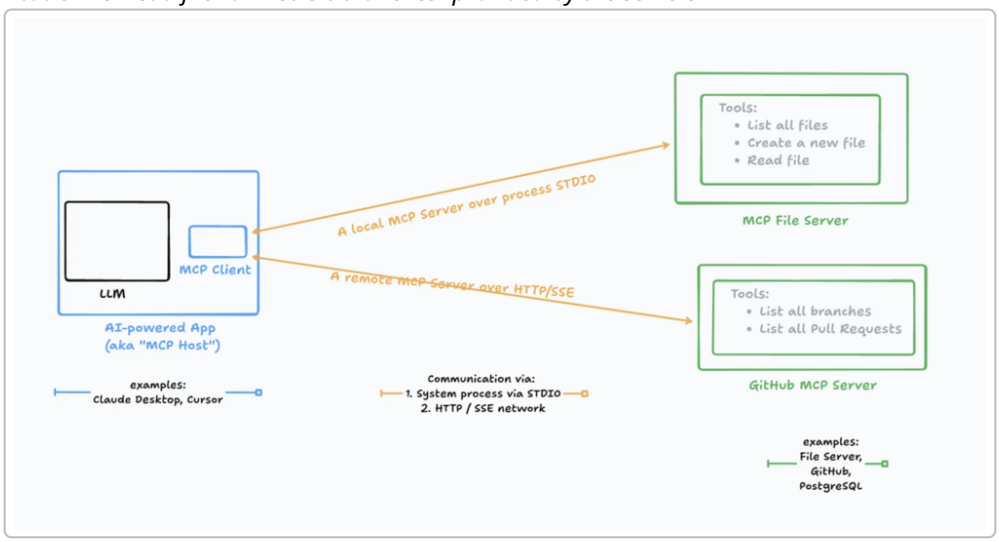
ChatGPT

Building an MCP Server from Scratch in Python

Developers can leverage the **Model Context Protocol (MCP)** to expose data and functionality to large language model (LLM) applications in a standardized way. Think of MCP as a "USB-C port" for AI – a universal connector that lets LLMs plug into external tools, APIs, and data sources [1] [2]. In MCP's architecture, an **LLM host application** (e.g. Claude Desktop, VS Code, or a custom agent) runs an **MCP client** internally, which connects to one or more **MCP servers** that provide the actual tools and context [3] [4]. Each MCP server can offer:

- **Resources:** data that can be read (like files or API responses)
- **Tools:** functions that can be executed to perform actions or computations
- **Prompts:** pre-defined prompt templates to guide interactions

Figure 1: *MCP architecture – an AI application (host + client) connecting to MCP servers. One server is running locally via STDIO (as a subprocess), and another is accessed remotely over HTTP/SSE. This design allows LLMs to list and call tools like "read file" or "list Git branches" provided by the servers*



.

MCP Architecture and Transport Mechanisms

MCP follows a client–server model. The LLM host's MCP **client** initiates a connection to an MCP **server**, then exchanges messages using JSON-RPC over a selected transport [5]. The MCP specification currently defines two primary transports [6] [5]:

1. **STDIO (Standard I/O):** The client launches the server as a local subprocess. All communication happens via the process's stdin/stdout streams [7]. This is essentially a **local** transport – the server doesn't listen on a network port; instead it reads JSON-RPC requests from stdin and writes responses to stdout. STDIO is simple and fast for local tools, and MCP clients **should** support it when possible

[8]. For example, an IDE plugin can spawn an MCP server (like a filesystem tool) on the fly and talk to it via pipes.

2. **HTTP (Streamable HTTP / SSE):** The server runs as an independent service and accepts network connections over HTTP. Originally, MCP servers used HTTP + SSE (Server-Sent Events) for streaming [5] [9]. Newer implementations use **Streamable HTTP**, which combines request/response and streaming in one endpoint [10] [11]. In either case, the client connects via a URL (e.g. `http://<server_host>/mcp` or `/sse`), and JSON-RPC messages are exchanged over HTTP. This transport allows **remote** connectivity – the client and server can be on different machines, communicating over the internet.

In practice, both HTTP transports function similarly: the client sends JSON-RPC requests in HTTP POST bodies, and the server responds either with a single JSON response or by keeping the connection open to stream multiple messages (using SSE events) [12] [13]. The server also supports an HTTP GET on the same endpoint to establish a pure event stream (for server-initiated messages) [14]. The newer **Streamable HTTP** spec consolidates these into one `/mcp` endpoint handling both POST and GET, improving compatibility with serverless platforms [15].

