ChatGPT

Building a FastMCP Server with Python & Docker for Modern MCP Clients

Table of Contents

Introduction

Model Context Protocol (MCP) is an open standard for connecting AI models to tools and data via a unified interface. First introduced by Anthropic in late 2024 and now adopted by OpenAI, MCP replaces fragmented custom integrations with a single secure protocol [1]. In practical terms, an **MCP server** is like a mini-API designed specifically for AI agents (LLMs) rather than human developers. Modern AI apps like **Claude Desktop, Cursor IDE, OpenAI's agent interfaces, and Docker's MCP Toolkit** can all act as **MCP clients**, automatically discovering and using these servers as "tools" in their workflows.

This guide will show you how to **build, design, and ship a FastMCP-based MCP server** in Python – from running it on your machine to packaging it in Docker and publishing on Docker Hub. We'll keep things practical and minimal (we're aiming for a hackathon MVP in ~6 hours), with a focus on **structure and best practices** rather than complex logic. Along the way, we'll cover critical topics like handling per-user API keys

(so each user supplies their own secret, e.g. an API token, when they run your tool) and how designing an MCP tool for AI agents differs from building a traditional REST API.

What we're building: For illustration, imagine we need an AI assistant to interact with a service (say, a database or a web API). In our case, we'll use a GraphQL API (e.g. the Canvas LMS API) as the backend that our MCP server will call. We'll create an MCP server with at least two example tools (endpoints) – one that calls the GraphQL API (using a user-provided API key) and another simple utility tool. We'll implement it with **FastMCP** (the official Python framework for MCP servers) and demonstrate how to run it on bare metal and in Docker. We'll also discuss how to publish it to Docker Hub and integrate it with MCP-enabled clients (like one-click adding your tool in Claude or Cursor).

1. Bare-Metal FastMCP MCP Server

In this section, we'll set up a basic FastMCP server project and run it locally (without Docker). This is useful for quick development and testing. We assume you already have Python 3.10+ installed (FastMCP requires Python ≥3.10 [2]).

Project Setup and Structure

Start by creating a project directory for your MCP server. For a simple project, you can keep everything in a single module/file, or a very small folder structure. For example:

```
$ mkdir fastmcp_canvas_tool
$ cd fastmcp_canvas_tool
```

Inside this folder, we'll have the following structure:

```
fastmcp_canvas_tool/
├── mcp_server.py          # The FastMCP server definition and tools
├── requirements.txt       # Python dependencies
└── Dockerfile             # (optional for later) Containerization recipe
```

(If you prefer, you can use a `pyproject.toml` *with Poetry or Hatch. But for simplicity, we'll use a requirements file.)*

Install FastMCP and any other dependencies. For our use case we'll need `fastmcp` and `requests` (for calling the GraphQL API). You can install them with pip:

```
pip install fastmcp requests
```

> *Note:* FastMCP also provides a CLI tool `fastmcp` and can integrate with a tool called `uv` for managing environments [3]. However, for a quickstart, a direct `pip install` is fine. Ensure FastMCP is at least version 2.x for the latest features.

Now, let's write the server code.

Writing the MCP Server Code

Open `mcp_server.py` in your editor and add the following code (with explanatory comments):

```
# mcp_server.py
from fastmcp import FastMCP
import os
import requests

# Instantiate the MCP server
mcp = FastMCP("CanvasMCP")  # Give your server a name (for identification)

# Read configuration (API endpoint and key) from environment variables
CANVAS_API_URL = os.environ.get("CANVAS_GRAPHQL_URL", "https://
canvas.example.edu/api/graphql")
API_KEY = os.environ.get("CANVAS_API_KEY")  # Users must provide their own API
key at runtime

@mcp.tool
def get_course_info(course_id: str) -> dict:
    """Fetch details of a course by ID from the Canvas GraphQL API."""
    if API_KEY is None:
        # If no API key is provided, we raise an error (the LLM client will
receive this message).
        raise RuntimeError("No API key provided. Please set the CANVAS_API_KEY
environment variable.")
    # Construct GraphQL query and variables (placeholder example)
    query = """
    query getCourse($id: ID!) {
      course(id: $id) {
        name
        description
        startDate
      }
    }
    """
    variables = {"id": course_id}
    headers = {"Authorization": f"Bearer {API_KEY}"}
    try:
        response = requests.post(CANVAS_API_URL, json={"query": query,
"variables": variables}, headers=headers)
        data = response.json()
    except Exception as e:
        # Network or unexpected error
        raise RuntimeError(f"GraphQL request failed: {e}")
```

```
    if 'errors' in data:
        # If GraphQL returned errors, include them in the exception message
        raise RuntimeError(f"GraphQL error: {data['errors']}")
    return data.get('data', {})  # Return the "data" field of the GraphQL
response (a dict)

@mcp.tool
def echo(message: str) -> str:
    """A simple echo tool that returns the input message."""
    return f"You said: {message}"

if __name__ == "__main__":
    # Run the MCP server in HTTP mode for local testing
    mcp.run(transport="http", host="0.0.0.0", port=int(os.environ.get("PORT",
8000)))
```

Let's break down what's happening here:

- We create a `FastMCP` server instance and decorate two functions with `@mcp.tool`. By decorating a function, it's automatically **registered as a tool** in the MCP server [4]. Each tool has a name (defaulting to the function name, e.g. `"get_course_info"` and `"echo"`) and uses the function's type hints to define its input and output schema. We've provided docstrings for each tool – these descriptions will be visible to the LLM agent when it inspects the tool's capabilities.

- **API Key config:** We expect an environment variable `CANVAS_API_KEY` to hold the user's Canvas API token. We do **not** hard-code any secrets. At runtime, if `API_KEY` is missing, the tool raises a clear error (so the user/agent knows they need to provide a key). We similarly take `CANVAS_GRAPHQL_URL` from env, with a default placeholder. In a real scenario, each user might point to their own Canvas instance URL; making it configurable helps reuse the container in different contexts.

- **GraphQL call logic:** In `get_course_info`, we prepare a GraphQL query string and variables. We then `POST` to the Canvas GraphQL API endpoint with the appropriate Authorization header. The result is expected to be JSON; we check for an `'errors'` field and raise an exception if present (the exception message will be sent back through MCP as an error response). On success, we return the `'data'` portion of the response. This way, the output is a structured Python dict (which gets serialized to JSON for the LLM). We keep the logic simple and **stateless**: each call uses the input parameters and returns results without relying on any global mutable state.

- **A second tool (`echo`)** is provided to demonstrate multiple tools. It's trivial (just returns the input message), but it shows how easy it is to add another endpoint. In a real project, this might be another GraphQL-backed function (e.g. `list_user_courses(user_id)` vs `get_course_info(course_id)`), but the pattern is the same.

- **Server launch (`mcp.run`):** In the `if __name__ == "__main__":` block, we call `mcp.run(transport="http", host="0.0.0.0", port=...)`. This starts the server. By

specifying `transport="http"`, FastMCP will run an HTTP server for our tools (suitable for remote or browser-based clients). We bind to `0.0.0.0` so that it's accessible to Docker or other machines if needed, and use port 8000 by default [5]. (FastMCP's built-in server uses Uvicorn under the hood to serve HTTP requests.) When this runs, our MCP server will listen at `http://localhost:8000/mcp` by default [6] – the `/mcp` path is the standard endpoint for MCP interactions.

