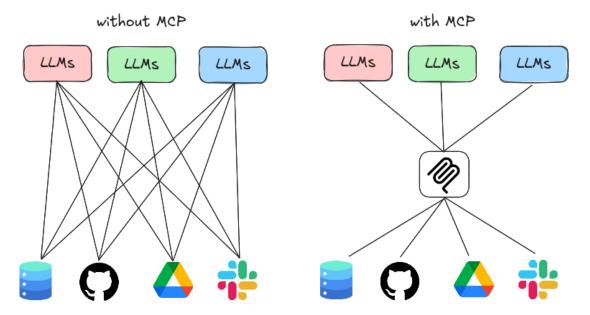


Figure 4 - solve n*m problem to m+n

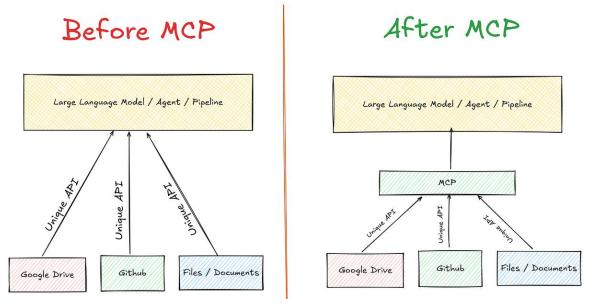
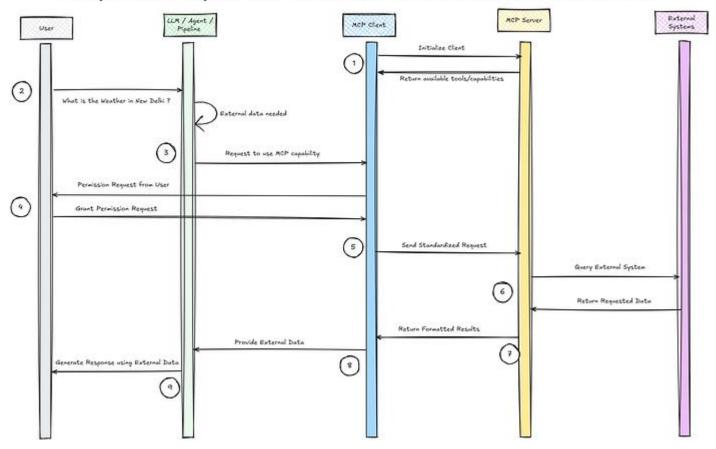


Figure 5 -MCP (BEFORE / AFTER)

Integration of Agent, MCP Client, MCP Server and External Resources



SIMPLE EXAMPLE (not our implementation):

SERVER:

```
from mcp.server.fastmcp import FastMCP
mcp = FastMCP("Calculate Compound Interest Server")
print(f"Starting Server: {mcp.name}")

@mcp.tool()
def compound_interest(
    principal: float, rate: float, years: int, compounds_per_year: int = 1
) -> float:
    """

Calculate compound interest using the formula: A = P(1 + r/n)^(nt)
    Args:
    principal: Initial amount of money
    rate: Annual interest rate (as percentage, e.g., 0.10 for 10%)
    years: Number of years
    compounds_per_year: Number of times interest is compounded per year (default=1)
    """

amount = principal * (1 + rate / compounds_per_year) ** (compounds_per_year * years)
    return (
        principal - round(amount, 2)
    ) * 5

# Just to differentiate with actual compound interest, multiplying by 5.

# So that we can be sure model is not returning its own calculation.

# Do not use this actual project

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

Client:

```
. .
from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client
from openai import AzureOpenAI from dotenv import load_dotenv
server_params = StdioServerParameters(command="python", args=["mcp_server_code.py"])
def llm_client(message: str):
    client = AzureOpenAI(
           azure_endpoint=os.environ["AZURE_ENDPOINT"],
api_key=os.environ["AZURE_OPENAI_API_KEY"],
api_version="2024-12-01-preview",
     response = client.chat.completions.create(
  model=os.environ["GPT_4_1"],
  messages=[
           max_tokens=250,
temperature=0.2,
return response.choices[0].message.content.strip()
def get_prompt():
      You are a helpful assistant with access to the following tools:
     {tools description}
     If no tool is required, respond directly without using any tools.
When you decide to use a tool, respond only with the exact JSON object in the following format, and nothing else:
            "tool": "tool-name",
           "arguments": {{
    "argument-name": "value"
async def run(query: str):
           async with ClientSession(read, write) as session:
    await session.initialize()
                  tools = await session.list_tools()
print(f"Available tools: {tools.tools}")
tools_description = "\n".join(
                              f"- {tool.name}, {tool.description}, {tool.inputSchema} " for tool in tools.tools
                  prompt = prompt.format(query=query, tools_description=tools_description)
llm_response = llm_client(prompt)
                  print(f"LLM Response: {llm_response}")
tool_call = json.loads(llm_response)
result = await session.call_tool(
f'(Compound interest for principal \{tool_call["arguments"]["principal"]\} at rate \{tool_call["arguments"]["rate"]\}% for \{tool_call["arguments"]["years"]\} year(s) is \{tool_call["arguments"]["years"]\}
if __name__ == "__main__":
    query = "Calculate the compound interest for amount 1000 at 10 percent for 1 year"
```

But, ... we aim to expose our FastAPI endpoints as MCP tools without having a separate MCP server.

Exposing Your FastAPI App as an MCP Server

Introduction

You're not just *consuming* Model Context Protocol (MCP) services—you're ready to become an MCP provider. By exposing your FastAPI application as an MCP server, you transform your existing API endpoints into standardized, Al-discoverable tools that any MCP-compatible client (like LLM agents, Al assistants, or orchestration platforms) can automatically integrate with.

This turns your FastAPI app from a traditional REST API into a plug-and-play AI capability that can be:

- Discovered automatically by AI systems
- Invoked programmatically through standardized JSON-RPC calls
- Composed with other MCP tools in complex AI workflows
- Integrated seamlessly into any MCP ecosystem without custom client code

Why Expose FastAPI as MCP?

Traditional REST APIs require manual integration:

Every AI developer must write custom code to understand your endpoints, parameters, authentication, and response formats.

MCP provides automatic interoperability:

Your FastAPI app becomes instantly usable by any MCP client through standardized discovery and invocation protocols.

What This Means for Your Application

When you expose your FastAPI endpoints as MCP tools, you're essentially creating a universal adapter layer that:

- 1. Maps your existing endpoints to MCP's listTools and callTool methods
- 2. Describes your capabilities using standardized JSON schemas
- 3. Handles MCP's JSON-RPC protocol while preserving your existing business logic

4. Maintains your current API while adding MCP compatibility as an additional interface

The Transformation

- Before: Your FastAPI app is a standalone service requiring custom integration
- After: Your FastAPI app is an MCP server that AI systems can discover, understand, and use automatically

This approach is particularly powerful if your FastAPI application provides:

- Domain-specific data retrieval (e.g., customer records, product catalogs)
- Business logic operations (e.g., calculations, validations, workflows)
- External service integrations (e.g., payment processing, notifications)
- Specialized AI/ML capabilities (e.g., custom models, data processing)

```
https://github.com/tadata-org/fastapi_mcp
>> uv add fastapi-mcp (or use pip install)
BASIC USAGE:
from fastapi import FastAPI
from fastapi_mcp import FastApiMCP
app = FastAPI()
mcp = FastApiMCP(app)
# Mount the MCP server directly to your FastAPI app.
mcp.http_mount()
```

That's it! Your auto-generated MCP server is now available at https://app.base.url/mcp

Connecting a client to the MCP server

Once your FastAPI app with MCP integration is running, you would want to connect it to an MCP client.

Connecting to the MCP Server using SSE

For any MCP client supporting SSE, you will simply need to provide the MCP url. All the most popular MCP clients (Claude Desktop, Cursor, ...) use the following config format:

```
{
   "mcpServers": {
     "fastapi-mcp": {
        "url": "http://localhost:8000/mcp"
     }
   }
}
```

If you want to support authentication, or your MCP client does not support SSE, we recommend using mcpremote as a bridge.