WebSocket Support: Not officially part of the MCP spec, but developers can implement custom transports [16]. WebSockets provide full-duplex communication, which can simplify client–server messaging. In fact, community projects have created WebSocket bridges for MCP. For example, **mcp-server-runner** is a tool that launches an MCP server process and relays messages over a WebSocket, acting as a bridge [17]. Using such a bridge, a web app could connect via `ws://your-server:PORT` and exchange MCP JSON-RPC messages in real time. Keep in mind that these custom solutions may have limitations – e.g. the example runner supports only one client and has no built-in TLS or auth [18] [19]. If you choose to roll out your own WebSocket layer, you'll need to handle authentication and encryption (e.g. by running behind a wss proxy) and ensure proper routing of JSON-RPC messages to the MCP server.

Setting Up the Python MCP Server Environment

To build an MCP server from scratch using Python, we'll use the official **MCP Python SDK** (which includes a high-level framework called **FastMCP**). Below are the steps to get started:

1. **Install the MCP SDK:** You can install the library via pip. It's recommended to include the `[cli]` extra for development tools. For example:

```
pip install "mcp[cli]"
```

This will install the `mcp` package and its CLI. (If you prefer, you can use the Astral `uv` tool as shown in official docs [20] [21], but using pip directly is straightforward.)

1. **Create a Python script for your server:** Choose a project directory and create, for example, a file `server.py`. In this script, import FastMCP and instantiate a server:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("My API Wrapper")  # Give your server a name
```

This `FastMCP` object represents your MCP server. Next, define the tools, resources, or prompts you want to expose. You do this by decorating Python functions with `@mcp.tool()`, `@mcp.resource()`, or `@mcp.prompt()`. For instance, if we want to wrap some external API and a local function:

```
import requests

@mcp.tool()
def get_time(city: str) -> str:
    """Get current time in the specified city (via WorldTime API)"""
    resp = requests.get(f"http://worldtimeapi.org/api/timezone/{city}")
    data = resp.json()
    return data.get("datetime", "Unknown")

@mcp.tool()
def square(num: int) -> int:
    """Square a number (example of a local computation)"""
    return num * num
```

Each tool function should have type-annotated parameters and a return type. The docstring serves as the tool's description. MCP uses these to automatically generate a schema that the LLM can see [1]. In our example, we defined two tools: one that calls an external service (HTTP request) to get the time, and another that performs a simple calculation. You could similarly wrap database queries, file system operations, or calls to any Python library. The logic inside the function can be as simple or complex as needed – the MCP server just needs to return serializable results (like str, int, dict, etc.).

1. **Implement any needed resources or prompts:** Tools are like "actions" (they may have side effects or do computations), whereas **resources** are read-only data endpoints. For example, if you wanted to expose a configuration or a static dataset to the LLM, you could do:

```
@mcp.resource("config://app")
def get_config() -> str:
    """Application config in JSON"""
    return open("config.json").read()
```

A resource is identified by a URI (here we use a custom scheme `config://`). Clients can request it by that URI [22]. **Prompts** are predefined prompts or conversation templates. For instance, you might define a

prompt that the LLM should use for a certain tool usage pattern. Prompts aren't mandatory – the main focus in most MCP servers is on tools and resources.

1. **Finalize server startup code:** At the end of your script, add a conditional to run the server when the script is executed:

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

By default, `mcp.run()` will launch the server in STDIO mode (the default transport) [23]. This is great for local development or if you intend to use the server only as a subprocess. However, since we want to support remote connections (over the network), we'll need to adjust the transport and provide network parameters.

Running the MCP Server – STDIO vs Network Mode

Running in STDIO mode (Local): If you keep `mcp.run()` with no arguments (or explicitly `transport="stdio"` [24]), the server will wait on stdin/stdout for a client. You typically do **not** manually run a standalone STDIO server for remote use; instead, a host application (like an agent) would spawn the `server.py` process itself. For example, OpenAI's Agent SDK can spawn an MCP server by command (`MCPServerStdio` with a given `command` and `args`) [25]. STDIO servers are useful in that scenario, as no networking is needed – the client and server just pipe data. If you test your server locally using the CLI (e.g. `mcp dev server.py`), it will also use STDIO under the hood to interact with your server process.

Running in network mode (HTTP/SSE): To allow clients on other machines to connect, run the server with an HTTP-based transport. The Python SDK supports both the legacy SSE transport and the new streamable HTTP transport. You can enable these by passing `transport="sse"` or `transport="streamable-http"` to `mcp.run()`, along with a host and port. For example:

```
if __name__ == "__main__":
    mcp.run(
        transport="streamable-http",
        host="0.0.0.0",
        port=8000,
        path="/mcp"
# endpoint path (optional; default is "/mcp" for streamable HTTP)
    )
```

This will start an HTTP server bound to all interfaces (`0.0.0.0`) on port 8000, with the MCP endpoint at `/mcp` [26] [27]. Now your MCP server is a web service. An LLM client (for example, Claude Desktop or a custom agent) can connect by pointing to `http://<YOUR-IP>:8000/mcp`. Under the hood, the MCP SDK will use an HTTP client to POST requests and receive events. The code above uses the streamable HTTP transport, which is generally recommended for new deployments (it supports both request/response and streaming in one endpoint). If you need the SSE transport specifically (for compatibility with certain older

clients), you could do `transport="sse"` and perhaps `path="/sse"` (default for SSE is `/sse`) [28] – the usage is similar. In SSE mode, the client might open `http://<YOUR-IP>:8000/sse` for a streaming connection [29].

Multi-Client and Session Handling: An HTTP-based MCP server can handle multiple clients connecting concurrently, each maintaining its own session state if needed. FastMCP supports both stateless mode and stateful sessions. By default, streamable HTTP runs in stateful mode, assigning a session ID when the client sends an Initialize request [30]. You don't need to manage this manually – the SDK handles session IDs and routing of messages to the correct session. Just be aware that if your tools have side effects or stored context, each client connection may have isolated state (unless you design the server to be stateless). If you prefer stateless operation (each request independent), FastMCP allows `FastMCP(..., stateless_http=True)` when creating the server [31].

Networking Configuration for Remote Access

Running the server on `0.0.0.0` with a chosen port (e.g. 8000) is the first step to make it accessible beyond your local machine. Next, consider the following to enable access **over the internet** (for example, if your server is on a home or office network and you want to connect from elsewhere):

- **Port Forwarding:** If the server is behind a router or firewall, you must forward the chosen port to your machine. For instance, log into your router and forward port 8000 (or whichever you used) to the internal IP address of the machine running the MCP server. This allows external traffic to reach the server. On a cloud VM or VPS, ensure any cloud firewall or security group allows inbound traffic on that port.

- **IP or Domain:** Determine the public IP address of your server's network (e.g. use a service like `ifconfig.me`). You can use this IP to connect. A better approach is to use a DNS name – for example, a dynamic DNS service or a subdomain pointing to your IP. Clients will use `http://<your-domain>:8000/mcp` (or the appropriate path) to connect. If you own a domain, you can create an A record for a subdomain pointing to your IP. If your ISP gives a dynamic IP, dynamic DNS services can help keep it updated.

- **Local Network vs Internet:** If client and server are on the same LAN, you can use the server's LAN IP (e.g. 192.168.x.x). No port forwarding is needed in that case (just ensure the machine's firewall allows it). For access across the internet, however, the public IP/port is needed as above.

- **Operating System Firewall:** On Windows 10, for example, when you first run the script, Windows may prompt to allow Python through the firewall. Choose to allow it on appropriate networks. On Ubuntu or other Linux, if ufw or firewalld is active, open the port (`sudo ufw allow 8000`).

Once these steps are done, your server should be reachable from outside. For instance, if your machine's public IP is `203.0.113.45` and you forwarded port 8000, a remote client could connect to `http://203.0.113.45:8000/mcp`. You can test connectivity by accessing that URL in a browser – it won't show a webpage (since the MCP endpoint expects JSON/SSE), but you should at least get an HTTP 405 or 404 response instead of no response. That indicates the port is open. (If you see nothing or it times out, the networking isn't configured correctly.)

Note: By default, the MCP spec advises servers to bind to localhost for security [32]. We overrode this to `0.0.0.0` for remote access, but be aware that this exposes the server to any incoming connection. You should **only** do this in a controlled environment or if you implement security, as discussed next.