- *Note:* If we omitted `transport="http"`, FastMCP would default to the `stdio` transport (where the server communicates via standard I/O streams). STDIO is used when a client launches the process directly (e.g. Claude Desktop can spawn a local server and talk via pipes). For local testing and for Docker, HTTP is more convenient, so we explicitly choose it. FastMCP also supports running as an ASGI app (which you could mount on a larger FastAPI app or run via Uvicorn manually) [7] [8], but that's not needed for a simple deployment.

Running the server locally: Ensure you've set the required environment variable for the API key before running. For example, in your terminal:

```
export CANVAS_API_KEY="sk-...your Canvas token..."
# (Optional) export CANVAS_GRAPHQL_URL="https://<your_canvas_domain>/api/
graphql"
python mcp_server.py
```

If everything is correct, you should see FastMCP start up an HTTP server (likely it will log something to the console about the server running on port 8000). The server is now waiting for MCP client connections.

Testing (optional)

At this point, our MCP server is running locally. To test it manually, you could use the FastMCP Python client or an MCP-compatible LLM client:

- **Using FastMCP's client (for a quick sanity check):** Run a Python snippet to call your tool:

```python
import asyncio
from fastmcp import Client

async def test():
    client = Client("http://localhost:8000/mcp")
    async with client:
        result = await client.call_tool("echo", {"message": "Hello MCP"})
        print(result)
asyncio.run(test())
```

This should print the result of the echo tool. You could also call `"get_course_info"` with a sample course_id if you have one (and a valid API key).

- **Using an LLM client:** You could configure Claude Desktop or Cursor to connect to `http://localhost:8000/mcp` if they allow remote connectors. For instance, OpenAI's developer console (or API via Azure's tool system) supports referencing an MCP server by URL [9]. If your client can't connect to an arbitrary URL due to CORS or network, another approach is to run in stdio mode (which those clients can spawn), but that's beyond our scope here.

Bare-Metal Setup Checklist

- [x] **Dependencies installed:** FastMCP and any HTTP clients (e.g. `requests`).
- [x] **FastMCP server code written:** with at least two `@mcp.tool` functions defined (including docstrings and type hints for clarity).
- [x] **Environment variables set:** Your API key (and any configurable URLs) exported to the environment *before* running the server.
- [x] **Server running:** Use `mcp.run(..., transport="http")` for an HTTP server accessible at `http://localhost:8000/mcp` [6].
- [x] **Basic test done:** Optionally, called a tool (e.g. via FastMCP client or another method) to verify the server responds correctly.

With the server working on bare metal, let's containerize it for easier distribution.

2. Dockerizing the MCP Server (Local Container)

Packaging your MCP server in a Docker container ensures it runs in a consistent environment and makes it easy to share or deploy. We'll create a simple Docker image that includes our Python code and runs the FastMCP server. This is useful for local use (instead of running the Python script directly) and is a stepping stone to publishing on Docker Hub.

Writing a Minimal Dockerfile

Create a file named `Dockerfile` in the project directory. We will use a lightweight Python base image and copy our code into it:

```
# Use a slim Python image as base
FROM python:3.11-slim

# Set working directory
WORKDIR /app

# Install dependencies
# (copy requirements.txt separately to leverage Docker cache on rebuilds if deps
don't change)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy the application code
COPY mcp_server.py .
```

```
# Expose port 8000 for MCP HTTP
EXPOSE 8000

# Set environment variables for security (avoid Python buffering issues, etc.)
ENV PYTHONUNBUFFERED=1

# When container starts, run the MCP server
# Use uvicorn or fastmcp CLI if desired, but running our script directly is
simplest:
CMD ["python", "mcp_server.py"]
```

A few notes on this Dockerfile:

- We use an official Python slim image for smaller size. It already has Python installed. We choose 3.11 (which supports FastMCP); you could use 3.10+.
- We set the working directory to `/app` and then copy in `requirements.txt` first. Installing dependencies first (before copying the full code) allows Docker to cache the pip install layer, so rebuilding the image is faster if you change only your code but not the dependencies.
- We copy our `mcp_server.py` into the image. Our server doesn't depend on any other files, so this is sufficient. If you had more modules, you might copy a whole directory.
- We expose port 8000, which is the port our server runs on. (This is mainly informational for the user running `docker run -p`; Docker doesn't automatically publish it, it just denotes the default.)
- We set `PYTHONUNBUFFERED=1` as a best practice for containers to ensure logs print immediately.
- The `CMD` runs the Python server. This will execute the `if __main__` block in our script, thus calling `mcp.run(...)` as we wrote it. In our case, that launches the HTTP server on port 8000. We could alternatively use `CMD ["fastmcp", "run", "mcp_server.py:mcp", "--transport", "http", "--port", "8000"]` to achieve the same via the FastMCP CLI, but since our script already handles it, we keep it simple.
    - *Tip:* The FastMCP docs note that by default the MCP server will mount at the `/mcp` path on the server [6]. If you needed a different base path, FastMCP's `run()` supports a `path` parameter or you can configure it when creating an ASGI app [10]. We're fine with the default.
- We didn't explicitly handle the API key in the Dockerfile. **Important:** We do **not** bake any secrets into the image. Users will provide `CANVAS_API_KEY` at runtime (as an environment variable when running the container). We'll show that next.

Before building, double-check that `requirements.txt` exists and lists our dependencies (`fastmcp` and `requests`, possibly pinned to specific versions if needed). For example:

```
fastmcp>=2.0
requests>=2.28
```

Now build the Docker image. Run this in the project directory (where the Dockerfile is):

```
docker build -t fastmcp-canvas-tool:latest .
```

This will produce a local image named `fastmcp-canvas-tool:latest`. The build process will install FastMCP and our other dependencies in the image, then add our code.

Building and Running the Docker Container

Once built, run the container locally to test it:

```
docker run -d \
  -p 8000:8000 \
  -e CANVAS_API_KEY=<<your_canvas_api_token>> \
  --name my_canvas_mcp \
  fastmcp-canvas-tool:latest
```

Let's break down this `docker run` command:

- `-d` runs the container in detached mode (in the background). You can omit `-d` to run in the foreground and see logs directly.
- `-p 8000:8000` publishes the container's port 8000 to your host's port 8000. This means you can access the MCP server at `http://localhost:8000/mcp` outside the container (just like when you ran it bare-metal). If you already have something on 8000 or want a different port externally, adjust the left side (e.g. `-p 9000:8000` would expose it on localhost:9000).
- `-e CANVAS_API_KEY=...` sets the environment variable inside the container. **This is where you provide your secret API key at runtime.** Notice, each user can supply their own key when they run the container. The image itself doesn't contain any API keys – it reads whatever the user passes. This pattern ensures **per-user secrets** and prevents hardcoding sensitive info. (If there were other config env vars like `CANVAS_GRAPHQL_URL`, you'd pass those with `-e` as well. If you don't pass it, our code will use the default example URL.)
- `--name my_canvas_mcp` is optional, just naming the container for easier reference (so you can stop it with `docker stop my_canvas_mcp`).
- Finally, `fastmcp-canvas-tool:latest` is the image name:tag to run.

