ChatGPT

Complete Installation and Usage Guide for Hanzo MCP (hanzoai/mcp)

This guide provides a step-by-step walkthrough for installing and using the **Hanzo MCP** project in an existing directory named `hanzo-mcp` using a Python virtual environment (venv). It covers prerequisites, detailed installation steps, configuration (including environment variables), compatibility notes for GPT-4.1 and Claude, an overview of MCP's features and tools, and known issues (especially on Windows/WSL and GPU/CUDA considerations). All instructions prioritize technical accuracy and reflect the official documentation and best practices in the ML environment.

Prerequisites and System Dependencies

Before installation, ensure your system meets the following requirements:

- **Operating System:** Linux or macOS is recommended for smooth operation. Windows users should preferably use **WSL (Windows Subsystem for Linux)** to run the MCP server. *Running the MCP natively on Windows can be problematic; using WSL to host the server and letting the client (e.g. Cursor or Claude Desktop) communicate with it is a proven approach* [1].
- **Python Version:** Python **3.12** or higher is required [2]. (The project takes advantage of features in Python 3.12+, so ensure you have the correct Python version installed.)
- **Git:** Ensure Git is installed (needed to clone the repository).
- **Build Tools:** For certain Python packages (if any require compilation), having a C/C++ build environment is recommended (e.g. `build-essential` on Debian/Ubuntu, or Xcode Command Line Tools on macOS).
- **Virtual Environment Tool:** While not strictly required (you can use the built-in `venv` module), having tools like `venv` or `virtualenv` will help isolate the Python environment for this project.
- **CUDA (Optional):** If you plan to run large language models locally on an NVIDIA GPU, install appropriate NVIDIA drivers and CUDA toolkit. This is not required if using only remote API models (OpenAI or Anthropic), but is necessary for local model acceleration.
- **DeepSpeed (Optional, for advanced use):** If you intend to work with very large local models or multi-GPU setups, consider installing **DeepSpeed** for optimization. DeepSpeed is a library that can partition model inference across multiple GPUs to handle models larger than a single GPU's memory, enabling huge models to be loaded across GPUs [3]. This is only needed if you deploy your own large models (e.g. a local GPT-4-scale model) in conjunction with Hanzo MCP – it is *not* needed for using OpenAI's GPT-4 or Anthropic's Claude via their APIs.

Installation in the `hanzo-mcp` Directory (Step-by-Step)

Follow these steps to install Hanzo MCP into an existing folder named `hanzo-mcp`, using a Python virtual environment for isolation:

1. **Create or Navigate to the Target Directory:**
   If you already have an empty directory called `hanzo-mcp`, navigate into its parent directory. Otherwise, decide on a location and create the `hanzo-mcp` directory.

```
# If the directory doesn't exist, create it:
mkdir hanzo-mcp
cd hanzo-mcp
```

Note: The instructions will assume this directory will contain both the code and a virtual environment.

2. **Clone the GitHub Repository:**
   Use Git to clone the official repository into the `hanzo-mcp` folder. You can clone directly into the existing directory by specifying `.` as the target (assuming it's empty), or clone outside and move the contents. For clarity, here's how to clone directly into the directory:

```
git clone https://github.com/hanzoai/mcp.git ./
```

   This will download the project's code into `hanzo-mcp`. After cloning, verify that the directory now contains the project files (you should see files like `README.md`, `setup.py`, etc.).

3. **Create a Python Virtual Environment:**
   Inside the `hanzo-mcp` directory, create a virtual environment to install the project's Python dependencies. For example:

```
python3.12 -m venv venv
```

   This creates a subdirectory `venv` (you can name it differently if desired) containing a clean Python environment.

4. **Activate the Virtual Environment:**
   Activate the venv before installing packages:

5. On **Linux/Mac**:

```
source venv/bin/activate
```

6. On **Windows (WSL or CMD)**:

```
source venv/bin/activate
```

   (If using PowerShell, use `venv\Scripts\Activate.ps1`*.)*
   Once activated, your shell prompt may prefix with the environment name, indicating that the venv is active.

7. **Upgrade pip (optional but recommended):**
   It's good practice to ensure you have an up-to-date pip inside the venv:

```
pip install --upgrade pip
```

8. **Install Hanzo MCP and its Python Dependencies:**
   You have two main options to install the project in your venv:

9. **Option A: Use PyPI release** – Install the latest stable release from PyPI:

   ```
   pip install hanzo-mcp
   ```

   This will fetch the package and install all required dependencies automatically. The package is published under the name `hanzo-mcp` on PyPI (current latest version 0.5.0 as of June 2025) [4].

