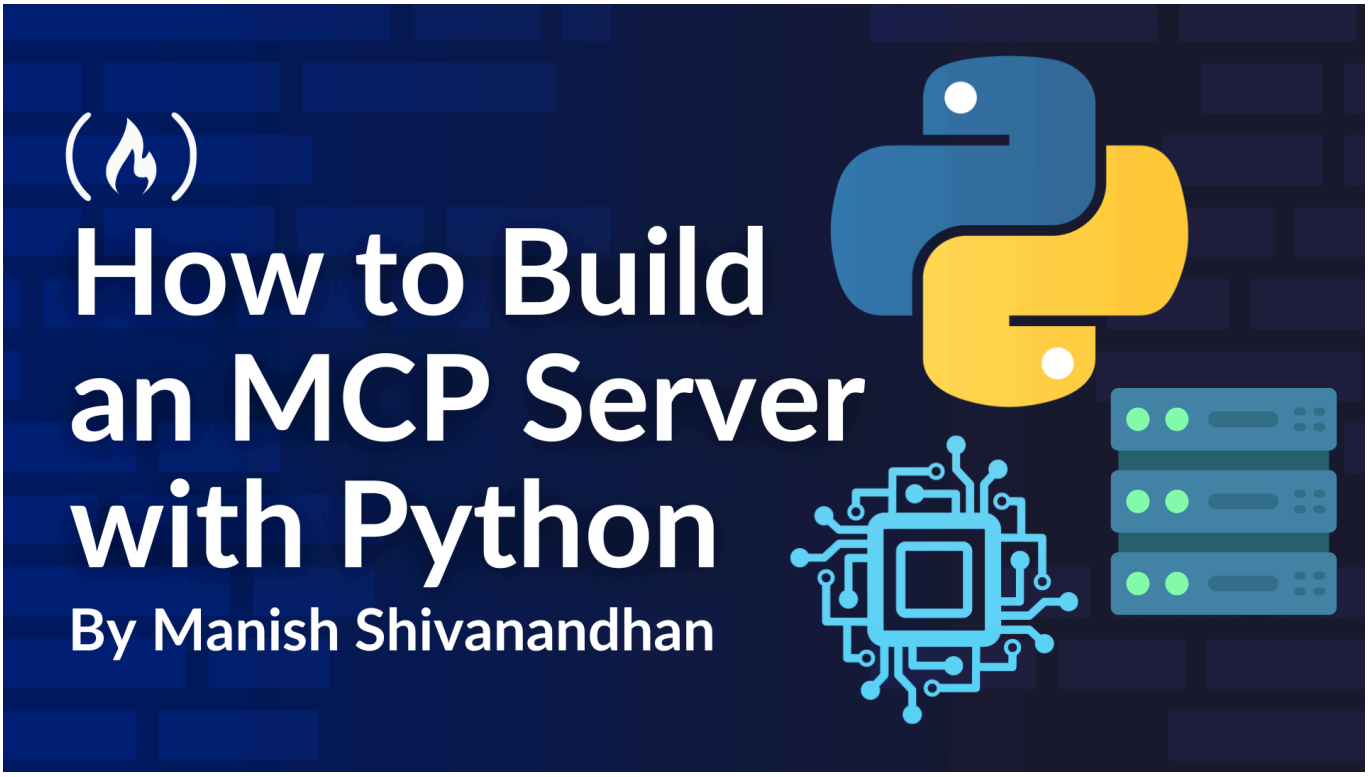
OCTOBER 30, 2025 / #PYTHON

How to Build Your Own MCP Server with Python

Manish Shivanandhan



Models today can reason, write, code, and analyze information in ways that once seemed impossible.

But there's one major limitation that still holds them back: context.

Most AI models don't have access to your system, files, APIs, or live data. They only know what you tell them in a prompt.

The Model Context Protocol, also known as MCP, was created to address this problem. It enables AI models to securely connect to your own tools, APIs, and systems via small, structured servers known as MCP servers.

In this guide, you'll learn how to build your own MCP server using Python. We'll walk through each part of the code and I'll explain how it works.

By the end, you'll have a running MCP server that can add numbers, return random words, and fetch live weather data from the internet. We will also see how to host this MCP server on the cloud.

What we'll cover:

What is Model Context Protocol?