After running this, you can check logs to ensure it started properly:

```
docker logs my_canvas_mcp
```

You should see output indicating the server is listening (the FastMCP server might log something like "Serving on http://0.0.0.0:8000/mcp"). If everything is good, try accessing a tool. For example, you can exec into the container or from your host use the FastMCP client as before, pointing to `localhost:8000/mcp`. Any AI agent that supports MCP can now also connect to this URL (if network-accessible). For instance, if you have Cursor's CLI, you might add this as a remote tool; or in Claude Desktop, you might configure a "remote connector" (though Claude often prefers launching its own processes or using the Docker MCP Toolkit – more on that soon).

Docker Container Checklist

- [x] **Dockerfile created:** Base image appropriate and `fastmcp` installed inside.
- [x] **Image built successfully:** (`docker build` completes without errors).
- [x] **Container runs and exposes MCP port:** Use `docker run -p 8000:8000 -e CANVAS_API_KEY=...` to start it.
- [x] **Secrets passed at runtime:** API key provided via `-e` flag (verified that no sensitive data is in the image).
- [x] **Basic functionality test:** Confirmed that the container's MCP server responds (e.g., via logs or a test call to a tool).

Now that it works locally, let's share this tool with others by publishing the image.

3. Publishing to Docker Hub and MCP Client Integration

Sharing your MCP server image on Docker Hub allows others (or yourself on another machine) to pull and run it easily. In the context of modern MCP clients, Docker Hub plays a special role: it has an **MCP Catalog** where certain images are listed for easy discovery, and tools like Docker's MCP Toolkit can fetch images directly. We'll cover how to push your image to Docker Hub and how an end-user would configure it.

Tagging and Pushing the Image

First, make sure you have a Docker Hub account (and you're logged in via `docker login`). Choose a repository name for your image, typically in the format `dockerhub_username/imagename`.

For example, if your Docker Hub username is `aihacker`, you might tag the image as `aihacker/canvas-mcp:1.0` (the `:1.0` is a version tag; you can also use `:latest`). To tag and push:

```
docker tag fastmcp-canvas-tool:latest aihacker/canvas-mcp:1.0
docker push aihacker/canvas-mcp:1.0
```

This uploads the image to Docker Hub. Once pushed, anyone can pull and run it using `docker run` as we did (they'll need to provide their own API key env var).

When publishing, it's good to provide a README or description on Docker Hub explaining what your MCP server does and how to configure it. For example, mention that it expects `CANVAS_API_KEY` (and any other env vars) at runtime, and what tools it provides. This helps users know how to use it.

If you want your server to be discoverable in Docker's **MCP Catalog** (which surfaces in Docker Desktop's MCP Toolkit UI), you might need to follow Docker's guidelines (such as using the `mcp/` prefix for official images, or submitting to their catalog). For a hackathon project, this is likely not necessary – you can simply share the repo name, or manually add it to clients.

Configuring MCP Clients with Docker Hub Images

There are a few ways an end-user (or you, on your own machine) can run your MCP server via Docker:

- **Manual `docker run`:** The user can always run `docker run -p 8000:8000 -e CANVAS_API_KEY=... yourrepo/canvas-mcp:1.0` themselves. Then configure their LLM client to connect to `http://localhost:8000/mcp`. This is a bit manual but works everywhere.

- **Docker Desktop MCP Toolkit:** Docker has introduced a GUI integration for MCP. In Docker Desktop, there's an **MCP Toolkit** that lets users browse a catalog of MCP servers and run them with one click [11] [12]. If your image is on Docker Hub, a user can add it through the *Add Server* flow. For example, they might find it by name or add a custom server by providing the image name. Once they select it, the toolkit will pull the image and prompt the user for any required configuration.

- In Docker's MCP Toolkit UI, after adding a server, there's typically a **Configuration tab** where the user can input secrets or options. For instance, the **Docker Hub MCP Server** (an official tool) requires a username and token – the UI asks for those and sets them as env vars when launching the container [13]. For our server, the user should see an option to set `CANVAS_API_KEY` (Docker can glean known env vars if the image or catalog entry defines them; if not, the user can still manually add it).

- Once configured, the user can **start** the server container from the UI. Docker MCP Toolkit will manage the lifecycle of the container.

- Additionally, Docker's MCP Toolkit can act as a **gateway** for clients: e.g., one can connect Claude Desktop through the Toolkit rather than directly to `localhost`. The Toolkit essentially runs a local orchestrator that both Claude and the container talk to [14]. This is advanced usage, but the key point is Docker is smoothing the UX of connecting AI apps to containerized MCP servers.

- **Claude Desktop direct config:** Claude's desktop app normally runs local connectors via config JSON (spawning processes). However, if using Docker, the recommended approach is now to use the Docker MCP Toolkit. Claude can connect to the toolkit as a client, gaining access to all running MCP containers [15]. So a user would: (1) Launch the server in Docker via the toolkit, (2) In Claude Desktop, enable "Connect to external MCP Toolkit" (this might be a one-time setting), and then Claude will list those tools automatically. This way, you don't need to manually edit Claude's JSON for a Docker-based server.

- **Cursor IDE:** Cursor has an MCP integration as well. If using Docker, one could run the container and then configure Cursor to use it (Cursor's docs mention connecting external MCP servers via a config or command). For example, Cursor might allow specifying an MCP server URL in its settings or CLI (like `cursor-agent --connect http://localhost:8000/mcp`). If Cursor has its own "directory," publishing your image on Docker Hub means it could show up in their list (some community sites aggregate known MCP servers).

Regardless of method, the core is the same: **the code inside your image doesn't change** whether run locally or via Docker Hub. The difference is how the user supplies configuration:

- Locally: environment variables (or a `.env` file) when running the Python script.
- Docker local: `-e` flags or Docker Compose env config.
- Docker Hub/Toolkit: through a UI form that ultimately sets environment variables for the container [13].
- Claude's JSON (if not using Toolkit): by providing an `env` map in the config (as seen in the Claude config snippet, they pass API keys via env for each server [16]).

Ensure that in your Docker Hub README or documentation, you **clearly state how to configure the container at runtime**, especially any required env vars (like our `CANVAS_API_KEY`). For example, "This MCP server requires a Canvas API token. When running the container, set the `CANVAS_API_KEY` environment variable to your token. If using Docker's MCP UI, enter the API key in the Configuration tab before starting the server [13]."

Publish & Integration Checklist

- [x] **Image tagged with your Docker Hub repo name:** e.g. `username/imagename:tag`.
- [x] **Image pushed to Docker Hub:** (Verify on Docker Hub that the repo is visible, and includes a description of usage and env vars).
- [x] **Documentation of config:** Users know to supply their API key as `CANVAS_API_KEY` at runtime (via `-e` or UI).
- [x] **Test pull/run:** Try pulling the image on another machine or with `docker run` (no local files) to ensure it works independently.
- [x] **(Optional) Docker MCP Catalog:** Consider if you want to submit the image to Docker's official MCP catalog for easier discovery (not necessary for hackathon, but nice if aiming for wider use).

Now that the practical setup is covered, let's dive into design principles: how to think about building MCP tools versus traditional APIs.

4. Managing API Keys and Secrets in MCP Servers

One of the critical aspects of MCP server design is **secret management**. Typically, your MCP server might need to call external APIs (like our GraphQL service) which require API keys or tokens. It's important to handle these secrets in a secure, flexible way so that *each user of your tool can provide their own credentials*. This section explains best practices for secrets in MCP deployments and how they differ between build-time and runtime, and across bare metal, Docker, and cloud usage.