10. **Option B: Install from the cloned source** – If you prefer using the source you just cloned (for development or to ensure you have the absolute latest code), run an editable install:

    ```
    pip install -e .
    ```

    This tells pip to install the package in "editable" mode from the current directory. It will read the `pyproject.toml` / `setup.py` and install all dependencies listed. (Alternatively, you could run `pip install .` for a standard install from source.)

Both approaches will pull in the necessary Python libraries. The core dependencies (as defined by the project) will include things like **LiteLLM** (for LLM integration), libraries for file and notebook handling, etc., so you don't need to install those separately – pip will handle it. For reference, the project supports extra dependency groups named "dev", "test", "performance", etc., which you usually don't need unless you have those specific needs [5]. The standard installation covers all runtime requirements.

1. **Verify Installation:**
   After installation, you should be able to run the `hanzo-mcp` command. For example:

   ```
   hanzo-mcp --version
   ```

   This isn't explicitly documented in the README, but many Python CLI tools support a `--version` flag. If one is not available, you can simply try running `hanzo-mcp` without arguments to see if it starts (and likely prints usage help or starts the server). If the command is not found, ensure your venv is activated and that the install step succeeded.

At this point, **Hanzo MCP is installed** in your `hanzo-mcp` directory and isolated in the venv. Next, we configure it for use with your LLMs and tools.

Configuration and Environment Setup

Depending on how you plan to use Hanzo MCP (with Claude Desktop, with ChatGPT GPT-4.1, or other clients), you may need to configure a few settings:

- **Claude Desktop Integration:** If you are using **Claude Desktop**, Hanzo MCP can install itself into Claude Desktop's MCP configuration. Simply run:

```
hanzo-mcp --install
```

from within the activated environment. This command will perform any necessary setup so that Claude Desktop recognizes the new MCP server. According to the documentation, this command installs a configuration entry for Hanzo MCP with default settings into Claude Desktop [6] [7]. After running it, **restart Claude Desktop**. You should then see an entry named "hanzo" (or a custom name if set) in Claude Desktop's MCP server selector [8]. Selecting it will allow Claude to use this MCP server for enhanced capabilities.
Behind the scenes: the `--install` likely writes to Claude Desktop's MCP config (on Windows or Mac, a config file or registry) so that Claude knows how to launch the server. The default server name is "hanzo".

- **Allowed Paths (File Access Restrictions):** By default, Hanzo MCP restricts file operations to certain directories for security. You can customize these allowed paths. For example, if you want the MCP to have access to specific project folders, you can specify them during installation. Using the Makefile approach (for development install), they demonstrate:

```
make install-desktop ALLOWED_PATHS="/path/to/projects,/another/path"
SERVER_NAME="hanzo-dev"
```

In a manual install context, you can achieve the same by setting environment variables or editing config. For instance, you might set an environment variable `ALLOWED_PATHS="/path1,/path2"` before running `hanzo-mcp --install` to embed those paths. This ensures the MCP server only allows file operations in the specified directories [9]. If not set, a default (often the current directory or a safe subset) is used.

- **Custom Server Name:** Similarly, you can customize the server's name as it appears in clients. Setting `SERVER_NAME="hanzo-dev"` (or another name) will register it under that name instead of the default "hanzo" [9]. This is useful if you run multiple MCP servers or want a descriptive name.

- **Disabling Write Operations:** For safety, you may choose to run in a read-only mode (no file modifications). You can configure `DISABLE_WRITE=1` to turn off tools that write or modify files [10]. In practice, this could be set as an env var or via a flag if available. With writes disabled, the MCP will refuse any tool invocation that attempts to alter the filesystem, effectively making it a read-only assistant (you might prefer this if you want to review changes in your IDE instead).