Before diving into the code, it's important to understand what the Model Context Protocol actually is.

MCP is an open standard that defines how AI models and external systems communicate. You can think of it as an API that's designed specifically for AI assistants.

If an API lets two software programs exchange data, MCP allows an AI model to talk to your system. This opens up endless possibilities.

You could build an MCP server that lets ChatGPT read files from your local machine, or one that calls your company's internal APIs to fetch data. You could even expose your own Python functions so that a model can use them as tools.

Setting Up Your Environment

To follow along, you'll need Python version 3.9 or higher. You can find the code for this example in this repository.

We'll use a library called FastMCP that simplifies the process of building MCP servers. You can install it using pip:

```
pip install fastmcp requests
```

The `requests` library will be used to make HTTP calls later in the example. Once installed, you're ready to create your first MCP server.

Creating the Project

Create a new file called `server.py` and start by importing the necessary modules:

```python
import logging
import os
import random
import sys
import requests
from mcp.server.fastmcp import FastMCP
```

- `os` is used to access environment variables like port numbers.
- `random` will help us generate random words.
- `sys` allows the script to exit gracefully in case of errors.
- `requests` lets us fetch live data from APIs.
- And finally, `FastMCP` turns our Python functions into tools that can be called through the MCP protocol.

Configuring Logging

Logging gives you visibility into what your server is doing. It helps during development and is vital when you deploy your server in production.

```
name = "demo-mcp-server"
logging.basicConfig(
    level=logging.INFO,
    format='%(name)s - %(levelname)s - %(message)s',
    handlers=[logging.StreamHandler()]
)
logger = logging.getLogger(name)
```

This configuration prints log messages to the console in a simple format showing the server name, the log level, and the message. Every time a tool runs, a message will appear in the logs such as:

Creating the MCP Server

Next, we'll create the server instance that will host our tools.

```
port = int(os.environ.get('PORT', 8080))
mcp = FastMCP(name, logger=logger, port=port)
```

The server will run on the port specified by the environment variable `PORT`. If that variable isn't set, it defaults to 8080. The `FastMCP` object now represents your running MCP server.

Defining Tools

Each function that you decorate with `@mcp.tool()` becomes an accessible tool that clients can call. Let's start with a simple example: an addition tool.

Example 1: Adding Two Numbers

```
@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    logger.info(f"Tool called: add({a}, {b})")
    return a + b
```

Even though it's simple, this shows the basic structure of every MCP tool: input parameters, a logging statement, and a return value.

Example 2: Returning a Random Secret Word

Let's make another tool that returns a random word from a small list.

```
@mcp.tool()
def get_secret_word() -> str:
    """Get a random secret word"""
    logger.info("Tool called: get_secret_word()")
    return random.choice(["apple", "banana", "cherry"])
```

When you call this function, it picks one of the three words at random. Each time you call it, you might get a different result. This function demonstrates how MCP tools can use logic or randomness just like any regular Python function.

Example 3: Fetching Weather Data

Now let's build something more practical. We'll create a tool that fetches live weather data from the web using the `requests` library.

```
@mcp.tool()
def get_current_weather(city: str) -> str:
    """Get current weather for a city"""
    logger.info(f"Tool called: get_current_weather({city})")
```

```
        return response.text
    except requests.RequestException as e:
        logger.error(f"Error fetching weather data: {str(e)}")
        return f"Error fetching weather data: {str(e)}"
```

This tool accepts a city name, sends a request to the public weather service at `wttr.in`, and returns the text-based weather report. If there's any issue, such as a network timeout or invalid city name, the function logs an error and returns a descriptive message.

Calling `get_current_weather("London")` will print a short weather summary for that city.

Running the Server

Once all your tools are defined, you can start the server. Add the following code to the bottom of your file:

```
if __name__ == "__main__":
    logger.info(f"Starting MCP Server on port {port}...")
    try:
        mcp.run(transport="sse")
    except Exception as e:
        logger.error(f"Server error: {str(e)}")
        sys.exit(1)
    finally:
        logger.info("Server terminated")
```

This block starts the server using the Server-Sent Events transport method. If anything goes wrong, it logs the error and shuts down

```
python server.py
```

If everything is working, you'll see:

```
demo-mcp-server - INFO - Starting MCP Server on port 8080...
```

Your MCP server is now live and ready to accept requests.