Authentication and Security Considerations

Authentication: The simplest working setup might be *no auth at all*, relying on security through obscurity or network isolation. However, **this is not recommended for an internet-exposed service.** Without authentication, anyone who finds the URL could call your MCP tools. Even if your tools seem harmless (e.g. checking weather), consider that an attacker could abuse them or gain information. Moreover, if you later add tools that modify data or control devices, an open endpoint would be a serious risk. The MCP specification urges implementing proper auth for any network-accessible server [32]. The official SDK has built-in support for OAuth2 flows (the MCP client can obtain tokens and include them) [33], suggesting a robust way to secure servers. But OAuth2 can be complex to set up just for a personal project.

For a **simple authentication** approach, you have a few options:

- **Static API Key:** Define a shared secret token that the client must provide. For example, you could require a special header in every request (like `X-API-Key: <your-secret>`). In your server code, you'd need to intercept requests to check this header. Since FastMCP runs on Starlette (an ASGI framework) internally, one way is to mount it inside a small FastAPI app and use a dependency or middleware to check headers. For example, create a FastAPI instance, add a middleware that rejects requests without the correct header, then `app.mount("/mcp", mcp.streamable_http_app())`. This way, before MCP handles the request, your middleware can enforce the API key. This approach is relatively easy and works if you can modify client requests (your custom client or agent should send the header). Keep the token long and random, and **never hard-code it in any client that could be public**.

- **HTTP Basic Auth or Bearer Token:** If you put your MCP server behind a reverse proxy (like Nginx or Caddy), you could use basic auth or a static bearer token. For example, require a username/password or an Authorization header for the `/mcp` path. Reverse proxies can handle TLS and auth, then forward authorized requests to your MCP server. This adds a layer of security without modifying the MCP server code, but it's a bit more setup (and not "in-code" authentication).

- **Origin Checks:** MCP servers should verify the `Origin` of incoming connections to prevent malicious web pages (running in a user's browser) from accessing your local MCP server via AJAX (a DNS rebinding attack) [32]. If you're exposing the server on the web for your own use, you might only expect specific clients (not browsers) to connect. You can implement an origin check in a similar way to the API key – e.g. allow only a blank or specific Origin. At minimum, when running on `0.0.0.0`, do **not** visit untrusted websites while your server is running without an origin check; a crafted site could try to connect to `http://localhost:8000/mcp` via JavaScript. Authentication also mitigates this risk.

Apart from authentication, consider enabling **encryption** if you run over the open internet. You can serve the MCP endpoint over HTTPS by running behind a TLS termination proxy or using a service like Cloudflare Tunnel or ngrok to provide an https URL. The community is already exploring solutions like hosting MCP

servers on Cloudflare Workers, etc., which automatically give a secure domain [34] [15]. For a home setup, a simple way to get HTTPS is using a tunneling service (ngrok can forward `https://randomsub.ngrok.io -> http://localhost:8000`). If using your own domain, obtaining a certificate (via Let's Encrypt) for your domain and configuring a proxy is worthwhile for security.

Security of the Tools: Also think about what your MCP server is exposing. Make sure to **sandbox or limit** any potentially dangerous functionality. For instance, if you write an MCP server that wraps shell commands or file system writes, be cautious – even with auth, a vulnerability or leaked credential could lead to misuse. Treat your MCP server like any API server in terms of security best practices.

Example: A Reusable MCP Server Framework

Let's pull it all together and outline a simple but flexible MCP server implementation. The goal is to make it **modular**, so you could wrap any API or service by adjusting the tool functions. Below is a skeletal example of an MCP server that we can easily modify for different use cases:

```
from mcp.server.fastmcp import FastMCP
import requests

# Initialize the MCP server
mcp = FastMCP("Example MCP Server")

# Example Tool 1: Wrap an external REST API
@mcp.tool()
def get_weather(city: str) -> dict:
    """Get current weather for a city using a public API"""
    url = f"https://wttr.in/{city}?format=j1"  # Weather API that returns JSON
    resp = requests.get(url, timeout=10)
    resp.raise_for_status()
    return resp.json()  # return the JSON data as a Python dict

# Example Tool 2: Wrap a local library or service
@mcp.tool()
def calc_factorial(n: int) -> int:
    """Compute factorial of n"""
    # (Using math library for demonstration of a local computation)
    import math
    return math.factorial(n)

# You could add more tools or resources here, wrapping any API or functionality
needed.

if __name__ == "__main__":
    # Run in HTTP mode for remote access. Use environment variables or config
```

```
for secrets if needed.
    mcp.run(transport="streamable-http", host="0.0.0.0", port=8000)
```

A few notes on this example framework:

- **Modularity:** The `mcp.tool()` decorator makes it straightforward to wrap arbitrary functions. You can think of each tool as one "API endpoint" for the LLM. If tomorrow you want to expose a database query, you can just add a new function with @mcp.tool. The rest of the server code doesn't need to change. This separation means you can wrap *any* API – whether it's a web API (like the weather example), a database call, or an IoT device control – as long as you can implement it in Python. The MCP server acts as a thin wrapper that translates LLM requests into those API calls.

- **Testing Locally:** You can run `python server.py` to start the server. For quick local testing, you might run it with `host="127.0.0.1"` and use the MCP CLI's inspector: `mcp dev server.py` (this launches the server and opens an interactive UI to simulate an LLM client). This is a good way to ensure your tools behave correctly. For example, you can call `list_tools()` to see if your tools are registered, and `call_tool("get_weather", {"city": "London"})` to see the response.

- **Client Connection:** In a real scenario, a client (LLM agent) must connect to this server. If using OpenAI's functions/agents, you might integrate via their Agents SDK `MCPServerStreamableHttp` class by providing the URL [35]. For instance, `server = MCPServerStreamableHttp(url="http://<your-ip>:8000/mcp")` and then add that to your agent's list of servers. Other platforms like Claude Desktop allow installing a local server by giving the path or URL. The key point is the client needs the address and any auth info to connect.

- **Authentication Integration:** The above code does not implement auth. To add a simple API key check, you could use an environment variable (say `AUTH_TOKEN`) and then wrap each tool's logic to check a global request context for the token. However, since the MCP protocol handles JSON-RPC calls, a cleaner method is to protect at the HTTP layer (as discussed earlier). For example, using FastAPI mount:

```
from fastapi import FastAPI, Request, HTTPException
app = FastAPI()

@app.middleware("http")
async def auth_middleware(request: Request, call_next):
    # Simple token check
    token = request.headers.get("x-api-key")
    if token != "MY_SECRET_TOKEN":
        raise HTTPException(status_code=401, detail="Unauthorized")
    return await call_next(request)

app.mount("/", mcp.streamable_http_app())  # mount MCP at root with auth
middleware protecting it
```

Then run `uvicorn app:app --host 0.0.0.0 --port 8000`. This way, every request requires the correct `X-API-Key`. This is just one approach – you could similarly enforce basic auth, etc. The idea is to put a minimal auth gate in front of the MCP server.

- **Security and Maintenance:** Keep your server up to date with the latest MCP SDK versions, as the protocol is evolving rapidly. Check official docs for any changes (for example, SSE was superseded by streamable HTTP, and there may be new transports or features in future). Also, log your server's output (`stderr` from FastMCP is used for logging [36]) so you can monitor usage and errors.

By following this guide, you have a basic MCP server running on Windows or Linux, accessible via a URL, and able to wrap virtually any API or function you need. This MCP server can now serve as a **bridge** between powerful AI models and the rich world of external data and actions – all through a standardized protocol that LLM applications understand [37] [38].

Sources: The information and examples above are drawn from official MCP documentation and community guides, including the MCP Python SDK reference and spec, as well as real-world MCP server implementations. Key references are included inline for further reading on specific points. Enjoy building your MCP server, and happy coding!

---

[1] [20] [21] [22] [28] [31] [33] GitHub - modelcontextprotocol/python-sdk: The official Python SDK for Model Context Protocol servers and clients
https://github.com/modelcontextprotocol/python-sdk

[2] MCP server: A step-by-step guide to building from scratch - Composio
https://composio.dev/blog/mcp-server-step-by-step-guide-to-building-from-scrtch/

[3] [4] [15] [29] [34] A Beginner's Guide to Visually Understanding MCP Architecture | Snyk
https://snyk.io/articles/a-beginners-guide-to-visually-understanding-mcp-architecture/

[5] [25] [35] Model context protocol (MCP) - OpenAI Agents SDK
https://openai.github.io/openai-agents-python/mcp/

[6] [7] [8] [9] [10] [11] [12] [13] [14] [16] [30] [32] [36] Transports - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-03-26/basic/transports

[17] [18] [19] GitHub - yonaka15/mcp-server-runner: A WebSocket server implementation for running Model Context Protocol (MCP) servers. This application enables MCP servers to be accessed via WebSocket connections, facilitating integration with web applications and other network-enabled clients.
https://github.com/yonaka15/mcp-server-runner

[23] GitHub - qdrant/mcp-server-qdrant: An official Qdrant Model Context Protocol (MCP) server implementation
https://github.com/qdrant/mcp-server-qdrant

[24] [26] [27] Running Your FastMCP Server - FastMCP
https://gofastmcp.com/deployment/running-server

[37] [38] Getting Started with Model Context Protocol (MCP) | by WS | Apr, 2025 | Medium
https://medium.com/@Shamimw/getting-started-with-model-context-protocol-mcp-3c2608a9b5b5