Runtime vs Build-Time Secrets

- **Build-Time Secrets:** These are secrets used *while building the container/image*, often to fetch private dependencies or perform setup. In our case, we have **no need for build-time secrets** (we use public packages and our code does not include sensitive data). As a rule, **do not bake API keys or user secrets into your image or code**. Not only would that make the image specific to one user, it could also leak the secret (anyone who pulls the image could inspect it). Build-time secrets might apply if, for instance, your Dockerfile had to download something from a private repository – but even then, those wouldn't be user-specific keys, they'd be for building.

- **Runtime Secrets:** These are provided when the server is launched (after it's built). In MCP scenarios, runtime secrets are the way to go for API keys. Each user who wants to run the MCP server should supply their own keys as environment variables, config files, or secure inputs. We designed our server to expect `CANVAS_API_KEY` at runtime. At runtime, secrets can be injected via:

- **Environment variables:** as we've shown (`-e` in Docker, `export` in shell, or via orchestrators). This is very common and straightforward.
- **MCP Client configuration:** e.g., Claude Desktop's JSON config includes an `env` section for each tool [16], which sets environment variables when launching the tool's process. Docker's MCP Toolkit UI similarly provides fields to set env vars [13].
- **Command-line args or config files:** Less common in MCP contexts, but possible (e.g., a user could pass `--api_key=XYZ` to a custom launch script, or mount a volume with a config file). For simplicity, we stick to env vars.

Why runtime secrets? Because it allows the same code/image to be used by everyone, each with their own credentials. Imagine an MCP server for GitHub API: you wouldn't bake your personal PAT (Personal Access Token) into it; you'd have each user provide theirs. This way, each user's permissions and rate limits apply to their instance of the tool, and you don't have to distribute secrets.

Secrets in Bare-Metal vs Docker vs Docker Hub Scenarios

Let's compare how the secret gets to the server in our three deployment scenarios:

- **Bare-Metal (direct run):** The user (developer) sets `CANVAS_API_KEY` in their environment (e.g., in the terminal or in an .env file that the Python app loads). In our code, `os.environ.get("CANVAS_API_KEY")` picks it up. If the user forgets, our code throws a clear error. The key never leaves the user's machine and isn't stored in our code. Also, avoid printing the key in logs. Our example only logs an error if missing, not the key value itself (which is good).

- **Local Docker Container:** The user passes `-e CANVAS_API_KEY=...` when running `docker run`. This injects the key into the container's environment. Inside the container, our code reads it the same way. The key isn't stored in the image, only in that container instance's memory. If the user stops the container, the env var is gone. **Do not** write the key to disk inside the container. (Our code doesn't.) Also be mindful of Docker commands – if you use `docker logs`, if our server had printed the key, it would appear. We made sure not to. If you accidentally did a `print(API_KEY)`, that would leak to logs; so avoid that. Runtime secrets via `-e` are straightforward, but ensure that any

CI/CD or scripts that run your container also handle secrets securely (e.g., don't commit them in a `docker-compose.yml` in a public repo).

- **Docker Hub / Cloud MCP Deployment:** When someone pulls your image from Docker Hub via an MCP client (like Docker's MCP Catalog or another service like Claude's extensions), they will be prompted to provide the API key. For example, Docker's UI will have a field for `CANVAS_API_KEY` [13]. The Docker Desktop MCP Toolkit actually can store secrets securely and inject them when running the container (it likely doesn't expose them to other processes). Another example: If an enterprise is using Azure API Management's MCP support, they might configure the secret in a profile that injects it into the container [17]. In any case, the pattern is the same: the *user* (or client app on behalf of the user) provides the key at runtime. The image author (you) just has to document what's needed and make sure your code reads from the environment.

A contrasting scenario: sometimes people think of putting an API key directly in code for convenience. **Avoid that for MCP tools intended for others.** The only time a secret might live in code is if it's purely an internal tool for yourself and you're not sharing the container – but even then, better practice is to keep it out of the code (e.g., in a local `.env` that you don't commit). For hackathons or demos, you might hardcode a key to move fast, but since our goal is to *publish* the tool possibly, we did it properly with env vars.

Per-User vs. Single-User: One more nuance – some MCP servers might support *multiple* API keys for multiple users (if the server itself had a user concept). That's advanced (you'd need the protocol to pass something per request, which MCP doesn't natively do for external secrets). Typically, the model is one running instance of the MCP server corresponds to one user's context (since the user running it supplies the config). If you needed to allow dynamic per-call credentials, you'd have to design that into your tool interface (not common; so we assume one key per server instance).

Other secret types: Apart from API tokens, OAuth flows could be used. Docker's MCP Toolkit can handle OAuth for some official servers (like GitHub) – it will do a browser auth and then inject a token [18] [19]. For your custom server, if you wanted OAuth, you'd have to implement an OAuth device flow or similar in your tool (beyond our scope). Most likely, you'll use static tokens in hackathons.

Do not log or return secrets: If your tool, say, takes an input that is a secret (not our case, but imagine a tool that asks for a password), be sure to not include that in any response. LLMs have had issues where tools would echo secrets inadvertently, which is bad [20]. Our design avoids this entirely by handling secrets out-of-band via env vars.

To summarize, **the MCP ecosystem expects secrets to be provided by the user's client configuration, not hardcoded in the server image.** This holds true for Claude Desktop config files [16], Docker Hub MCP setups [13], and others. As the MCP server developer, you just need to: 1. Use something like `os.environ.get()` to fetch the secret. 2. Document that environment variable. 3. Handle the case where it's missing (so the user gets an error rather than a puzzling failure).

Secrets Management Checklist

- [x] **No hardcoded secrets:** API keys or tokens are not present in code or Dockerfile.
- [x] **Uses environment variable:** Server reads `API_KEY` (and other secrets) from env at runtime.

- [x] **Per-user configurable:** Each user can supply their own key when running the server (via `docker run -e` or client UI).
- [x] **Missing key handled:** Server checks for the key and provides a clear error if not set (avoiding silent failures).
- [x] **No secret leakage:** The server does not log the secret or include it in responses. (Errors are generic and safe.)
- [x] **Documentation updated:** README or guide instructs users how to provide their secret (e.g. "set environment variable X to your API token").

With configuration and deployment covered, let's move to conceptual differences in designing the *API* of your MCP server (the tools) versus typical REST APIs.

5. MCP vs REST API: Design Differences and Best Practices

When building our MCP server, we need to shift our mindset from "designing a web service for human developers" (REST paradigm) to "designing tools for an AI agent". While both are HTTP-based in our case, the use-cases and consumption patterns are different. This section compares MCP and REST design and provides guidelines for creating agent-friendly tools.

MCP servers act as an intermediary layer between AI agents and traditional APIs or services. The AI (LLM) talks to the MCP server in high-level terms, and the server translates those requests into API calls to the underlying service [21] [22] (e.g., GraphQL or REST calls), then returns structured results back to the AI.