Testing the Tools

To test your tools, you need an MCP-compatible client such as ChatGPT with developer features or another app that supports the protocol. Once connected, the client will list your available tools.

For example, you can send a request like this:

```
{
  "tool": "add",
  "args": [5, 7]
}
```

The server will respond with:

```
}
```

The same applies to the other tools such as `get_secret_word` or `get_current_weather`.

If you want to test the server directly without the MCP client, you can still send HTTP requests manually (though this bypasses the full protocol logic).

For example, to test your weather tool, you can send a simple GET request:

```
curl http://localhost:8080/tool/get_current_weather?city=London
```

or in Python:

```
import requests
response = requests.get("http://localhost:8080/tool/get_current_weather",
print(response.text)
```

This won't use the MCP structure (like `sse` streaming), but it's a quick sanity check that your server works.

You can run this server locally for development. But if you want to use it in production applications, you have to deploy it to a server.

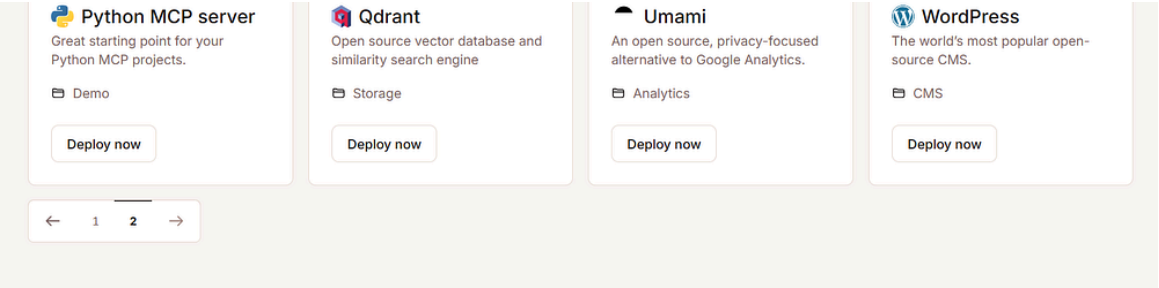
You can choose any cloud provider, like AWS, Heroku, or others to set up this project. But I will be using Sevalla.

Sevalla is a modern, usage-based Platform-as-a-service provider. It offers application hosting, database, object storage, and static site hosting for your projects.

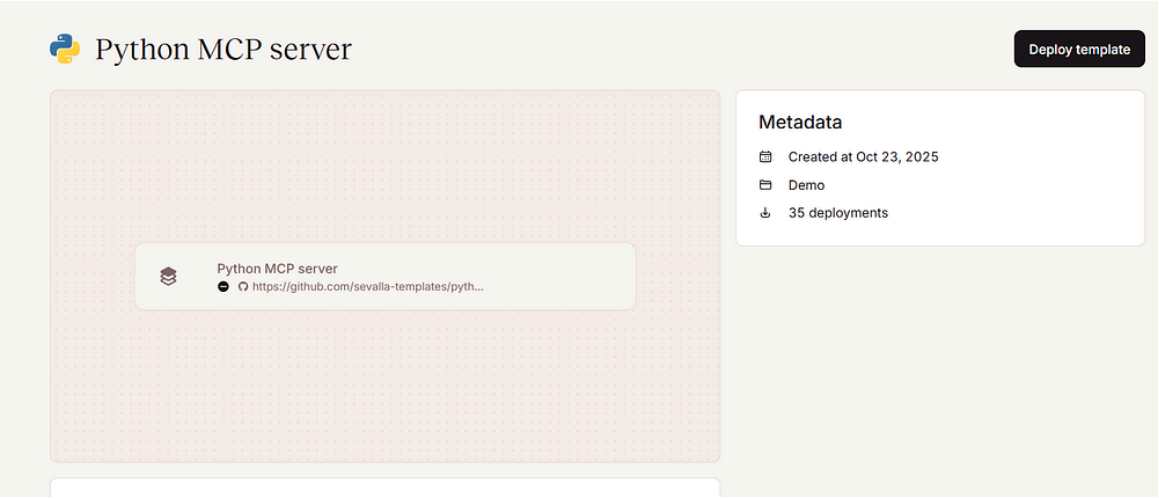
I am using Sevalla for hosting for two reasons:

- Every platform will charge you for creating a cloud resource. Sevalla comes with a $50 credit for us to use, so we won't incur any costs for this example.
- Sevalla has a template for Python MCP server, so it simplifies the manual installation and setup for each resource you will need for installation.