```
{
  "mcpServers": {
    "fastapi-mcp": {
        "command": "npx",
        "args": [
             "mcp-mente",
             "http://localhost:8000/mcp",
             "8080" // Optional port number. Necessary if you want your OAuth to work and you don't have dynamic client registration.
        }
    }
}
```

Tool Naming

FastAPI-MCP uses the operation_id from your FastAPI routes as the MCP tool names. When you don't specify an operation id, FastAPI auto-generates one, but these can be cryptic.

```
# Auto-generated operation_id (something like "read_user_users__user_id__get")
@app.get("/users/{user_id}")
async def read_user(user_id: int):
    return {"user_id": user_id}
```

Figure 6 = NO OPERATION ID> AUTO GENERATE NAMING OF THE TOOL

Best practice:

```
# Explicit operation_id (tool will be named "get_user_info")
@app.get("/users/{user_id}", operation_id="get_user_info")
async def read_user(user_id: int):
    return {"user_id": user_id}
```

Figure 7 - PROPER NAMING

Tool and schema descriptions

When creating the MCP server you can include all possible response schemas in tool descriptions by changing the flag describe_all_responses, or include full JSON schema in tool descriptions by changing describe_full_response_schema:

```
mcp = FastApiMCP(
    app,
    name="My API MCP",
    description="Very cool MCP server",
    describe_all_responses=True,
    describe_full_response_schema=True)
```

Customizing Exposed Endpoints

```
mcp = FastApiMCP(
    app,
    include_operations=["get_user", "create_user"]
)
```

The relevant arguments for these configurations are include_operations, exclude_operations, include_tags, exclude_tags.

When using FastApiMCP, it's important to mount the MCP server only after all your FastAPI routes are defined.

ARGS:

This is because MCP introspects the FastAPI application at the moment it is mounted, automatically registering all routes (endpoints) as MCP tools. If you mount MCP too early, any routes defined later will not be included in the MCP tool registry.

MCP works by mapping your FastAPI endpoints to JSON-RPC methods that an LLM can call. If you mount too early, the MCP registry won't know about the routes you add later, and they won't show up as callable tools.

FastAPI + MCP Server Example (our test imp.)

This example demonstrates how to expose your FastAPI routes as **MCP tools** that can be called by an LLM or any MCP-compatible client.

```
FROM FASTAPI IMPORT FASTAPI
FROM PYDANTIC IMPORT BASEMODEL
FROM FASTAPI MCP IMPORT FASTAPIMCP
FROM WEATHER_HELPER IMPORT MAKE_NWS_REQUEST, FORMAT_ALERT
IMPORT ASYNCIO
FROM DEEP_TRANSLATOR IMPORT GOOGLETRANSLATOR
APP = FASTAPI()
# REQUEST BODY SCHEMAS
CLASS SUMREQUEST(BASEMODEL):
  A: INT
  B: INT
CLASS GREETREQUEST(BASEMODEL):
  NAME: STR
CLASS ALERTREQUEST(BASEMODEL):
  STATE: STR
CLASS TRANSLATEREQUEST(BASEMODEL):
  TEXT: STR
 TARGET_LANG: STR # E.G., "ES" FOR SPANISH, "FR" FOR FRENCH
CLASS QUERYREQUEST(BASEMODEL):
  QUERY: STR
NWS API BASE = "HTTPS://API.WEATHER.GOV"
@APP.POST("/ALERTS", OPERATION_ID="GET_WEATHER_ALERTS")
ASYNC DEF GET ALERTS (REQ: ALERTREQUEST):
  """GET WEATHER ALERTS FOR A US STATE.
```