- **API Keys and LLM Provider Setup:** Hanzo MCP itself doesn't ship with an AI model – instead it *connects to external LLMs* (Large Language Models). It supports multiple LLM providers through the **LiteLLM** interface [11]. To use **OpenAI GPT-4.1** or **Anthropic Claude** with MCP, you need to provide API access to those models:

- **OpenAI (GPT-4.1):** Ensure you have an OpenAI API key. Set the environment variable `OPENAI_API_KEY` with your key (e.g., in your shell or a `.env` file that you source before running the server). Hanzo MCP (via LiteLLM) will detect this and use OpenAI's GPT models for its reasoning and task execution. By default it may use `gpt-4` as the model if available, or you might be able to specify a model name via an environment variable or config (check LiteLLM docs for model selection). If you are running a self-hosted equivalent of GPT-4.1, configure it similarly to mimic the OpenAI API.

- **Anthropic (Claude v1 or v2):** Ensure you have an Anthropic API key for Claude. Set the `ANTHROPIC_API_KEY` environment variable. Additionally, you might need to indicate the provider; some setups use an `API_PROVIDER` variable. For example, in related Hanzo projects (like "Overlord"), they use:

  ```
  export API_PROVIDER=anthropic
  export ANTHROPIC_API_KEY=<your_key>
  ```

  This signals to use Claude via Anthropic's API [12]. Similarly, for OpenAI one could do `API_PROVIDER=openai` with the OpenAI key. Check the Hanzo MCP documentation or LiteLLM configuration if unsure – but typically, simply having the key exported is enough, as the library will choose the provider based on which key is present.

- **Local Models:** If instead of API services you plan to use a local model (via Hugging Face transformers or others), additional configuration may be needed (e.g., model path, tokenizer, etc.). Hanzo MCP can work with any **LiteLLM-compatible** model [11], which includes local models loaded through a LiteLLM proxy. This advanced scenario might involve running a LiteLLM proxy server pointing to a model or using a local inference library. In such cases, consult LiteLLM's documentation for how to register your model and point Hanzo MCP to it. (Optional dependencies like PyTorch, Transformers, or DeepSpeed would come into play here.)

- **Starting the MCP Server:** In many cases, you do not need to manually start the server; the client (Claude Desktop, Cursor, ChatGPT, etc.) will launch it when needed via the configured command. For example, Claude Desktop will use the config from `hanzo-mcp --install` to start the server in the background when you select it. However, you can also run the MCP server by invoking the command directly. If you simply run:

  ```
  hanzo-mcp
  ```

  without arguments, the server should start listening (commonly on a localhost port or a socket). You might see log output indicating it's ready. By default, it likely runs until stopped (Ctrl+C). If integrating with ChatGPT (GPT-4.1 with MCP support), you might need the server running in the background and provide its address to ChatGPT. (For OpenAI's ChatGPT integration, you would typically register a remote MCP endpoint via their interface – beyond the scope of this guide – but know that the Hanzo MCP server adheres to the standardized **Model Context Protocol**, meaning GPT-4.1's agent can interface with it if configured.)

Tip: To persist environment variables (like API keys or allowed paths), you can create a `.env` file in the `hanzo-mcp` directory and use a tool like `dotenv` or simply source the file in your shell before starting the server. This way, whenever the server runs (whether manually or launched by a client), it has the necessary environment context.

Overview of Hanzo MCP's Features and Tools

Hanzo MCP is essentially an **MCP server** that exposes a wide range of development and filesystem operations to an AI agent (Claude, GPT-4, etc.) in a controlled manner [13] [14]. Once installed and connected, it enables the AI to perform various actions like reading files, modifying code, running

commands, analyzing the project structure, and more – all under your supervision. Below is an overview of its key capabilities:

- **Code Understanding and Analysis:** The AI can **read and understand code** by accessing files in your project, searching for patterns, and identifying key structures. It can analyze the project's architecture, dependencies, and frameworks to get context [15] [16]. (This is facilitated by tools like `read_files`, `search_content`, `project_analyze_tool`, etc. described later.)

- **Code Modification:** The AI can make **targeted changes to your code or text files**. It respects a permission system (usually asking for confirmation) before applying changes. This includes editing lines in a file, creating new files, or replacing content [17] [18]. Tools such as `write_file`, `edit_file`, and `content_replace` implement these functions.

- **File Operations:** Beyond editing, the AI can perform general file operations: listing directory trees, getting file info (size, type), and managing files (moving, renaming) – all within the allowed sandbox paths [19] [18]. The `directory_tree` and `get_file_info` tools support these read-only inquiries, while `run_command` can handle moves/renames (since it allows shell commands, under restrictions).