Designing for LLM Agents vs. Human Users

Audience: A REST API is typically consumed by human programmers. Those programmers read documentation, write code to call the API, handle errors, etc. In MCP, the "consumer" is an AI (or an AI-driven system) that *dynamically discovers and invokes* your tools. There may not be a human in the loop for making each call (the LLM decides to call tool A then B). This leads to several key differences:

- **Discovery vs Documentation:** In REST, you publish API docs and developers manually integrate. In MCP, the client can ask "what can you do?" and the server provides a list of tools and their schemas (this is part of the MCP handshake and protocol negotiation) [23] [24]. The LLM sees something like: tool name, description, input fields and types, output types. So it's crucial to make those clear and correct (use descriptive names, write helpful docstrings, use precise types). The agent will use that info to plan calls. Essentially, MCP has **automatic capability discovery** whereas REST relies on manual documentation [25].

- **Granularity of operations:** LLM agents are good at combining multiple simple tools to accomplish a complex task. With REST, you might be inclined to create one endpoint that does a complicated multi-step operation (to minimize round trips for the developer). For MCP, it can be better to expose *atomic actions* and let the LLM compose them. For example, in a RESTful design you might have a single endpoint `/report` that, given a user id, returns a summary containing user info, their courses, their grades, etc., because a human programmer requested all in one go. In MCP, you might instead have:

- `get_user(id) -> User`

- `list_user_courses(user_id) -> List[Course]`
- `get_course_performance(user_id, course_id) -> dict` (some stats)

The LLM could first call `get_user`, then `list_user_courses`, then for each course call `get_course_performance`, and finally synthesize a report itself. This aligns with the fact that the LLM can iterate and branch on results. It also keeps each tool's purpose narrow and clear.

- **Idempotence and side effects:** Agents might retry or call tools in unexpected orders. If a tool is non-idempotent (e.g. making a purchase or sending an email), you need to be careful. Ideally, **design tools to be idempotent or have clear, safe effects**. Read operations are naturally safe. Write operations (like "create" or "delete") are trickier – the AI could accidentally call them multiple times if it's unsure of the outcome. If your use case requires such actions, consider adding checks in the tool (e.g. if creating something that already exists, maybe return a graceful message instead of duplicating). Also use the MCP protocol's features if available – it might support indicating an action is irreversible (though as of now, it's mostly up to the agent's logic).

- **Structured outputs vs Presentation:** REST APIs often return JSON, which is fine for both humans and machines, but sometimes APIs include HTML or formatted text for humans. MCP tools should return data in a structured form that's easy for the LLM to parse (JSON is ideal, which is naturally what Python objects become). Then the LLM can decide how to present it to the end-user. So, avoid HTML or Markdown in the raw output of tools unless the tool's purpose is specifically to fetch formatted content (like a web scraper tool might return Markdown). In our example, returning a Python dict from `get_course_info` which becomes JSON like `{"course": {"name": ..., "description": ...}}` is good. The LLM can then incorporate that into its answer.

- **Error handling:** In REST, if something fails, you might return an HTTP error status and a message that a developer will interpret. In MCP, errors will be surfaced to the AI. FastMCP uses JSON-RPC, which has error codes and messages. You should raise exceptions with meaningful messages (not just "Error 123"). The AI might see the error text. For instance, we did `raise RuntimeError("GraphQL error: ...")` – that message could be read by the AI, which might then decide to tell the user "I couldn't fetch the course info due to a permissions error" or something. Keep error messages concise and factual (the AI might paraphrase them). And avoid leaking sensitive info in errors. Standardizing errors is beneficial – MCP tends to have a more consistent error format across tools (thanks to JSON-RPC) [26], whereas different REST APIs have different styles.

In summary, think of MCP tools as **functions you're exposing to an AI**, whereas REST endpoints are **services you expose to a developer**. The AI is both smarter and dumber than a human: smarter in that it can integrate on the fly and chain calls, dumber in that it doesn't truly understand semantics beyond what you give it and can't ask clarifying questions if your tool is ambiguous. So design with clarity and simplicity in mind.

Tool Granularity and Schema Design

Some concrete patterns:

- **One function per discrete action:** Each MCP tool should do one thing well. For example, one tool to retrieve data, another to update data, rather than a single tool that does both based on some parameter. This is partly because the AI might confuse which mode to use, and partly for safety (you don't want a read to accidentally trigger a write because the AI passed a wrong flag).

- **Input schema:** Use appropriate types. FastMCP allows types like `int`, `str`, `bool`, even `Enum` or Pydantic models for more complex input. Use these to constrain the AI's input. For instance, if a parameter should be one of a few options, use an `Enum` – the AI will see the allowed values and is less likely to hallucinate something else. In our simple case, `course_id` is a `str` (maybe Canvas uses opaque IDs or globally unique strings). If it were numeric, we'd use `int`. Using the correct type helps the MCP handshake inform the AI (the protocol shares type info, so the AI knows "this tool needs an integer" etc.).

- **Output schema:** Similarly, ensure the return type is something the AI can work with easily. If returning a complex object, you might define a dataclass or Pydantic model and use that as the return type in your function signature. FastMCP will translate it to JSON. For example, you could define a `@dataclass Course` with fields, and have your tool return a `Course` instance; the framework would turn it into a dict. For hackathon speed, returning a Python dict or list as we did is fine. Just make sure it's serializable to JSON (built-in types or dataclasses).

- **Naming:** Tool names should be short but descriptive (and usually action-oriented verbs). For instance, `get_course_info` is clear. Avoid overly generic names like `query` or `run` – remember the AI chooses tools by name sometimes. "query" could mean anything, but "get_course_info" is clear. Also, the tool description docstring is important – it should succinctly describe what the tool does, maybe mention important caveats. Our `get_course_info` docstring says "Fetch details of a course by ID from the Canvas GraphQL API." So the AI knows it needs an ID and what it will get.

- **Statelessness:** Each tool call should ideally not depend on a previous call's side-effects (unless absolutely necessary). This is because the AI might call them out of order or not realize it needs to call one first. If you do have state (like a login tool that sets a token for subsequent calls), consider encapsulating that within the MCP server (e.g., the first call saves something in memory for later). But avoid if possible in a short hack – it complicates things. In our design, each call is self-contained (you always provide the course_id, etc.). There's no hidden session state except the environment config (API key).

- **Multiple tools vs parameters:** Sometimes you face a design choice: should I make two separate tools, or one tool with a mode parameter? Example: a search functionality that could search by user or by course. You could do `search(type: str, query: str)` where type is "user" or "course", *or* two tools `search_users(name: str)` and `search_courses(keyword: str)`. The latter is often better for MCP. Why? The AI doesn't have to figure out the parameter to use; it will just pick the appropriate tool. It's less likely to mix up the modes. Each tool is simpler (no branches internally). The only cost is a bit of duplication or more endpoints – but that's fine, AI can handle many available tools.

Error Handling, Idempotence, and Retries

As touched on, these are important given the autonomous nature of AI agents:

- **Standardize error outputs:** MCP (especially via JSON-RPC) will typically give errors in a structured way (with a code and message). Ensure your errors are meaningful. For example, if an API call returns a 403, instead of a generic "GraphQL error", you might catch that and raise `RuntimeError("Unauthorized (API key may be invalid)")`. This string will go to the AI; a good AI might recognize it and inform the user to check the API key. If it was just "GraphQL error", the AI might be confused. Don't be too verbose or stack-tracey – just the high-level reason.