```
Github.com/AliQambari
usend@live.com
    STATE: TWO-LETTER US STATE CODE (E.G. CA, NY)
  URL = F"{NWS API BASE}/ALERTS/ACTIVE/AREA/{REQ.STATE}"
  DATA = AWAIT MAKE_NWS_REQUEST(URL)
 IF NOT DATA OR "FEATURES" NOT IN DATA:
    RETURN "UNABLE TO FETCH ALERTS OR NO ALERTS FOUND."
  IF NOT DATA["FEATURES"]:
    RETURN "NO ACTIVE ALERTS FOR THIS STATE."
  ALERTS = [FORMAT ALERT(FEATURE) FOR FEATURE IN DATA["FEATURES"]]
  RETURN {"ALERTS": "\N---\N".JOIN(ALERTS)}
@APP.POST("/SUM",OPERATION ID="SUM NUMBERS")
ASYNC DEF SUM NUMBERS (REQ: SUMREQUEST):
  RETURNS THE SUM OF TWO NUMBERS.
  PARAMETERS:
  REQ (SUMREQUEST): REQUEST BODY CONTAINING TWO NUMBERS TO BE ADDED.
  RETURNS:
    DICT: RESPONSE CONTAINING THE SUM OF THE TWO NUMBERS.
  RETURN {"SUM": REQ.A + REQ.B}
@APP.POST("/GREET", OPERATION ID="GREET USER")
ASYNC DEF GREET (REQ: GREET REQUEST):
  RETURNS A GREETING MESSAGE FOR A GIVEN NAME.
  PARAMETERS:
  REQ (GREETREQUEST): REQUEST BODY CONTAINING THE NAME TO BE GREETED.
  RETURNS:
    DICT: RESPONSE CONTAINING THE GREETING MESSAGE.
  RETURN {"MESSAGE": F"HELLO, {REQ.NAME}!"}
@APP.POST("/TRANSLATE", OPERATION ID="TRANSLATE TEXT") #ADD OPERATION ID FOR THE MCP
ASYNC DEF TRANSLATE_TEXT(REQ: TRANSLATEREQUEST):
  TRANSLATE TEXT FROM ONE LANGUAGE TO ANOTHER.
  PARAMETERS:
  REQ (TRANSLATEREQUEST): REQUEST BODY CONTAINING THE TEXT TO BE TRANSLATED AND THE TARGET LANGUAGE.
```

```
RETURNS:
    DICT: RESPONSE CONTAINING THE TRANSLATED TEXT, THE ORIGINAL TEXT, AND THE TARGET LANGUAGE.
  LOOP = ASYNCIO.GET_RUNNING_LOOP()
  #WRAP SYNCHRONOUS TRANSLATOR IN THREAD
  TRANSLATED_TEXT = AWAIT LOOP.RUN_IN_EXECUTOR(
    None, LAMBDA: GOOGLETRANSLATOR(SOURCE='AUTO', TARGET=RQ.TARGET LANG).TRANSLATE(REQ.TEXT)
  )
  RETURN {
    "ORIGINAL": REQ.TEXT,
    "TRANSLATED": TRANSLATED_TEXT,
    "TARGET LANG": REQ.TARGET LANG
  }
#CREATE MCP AFTER ROUTES EXIST
MCP = FASTAPIMCP(
  APP,
  NAME="FASTAPI-MCP",
  DESCRIPTION="MCP SERVER",
  DESCRIBE_ALL_RESPONSES=TRUE,
  DESCRIBE_FULL_RESPONSE_SCHEMA=TRUE
MCP.MOUNT_HTTP()
# REFRESH THE MCP SERVER TO INCLUDE THE NEW ENDPOINT
MCP.SETUP_SERVER()
IF __NAME__ == "__MAIN__":
  IMPORT UVICORN
  UVICORN.RUN(APP, HOST="127.0.0.1", PORT=8000)
```

Standard FastAPI routes are available (e.g. /sum, /greet, /translate, ...)

The **MCP endpoint** is exposed at:

http://127.0.0.1:8000/mcp

Using the FastAPI MCP Server (APP+ MCP SERVER) in Claude Desktop (or any MCP client)

Once you've defined your routes and mounted the MCP server with FastApiMCP, you can expose it to any MCP-compatible client such as Claude Desktop, VS Code (future plugins), or other apps that speak the MCP protocol.

For claude desktop app:

Edit or create:

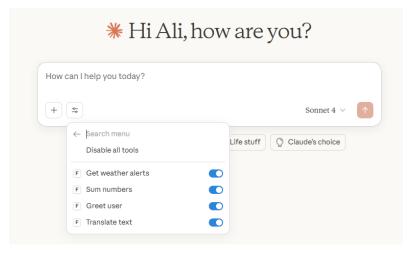
%APPDATA%\Claude\claude_desktop_config.json

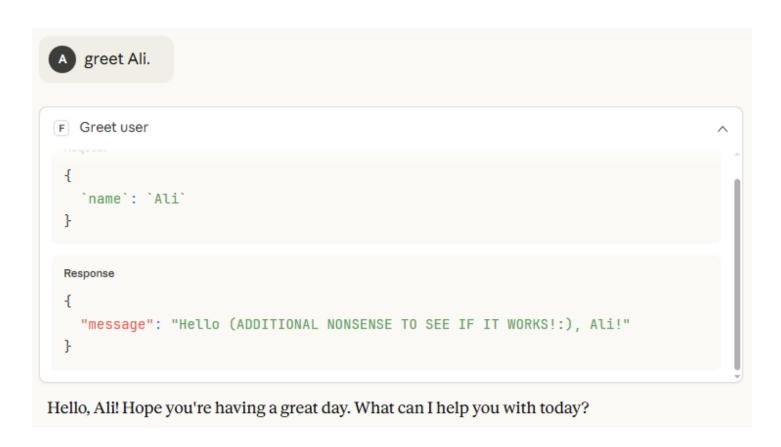
Add:

```
{
  "mcpServers": {
    "fastapi-mcp": {
        "url": "http://localhost:8000/mcp"
    }
}
```

The abovementioned script does not work while mentioned in the documentation. Use the following instead:

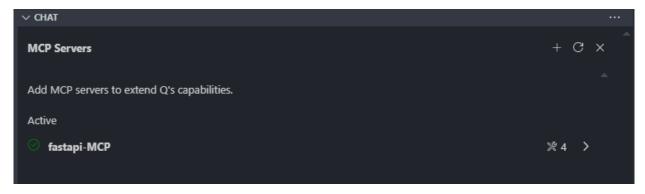
Then restart the claude app (while your fastapi app is running). You'll see:

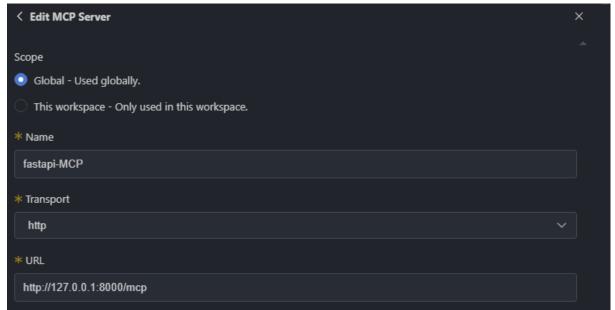




To make sure Claude (or another MCP client) is truly calling my MCP server and not "hallucinating" a response, I deliberately added a unique marker string to one of my tool responses. (DEPICTED IN THEB FIGURE ABOVE).

Also, our server can be used in any MCP compatible plugin in vscode.







```
Parameters

{    "text": "HOW ARE YOU DOING?",
    "target_lang": "es"

Result

{    "output": {
        "kind": "text",
        "content": "{\"content\":[{\"type\":\"text\",\"text\":\"{}
    }

Perfect! The translation is: "¿CÓMO ESTÁS?"