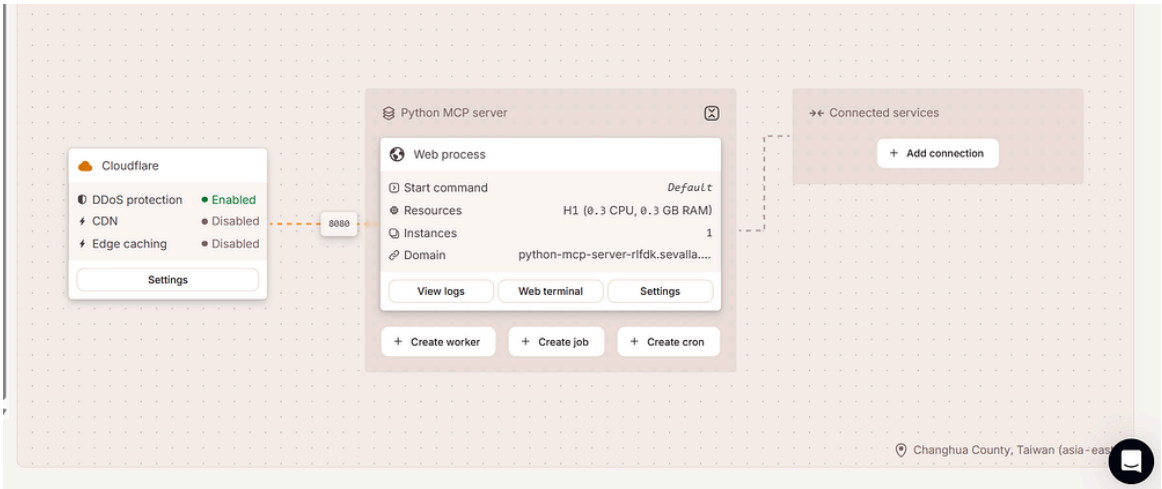
Log in to Sevalla and click on Templates. You can see Python MCP Server as one of the templates.



Click on the “Python MCP Server” template. You will see the resources needed to provision the application. Click on “Deploy Template”.



You can see the resource being provisioned. If the deployment doesn't start automatically, click “Deploy now”.



Wait for a few minutes. Once the deployment is complete, you will see a green checkmark.

Once deployment is complete, click on “Visit app”. You will get a cloud url eg. https://python-mcp-server-rlfdk.sevalla.app. Use this as the base url instead of the localhost:3000 url.

applications.

Why Build Your Own MCP Server?

Building an MCP server gives you control and flexibility.

You can connect AI models directly to your databases or internal systems, automate repetitive actions, and decide exactly what data an AI model can access.

It also allows you to experiment quickly. You can start small with a few simple tools and expand later into complex workflows.

By creating your own MCP server, you're not just writing code – you're defining how intelligent systems interact with the real world through your logic and data.

Expanding the Server

Once you've mastered the basics, it's easy to extend your server. You can add tools that read and write files, query databases, interact with APIs like GitHub or Slack, or monitor your system. Each new function becomes another tool that your AI can use.

This modular approach lets you build an entire ecosystem of AI-aware tools, each performing a specific task but working together through the same MCP interface.

Conclusion

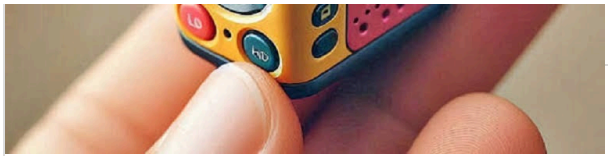
how easily these tools can expose real functionality, like fetching live weather data or performing basic computations.

This structure is simple yet powerful. With just a few lines of Python code, you can build bridges between your systems and intelligent models. The Model Context Protocol represents a step toward AI systems that can truly understand and interact with real-world data and actions.

*Hope you enjoyed this article. Signup for my free newsletter **TuringTalks.ai** for more hands-on tutorials on AI. You can also **visit my website**.*

Manish Shivanandhan

Read more posts.



Smart

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Our Charity