- **Idempotence:** If a tool is inherently idempotent (like getting info), no issues. If not, consider adding protective checks. Perhaps a `create_resource(name)` tool should check if the resource already exists with that name to avoid duplicates on retry. Or ensure the AI explicitly requests a non-idempotent action (maybe naming the tool clearly like `delete_course` so it's obviously destructive). The AI frameworks often ask user permission or have flags for dangerous tools, but as developer you should assume it might call it inadvertently. Some MCP clients have a concept of "ask for permission" for certain tools (especially ones labeled with side effects) [27]. It's not formal in the protocol but convention. So you might note in the docstring "(Caution: this action will delete data)" to signal to the agent or user.

- **Retries:** Agents sometimes retry the same call if they got an unexpected response or think maybe another attempt with slightly different input could work. Design your tools to handle multiple calls gracefully. For example, if two `get_course_info` calls for the same ID happen, that's fine (just repeat the query). If two `delete_course` calls happen on the same ID, the second might return an error ("not found") – which might confuse the agent unless you handle it (maybe by not erroring if already deleted, or by returning a message "course already deleted"). There's a balance: you want errors to be known, but not cause the agent to spiral. For hack purposes, handle the obvious cases if you can.

- **Testing with AI in the loop:** If possible, once your MCP server is up, try it with an actual agent (Claude, ChatGPT dev, Cursor, etc.) by giving it tasks that involve the tools. See how it interprets the tool outputs. This can reveal if your outputs are too verbose or unclear. For instance, if your tool returns too much data (pages of JSON), the AI's context might get flooded or it might truncate. Perhaps your tool should support parameters like `limit` or provide summary options if data can be large. Agents have limited context windows.

MCP uses a three-tier architecture: the AI application (client/host), the MCP server, and the underlying API/service. The client and server perform a handshake where the server advertises its tools and their schemas [23] [24]. The AI then selects which tool to call based on its plan. Communication is standardized (JSON-RPC over HTTP or stdio), unlike arbitrary REST conventions [25].

This diagram illustrates the flow: The AI (client) says "what can you do?" → Server says "I can `search()`, `create()`, `update()` (for example)" [24] → AI chooses an action and the server executes it (calls the real API, gets result, returns to AI) [28]. The important thing as a developer is making sure that "menu of actions" is well-designed and each action is easy for the AI to use correctly.

Finally, consider security: By exposing tools to an AI, you trust the AI not to misuse them beyond intended purpose. If your MCP server connects to a sensitive system (like a database with PII), ensure you implement proper permission checks, or only run it in secure environments. Some MCP hosts (like Docker's toolkit) provide sandboxing, e.g., filesystem or network isolation [29], but your code is the last line of defense if the AI tries something clever. For example, if you had a tool that executes an arbitrary database query, an AI might chain something harmful. Best practice is to limit the tool to specific safe operations.

MCP Design (vs REST) Checklist

- [x] **Tool = single focused action:** No multi-purpose "kitchen sink" endpoints; one function per task for clarity.
- [x] **Clear naming and docs:** Tool names and docstrings succinctly describe the action and avoid ambiguity.
- [x] **Structured I/O schemas:** Inputs use appropriate types/enums; outputs are JSON-serializable structures. No freeform text unless necessary.
- [x] **No reliance on external documentation:** All needed info for usage is conveyed in the tool's schema (so the AI doesn't need a human to fill gaps).
- [x] **Idempotent or safe where possible:** Tools either don't have side effects or handle repeated calls gracefully (to the extent possible).
- [x] **Error messages meaningful:** Errors are informative but not overly verbose, and use a consistent format (inherited from JSON-RPC).
- [x] **Tested with agent if possible:** Verified that an AI can successfully use the tools as intended (making adjustments if the AI got confused).

Armed with these principles, let's focus on our specific integration: using a GraphQL API inside the MCP server, and how to design tools around it.

6. Integrating a GraphQL API within Your MCP Tools

In our example, the MCP server's primary job is to act as a middle layer to a GraphQL API (Canvas LMS). There are a few ways to expose GraphQL functionality through an MCP server. We'll outline some approaches and recommend a hackathon-friendly solution, then show a code snippet of how one of our tools calls the GraphQL API (which we actually already did in `get_course_info`).

Approaches to Exposing GraphQL via MCP

Option A: Generic GraphQL query tool. You could make a single tool in your MCP server, say `graphql_query(query: str, variables: dict) -> dict`, that takes an arbitrary GraphQL query string and returns the result. This essentially turns your MCP server into a pass-through to the GraphQL endpoint. While this gives maximum flexibility (the LLM can formulate any GraphQL query on the fly), it has drawbacks: - The AI would need to understand the GraphQL schema to use it effectively – possibly requiring you to expose the whole schema or some introspection. That can be huge (Canvas schema might be large), leading to **context explosion** issues [30]. Giving the full schema as part of the tool description is not feasible (too large). - There are security concerns: the AI could write queries that are too expensive or that fetch data it shouldn't. With arbitrary queries, you have less control. - It's also harder for the AI to know what's possible without schema; you could provide an introspection tool or documentation, but now you're basically writing a GraphQL client agent inside an AI – complex for a short project.

Option B: Specific high-level tools for key operations. This means identify the main things you want the AI to do, and implement one tool per GraphQL query or mutation. For example, `get_course_info(course_id)`, `list_courses(user_id)`, `update_course_title(course_id, new_title)` etc., each internally executes a known GraphQL query/mutation. This has several advantages: - **Simplicity for the AI:** The AI doesn't need to compose GraphQL; it just calls a descriptive function. The complexity of GraphQL is hidden behind your tool interface. - **Controlled scope:** You only implement what's needed, so you don't have to worry about it asking for something you didn't account for. - **Ease of parsing:** Since each tool returns a specific shape of data, you can post-process the GraphQL result to exactly the fields you want to return, making the AI's job easier. - **Security:** The AI can't do anything beyond the predefined queries. You won't accidentally allow a mutation that deletes data unless you explicitly provided a tool for it (and if you did, you likely wrap it with any necessary checks).

The downside is you lose some flexibility – if the AI user asks for something slightly outside what your tools cover, it can't directly satisfy it unless you implement a new tool or combine outputs. But in a hackathon, you likely have a narrow scope of tasks, so this is fine (even beneficial to keep scope in check).

Option C: Thin REST wrapper or alternative abstraction. One might consider standing up a REST API that wraps GraphQL calls (perhaps to simplify the query format), and then the MCP server calls that REST API. This is generally not worth it for a quick project – it adds another layer and you might as well call GraphQL directly. The only time this helps is if the GraphQL is very hard to call and you have an easier API for the same data; but if you had that, you might just use that API. For completeness, option C is essentially building a custom backend for your MCP server. Unless the GraphQL API is extremely unfriendly to use directly, skip this.

There is also an emerging pattern (as seen in some products like Grafbase's approach) which is a hybrid: provide a few generic GraphQL navigation tools like `search_schema(keyword)` and `introspect_type(name)` and `execute(query)` [31] [32]. This lets the AI do a controlled exploration of the GraphQL schema and then execute. This is quite advanced and not needed in a short timeframe, but it's an interesting approach that mitigates the context size problem by giving the AI ways to fetch just the relevant parts of the schema [33] [34]. We won't implement that here, but it's good to know such patterns exist for complex GraphQL integrations.