Your MCP translate tool successfully translated "HOW ARE YOU DOING?" from English to Spanish.
```

ADDITIONAL INFO:

PYPROJECT.TOML

```
[project]
name = "mc"
version = "0.1.0"

description = "Add your description here"
readme = "README.md"
requires-python = ">=3.13"
dependencies = [
   "deep-translator>=1.11.4",
   "fastapi-mcp>=0.4.0",
   "fastapi[standard]>=0.117.1",
   "googletrans>=4.0.2",
   "httpx>=0.28.1",
   "openai>=1.109.1",
   "sseclient>=0.0.27",
   "uvicorn>=0.37.0",
]
```

Needless to say, our fastapi app also works at: 127.0.0.1:8000

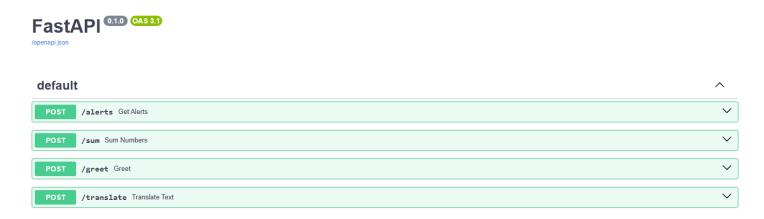


Figure 8 - FASTAPI ENDPOINTS STILL WORKING PROPERLY

Test in POSTMAN USING MCP PROTOCOL:

- WE CAN SEE ALL TOOLS LISTED.
- ALL TOOLS CAN BE CALLED DIRECTLY.

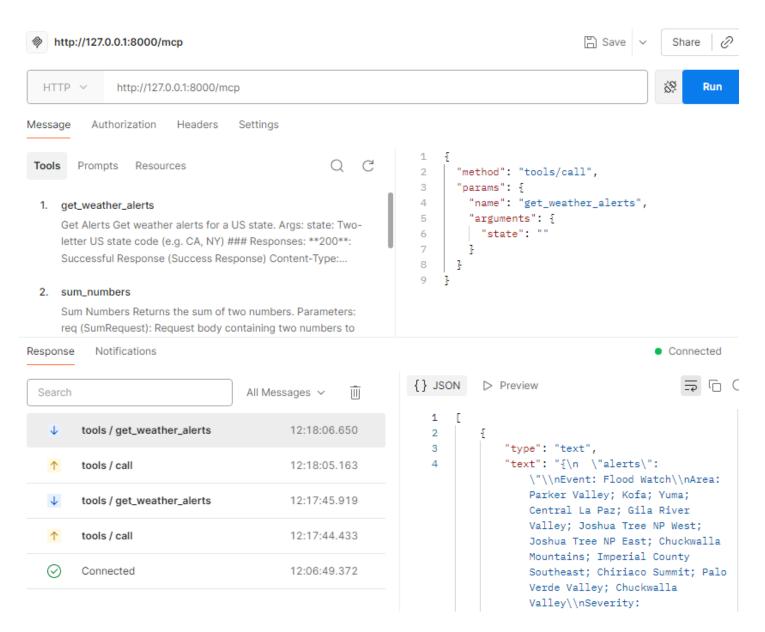


Figure 9 - TOOL CALL BASED ON MCP

FAQ

When should you add MCP?

Only if you want external systems to use your tools, such as:

Your own agent inside FastAPI uses tools	X No
You want Cursor / VS Code AI to use your tools	✓ Yes
Another team's LLM agent (in a different service) wants to call your tools	✓ Yes
You're building a tool market place or shared tool server	✓ Yes

What You Don't Need to Do

- You don't need to rewrite your entire agent in MCP.
- You don't need to use MCP for internal tool calls (your own agent can still call the functions directly).
- You don't need to send raw JSON-RPC yourself the MCP library handles that.

6 Key Insight

MCP is an interface, not an architecture.

You keep your agent logic as-is, but expose your tools through MCP so *other* agents (or IDEs, or LLM clients) can use them too.

Think of it like adding a universal plug to your tools — your own device uses them internally, but now anyone with the right socket can plug in.

WHAT IF I WANT TO DEFILE MY TOOLS BASED ON MCP PROTOCOL THEN AGENTS CAN USE TOOLS USINF JRPC:

Ah! Now I fully understand what you're asking — and it's a great design choice.

- You want to build your *own* agentic system where:
 - You run your own MCP server (with your tools)

• Your own agents (inside your FastAPI app or elsewhere) use those tools via MCP — not by calling functions directly, but by speaking the MCP protocol (JSON-RPC over stdio, HTTP, etc.)

This means:

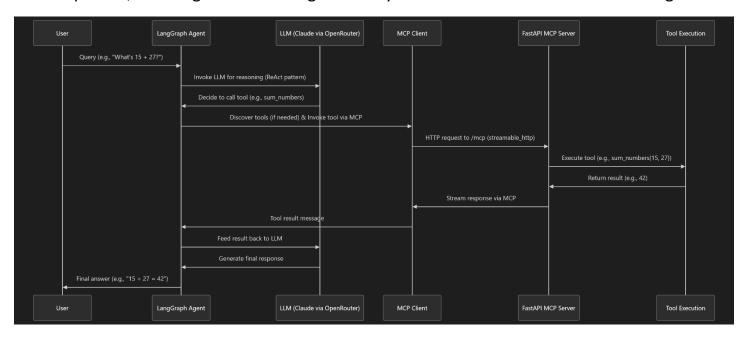
- ♦ Your agent acts as an MCP client
- ♦ Your tool server acts as an MCP server
- ♦ They communicate strictly over MCP even though they're in the same system

This gives you:

- Clean separation of concerns
- Easier testing & debugging
- Future compatibility (e.g., swap agents or tools independently)
- Full compliance with the MCP ecosystem

Still another test using Langgraph agent:

In this setup, our LangGraph ReAct agent (built with create_react_agent) dynamically discovers and invokes tools from your FastAPI-exposed MCP server at http://localhost:8000/mcp via MultiServerMCPClient. The agent uses the LLM (via OpenRouter) to reason about user queries, decide on tool calls (e.g., sum_numbers for math, get_weather_alerts for alerts), and execute them through the MCP transport. Results from the server are fed back into the agent's state for final responses, enabling seamless integration of your custom tools without hardcoding.



```
import asyncio
import os
from dotenv import load_dotenv
from langchain_mcp_adapters.client import MultiServerMCPClient
from langgraph.prebuilt import create_react_agent
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMessage, ToolMessage
load_dotenv()
async def main():
    client = MultiServerMCPClient({
         "my_fastapi_mcp": {
             "url": "http://localhost:8000/mcp",
"transport": "streamable_http"
    tools = await client.get_tools()
    print(f"Discovered {len(tools)} tools: {[t.name for t in tools]}")
    llm = ChatOpenAI(
         temperature=0,
api_key=os.getenv("OPENROUTER_API_KEY"),
         base_url="https://openrouter.ai/api/v1",
    agent = create_react_agent(llm, tools)
    test_queries = [
         "Tell me the weather in NY and greet Ali"
         print(f"\n{'='*60}")
        print(f"User: {query}")
print("-" * 60)
         response = await agent.ainvoke({
              "messages": [HumanMessage(content=query)]
         for msg in response["messages"]:
             if isinstance(msg, HumanMessage):
    print(f"User: {msg.content}")
             elif isinstance(msg, AIMessage):
                  print(f"Assistant: {msg.content}")
                      for tool_call in msg.tool_calls:
    print(f" Tool Call: {tool_call['name']} with args
{tool_call['args']}")
             elif isinstance(msg, ToolMessage):
                  print(f"Tool Result: {msg.content}")
         final_message = response["messages"][-1]
         print(f"\nFinal Answer: {final_message.content}")
    asyncio.run(main())
```

Github.com/AliQambari usend@live.com Sample answer: Discovered 4 tools: ['get_weather_alerts', 'sum_numbers', 'greet_user', 'translate_text'] _____ User: What's 15 + 27? Full Agent Response: User: What's 15 + 27? Assistant: Tool Call: sum_numbers with args {'a': 15, 'b': 27} Tool Result: { "sum": 42 Assistant: 42 Final Answer: 42 ______ User: Say hello to Ali -----Full Agent Response:

User: Say hello to Ali

Assistant:

Tool Call: greet user with args {'name': 'Ali'}

Tool Result: {

"message": "Hello (ADDITIONAL NONSENSE TO SEE IF IT WORKS!:), Ali!"

}

Assistant: Hello, Ali!

Final Answer: Hello, Ali!