Recommendation for a hackathon: Go with **Option B** – specific tools for specific queries. Determine the key queries (or mutations) needed for your demo/story. Implement each as a separate function in your MCP server that calls the GraphQL API internally. This way you can be up and running quickly, and you have full control over what's happening.

Implementing a GraphQL-Backed Tool (Example)

We've already done this for `get_course_info`. Let's walk through it conceptually as a template:

1. **Define the function signature** to match the task. Use simple types for inputs (IDs, strings, etc.) and a structured output (e.g., dict). For example, `def get_course_info(course_id: str) -> dict:`.

2. **Inside the function, construct the GraphQL query string**. You might store it as a triple-quoted string for readability. Only query the fields you actually need for the output. In our case, we fetched

`name, description, startDate` of the course. If our output is just going to include those, perfect. (We returned the whole `data` which contains `course` object – arguably we could return `data['course']` directly to simplify, but either way.)

3. **Prepare variables and headers**. We put the `course_id` into a variables dict. Header includes the API key. Possibly also an `Accept` or content type header if required by API (GraphQL usually uses JSON and many accept just application/json by default).

4. **Make the HTTP request**. We used `requests.post`. One might use `requests.post(CANVAS_API_URL, json={...}, headers=...)`. Using the `json=` parameter automatically JSON-encodes the payload. Ensure you handle exceptions – if the network is down or the server returns non-JSON, `.json()` will throw, so catch exceptions as we did and raise as MCP errors.

5. **Check the GraphQL response for errors**. GraphQL responses always return HTTP 200 even if the query had errors, typically, and then put errors in the JSON. So you must inspect the returned JSON. We did:

   ```
   data = response.json()
   if 'errors' in data:
       raise RuntimeError(f"GraphQL error: {data['errors']}")
   ```

   Here we directly include the GraphQL error details. Depending on what they are (they might be complex objects), an alternative would be to extract the message: e.g., `msg = data['errors'][0].get('message')` and include that. But for debugging, printing the whole list might be okay. Just be mindful it could be verbose. In a hack, this is fine.

6. **Return the useful part of data**. GraphQL nest results under a `data` key (and then by field name). In our case `data` might be `{"course": {...fields...}}`. Returning the whole `data` is okay, but we could also return `data["course"]` to simplify. It depends on what you documented the tool does. If the tool is `get_course_info`, probably returning the course object itself is expected. The AI can handle either, but less nesting is easier. For demonstration, we left it as `data.get('data', {})` which actually returns the `data` dict, which contains `course`. Slight tweak: it might have been clearer to do `return data["data"]["course"]`. Feel free to adjust based on what you want the AI to see.

7. **Testing the tool in isolation**. You can simulate a call by calling the function in Python (just remember to set the env var first or modify the function to accept the API key for testing). Or use the FastMCP client as shown.

A second example tool might be similar. If we wanted `list_courses_for_user(user_id)`, we'd do a GraphQL query like:

```
query($user: ID!) {
  user(id: $user) {
    enrolledCourses { id, name, startDate }
  }
}
```

Then return the list of courses. Each tool would be its own query and likely different return shape.

For **mutations**, you'd use the `mutation { ... }` GraphQL syntax inside the query string. E.g., `mutation($course: ID!, $title: String!) { updateCourse(id:$course, title:$title) { success } }`. The handling is similar, but you might want to confirm success and return some status (maybe the updated object or a simple True/False).

One more tip: If the GraphQL API requires certain headers (like an `X-Request-ID` or specific content type), handle that in the request. Usually `requests.post(..., json=payload)` sets `Content-Type: application/json` automatically. Canvas might require an Authorization as we did. Some GraphQL endpoints might require a specific path (like `/api/graphql` vs `/graphql` – make sure the URL is correct for Canvas, from their docs). By using env for `CANVAS_GRAPHQL_URL`, we made it configurable.

Configuring GraphQL Endpoints and Auth

We've mostly covered this, but to reiterate the setup for GraphQL specifically:

- **Endpoint URL:** Determine the base URL for the GraphQL API. For Canvas, it could be something like `https://<institution>.instructure.com/api/graphql` (the dev docs would confirm). For hack/demo, if you don't have an actual endpoint handy, you could point to a mock server or skip actual calls and stub the response (but we assume you want it working). We put this in `CANVAS_GRAPHQL_URL` env with a default placeholder, so if someone uses a different Canvas instance, they can override. Many APIs (like GitHub's GraphQL) have a single known endpoint, so you might hardcode that (less need to configure).

- **Auth token:** As discussed, `CANVAS_API_KEY` is expected to be a token that the GraphQL API will accept. Ensure in documentation to tell users where to get that token (e.g., "Generate a Canvas API token from your profile settings"). Also, check if the GraphQL API expects the token as "Bearer <token>" (we assumed yes, which is common). If it was a query param or something, handle accordingly.

- **Permissions and scopes:** If the API token needs certain scopes to access data (for example, a Canvas token might need a scope to read courses), mention that. Otherwise, the user might set a token that's valid but get errors for unauthorized queries.

- **Time-outs and rate limits:** Possibly not a big concern for hack, but if GraphQL calls are slow, you might want to set a timeout on the request (requests allows a `timeout=seconds`). If the external API has strict rate limits, advise users to use their own token (so they don't share limits). In an

enterprise scenario, you'd implement backoff or queueing if the AI might fire many calls quickly. In our simple design, we assume a moderate usage.

Finally, think about error messaging specifically for GraphQL: We do a generic `"GraphQL error: {errors}"`. It might be nicer to translate common errors. For example, if a 401/403 happens, the GraphQL error might say "Not authorized". You could catch the HTTP status from `response.status_code` too. If `status_code != 200`, you know something is wrong (maybe token invalid or server down). You could raise `RuntimeError(f"GraphQL HTTP {response.status_code}: {response.text[:100]}")` if not 200. That might catch e.g. 401 not authorized (maybe the token is wrong) separately from GraphQL query errors which still return 200. For brevity, we didn't do that in code, but it's a consideration.

GraphQL Integration Checklist

- [x] **Chose an integration strategy:** (We chose specific tools for specific queries to keep things simple and safe [33].)
- [x] **GraphQL queries defined for each tool:** Each tool has a corresponding GraphQL query/mutation string that it executes internally.
- [x] **GraphQL endpoint configurable:** The base URL (and perhaps API version) can be adjusted via config if needed (default provided for convenience).
- [x] **API key used in headers:** The Authorization header or equivalent is correctly set with the user's API token.
- [x] **Error handling for GraphQL:** Checks for `errors` in response and HTTP status codes, converting them to meaningful errors for the MCP layer.
- [x] **Output post-processed if needed:** The data returned by GraphQL is trimmed or reformatted to only what's needed by the AI, eliminating extraneous fields if necessary.
- [x] **Tested with a real query:** (If possible) Ran the tool against a real GraphQL endpoint with a sample token to ensure the query is correct and the response is as expected. If no real API access, at least tested with a dummy response to ensure parsing logic works.

Now that we've covered everything from development to design, let's conclude with a couple of high-level checklists summarizing the process. These can serve as a quick reference to ensure you didn't miss a step when building and shipping your MCP server.

Quick Start Checklist

If you're building your own MCP server similar to the above, follow these steps:

1. **Plan the Tools**: Outline which functions (tools) you want to expose. Keep them focused (one per action) and define their inputs/outputs (with types).
2. **Set Up Project**: Initialize a Python project with FastMCP installed. Prepare a `requirements.txt` with `fastmcp` and any other needed libs (requests, etc.).
3. **Write MCP Server Code**: Create a file (e.g. `mcp_server.py`) and set up the FastMCP server instance. Implement each tool function with `@mcp.tool` decorator. Include docstrings and type hints. Fetch any needed config (API URLs, keys) from `os.environ`. In the `__main__` block, call

`mcp.run(transport="http", host="0.0.0.0", port=8000)` [5] to run the server on port 8000 for testing.

4. **Test Locally (Bare Metal)**: Export necessary env variables (like API keys) and run `python mcp_server.py`. Ensure the server starts with no errors. If possible, test a tool by using a FastMCP client call or even curl (it's JSON-RPC though, so client is easier). Fix any bugs.
5. **Dockerize**: Write a Dockerfile to containerize the app. Use a Python base image, install deps, copy code, and set the entrypoint to run the server. Build the image with `docker build -t yourname/yourtool:dev .`.
6. **Test Docker Locally**: Run the container with `docker run -p 8000:8000 -e API_KEY=... yourname/yourtool:dev` (replace API_KEY with your env name). Verify that the container starts and the server is reachable (maybe exec into it or check logs). Test a tool again if possible (similar to step 4 but via the container's network).
7. **Publish Image (Optional)**: Tag the image with your Docker Hub repo and push (`docker push`). Add a README on Docker Hub explaining usage (especially any required env vars).
8. **Integration with Client**: In your AI client (Claude, Cursor, etc.), configure the MCP server:
9. If using Docker Desktop MCP Toolkit, add the server from the catalog or by image name, set config (env var) in UI, and launch it. Then connect your AI client to the toolkit's gateway.
10. If using an AI platform that accepts a direct URL, provide `http://localhost:8000/mcp` (or the appropriate address/port) as a remote tool endpoint.
11. If using Claude Desktop without Docker toolkit, edit the config JSON to include your tool under `mcpServers` with the appropriate launch command. (For example, you could use a command to run your Docker container. But leveraging the Docker MCP Toolkit is simpler if available.)
12. **Run End-to-End**: Now ask your AI client to use the tool! For instance, in natural language: "What is the name and start date of course 42?" The AI (if properly set up) should invoke `get_course_info` with `course_id="42"`, get the result, and reply with the info. Iterate if the agent has trouble (maybe adjust docstrings or output format).
13. **Clean Up**: If handing off the project, include instructions for others to run (basically the above steps). Make sure no sensitive info is left in code or image.

This quick start ensures you go from zero to a functioning MCP server ready to assist an AI.

MCP Design Checklist

Before considering your MCP server "production-ready" (even if it's just for a hackathon demo), run through this design checklist to avoid common pitfalls:

- [ ] **Tools are AI-friendly:** Each tool does one thing, has a clear name and description, and uses simple inputs/outputs. The AI will immediately know which tool to use for a given goal (no ambiguous overlap between tools).
- [ ] **No unnecessary complexity:** If something can be done by combining two simpler tools, prefer that over one complex tool, so the AI can mix-and-match. (Remember, AI agents excel at sequencing tools when they're granular.)
- [ ] **Schema accuracy:** All input parameters have correct types (int vs str etc.), and output schema fields are named and structured logically. (If using FastMCP advanced features, maybe you defined dataclasses or Pydantic models – ensure they reflect reality of the data.)

- [ ] **Secrets handled properly:** No hardcoded secrets, and your server expects the user's credentials at runtime (env vars or config). Tested that providing no key results in a helpful error, and providing an incorrect key returns the external API's error (caught and conveyed clearly).
- [ ] **Robustness:** The server can handle unexpected inputs gracefully (returning an error rather than crashing). For example, if the AI passes an invalid ID or empty string, maybe your code handles it (or the GraphQL will error – which you handle).
- [ ] **Idempotent where possible:** Tools that modify state won't break things if called twice. Either they check state or it's naturally idempotent (like setting a property to the same value twice is fine).
- [ ] **Secure and least-privilege:** The tools only expose what's necessary. If your underlying API has a lot of power, you limit what the AI can do. For instance, if you have an admin token, be careful not to expose a tool that could dump all user data unless that's intended. In a hackathon, this might not be a big concern, but good to keep in mind.
- [ ] **Logged appropriately:** While you shouldn't log sensitive info, you might log important events (like "Called get_course_info for course 42"). In debugging AI-agent interactions, having some logs can help. Just ensure not to log full request data with secrets.
- [ ] **Tested with real AI agent:** Ideally, connect an AI (Claude, ChatGPT, etc.) to the MCP and try a realistic scenario. This often surfaces integration quirks (maybe the AI didn't call the tool because it didn't realize it should, meaning your tool description might need tweaking in prompt or config). Since this might be tricky during a hackathon if you lack access, at least mentally simulate how an AI would decide to use your tools given a user request.

By following this guide and the checklists, you should be able to design an MCP server that is **practical, secure, and effective** for augmenting an AI's capabilities. We built a concrete example that uses FastMCP, Python, and Docker – demonstrating how to go from local development to a shareable Docker image. The key is thinking in terms of **LLM agents**: provide them with the right abstractions and guardrails, and they can do amazing things by leveraging your MCP tools. Good luck with your hackathon project, and happy building!

1 16 13 24 25 33

---

1 MCP (Model Context Protocol) | Glossary | VeloDB
https://www.velodb.io/glossary/mc-1

2 3 fastmcp · PyPI
https://pypi.org/project/fastmcp/2.0.0/

4 Quickstart - FastMCP
https://gofastmcp.com/getting-started/quickstart

5 6 7 8 10 HTTP Deployment - FastMCP
https://gofastmcp.com/deployment/http

9 27 Using MCP with OpenAI & MCP Servers | by Cobus Greyling | Medium
https://cobusgreyling.medium.com/using-mcp-with-openai-mcp-servers-1c479b5dc8a2

11 12 Docker MCP Catalog | Discover Secure, Top MCP Servers
https://hub.docker.com/mcp

13 14 15 Introducing Docker Hub MCP Server | Docker
https://www.docker.com/blog/introducing-docker-hub-mcp-server/

16 My Claude Workflow Guide: Advanced Setup with MCP External Tools : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1ji8ruv/my_claude_workflow_guide_advanced_setup_with_mcp/

17 Hub MCP server - Docker Docs
https://docs.docker.com/ai/mcp-catalog-and-toolkit/hub-mcp/

18 19 29 MCP Toolkit | Docker Docs
https://docs.docker.com/ai/mcp-catalog-and-toolkit/toolkit/

20 MCP's Quiet Crisis | How Credentials Leak in Plain Sight - Cyata
https://cyata.ai/blog/whispering-secrets-loudly-inside-mcps-quiet-crisis-of-credential-exposure/

21 22 23 24 25 26 28 Model Context Protocol (MCP) vs. APIs: Architecture & Use Cases | Codecademy
https://www.codecademy.com/article/mcp-vs-api-architecture-and-use-cases

30 31 32 33 34 Solving context explosion in GraphQL MCP servers – Grafbase
https://grafbase.com/blog/managing-mcp-context-graphql