- **Enhanced Command Execution:** Hanzo MCP allows the AI to execute shell commands in a controlled way. There is a `run_command` tool for simple shell commands (like compiling code, running tests, or creating directories) and a `run_script` / `script_tool` for running code in specific languages or scripts [20] [21]. These are executed with safety in mind (no unlimited access beyond allowed paths, and with user confirmation for dangerous operations). It also features improved error handling and support for shell environment, so the AI sees command outputs and errors.

- **Jupyter Notebook Support:** A standout feature is support for Jupyter notebooks. The AI can read notebooks (`read_notebook` tool) which extracts all code cells (and can also read outputs if needed) [22]. It can also edit notebooks (`edit_notebook` tool) to insert, modify, or delete cells [22]. This is especially useful if you work with `.ipynb` files; the AI can help refactor a notebook or experiment with code within it, treating the notebook structure properly.

- **Agent Delegation and Concurrency:** Hanzo MCP supports launching sub-agents for parallel tasks via the `dispatch_agent` tool [23]. This means the AI can spawn multiple helper agents (with read-only access) to concurrently gather information – for example, scanning multiple files in parallel for a pattern – and then aggregate results. This delegation improves efficiency for large projects.

- **"Think" Space:** A special tool named `think` provides the AI a scratchpad to perform complex reasoning without affecting files [24]. When invoked, it essentially asks the AI to reason or plan in a contained manner. This is useful for step-by-step problem solving or code analysis that shouldn't immediately change anything. It's like asking the assistant to "think out loud" in a structured way.

- **Security and Permissions:** Throughout all features, security is a priority. The MCP server implements permission prompts (it can ask the user for confirmation before a file is overwritten or a command executes), enforces the **allowed paths whitelist**, and sanitizes inputs to commands [25]. This ensures that even though the AI has powerful abilities, it operates within boundaries you set. For instance, by default it cannot wander outside your project directories,

create arbitrary network requests, or execute certain dangerous commands unless explicitly permitted.

Core Tools Implemented

For a more technical breakdown, the following table lists the **essential MCP tools** provided by Hanzo MCP and their functions (these are the "functions" or API calls the AI agent can use via MCP):

Tool Name	Description
`read_files`	Read one or multiple files. Returns file contents with automatic encoding detection [26]. Useful for allowing the AI to inspect code or text.
`write_file`	Create a new file or overwrite an existing file with given content [27]. Will usually prompt for confirmation if overwriting.
`edit_file`	Edit an existing text file *in-place*, with line-level precision [28]. The AI can specify which lines to change (add, modify, delete). This ensures structured modifications rather than blind overwrite.
`directory_tree`	Retrieve a recursive directory listing (tree view) of a given folder [29]. Allows the AI to understand project structure (files, directories).
`get_file_info`	Get metadata about a file or directory (size, modification date, etc.) [30]. No content read, just info.
`search_content`	Search for a text pattern across files' contents [31]. The AI can find where a function is referenced or a keyword appears in the project. Supports regex or plaintext search.
`content_replace`	Find and replace a pattern in files [31]. This tool can perform project-wide replacements, with safeguards (it likely shows a diff or requires confirmation for multiple changes).
`run_command`	Execute a shell command [32]. This is quite powerful: it can be used to run build tools, git commands, create directories, list directory contents (as an alternative to `directory_tree`), etc. Output and errors are captured and returned to the AI. By design, this is restricted to the allowed paths and certain safe commands (and will prompt for confirmation on risky actions).
`run_script`	Run a script with a specified interpreter [33]. For example, run a Python script, a Bash script, etc., by providing the code. This allows multi-line or more complex execution than a one-liner shell command, in a controlled interpreter environment.
`script_tool`	Alias or variant of running code in specific languages [33]. It may allow selecting a language (like `script_tool("python", code)`) which behind the scenes likely uses `run_script`.

Tool Name	Description
`project_analyze_tool`	Analyze the project's structure and dependencies [34]. This might gather info like what frameworks are used (e.g., detect if it's a Django project, or what libraries appear in requirements), how files relate, etc., and summarize for the AI. This helps the AI get context on the project as a whole.
`read_notebook`	Read all code cells from a Jupyter Notebook (`.ipynb`) including outputs [35]. Converts the notebook to a linear text that the AI can parse.
`edit_notebook`	Modify a Jupyter Notebook by adding, removing, or editing cells [35]. Enables the AI to refactor a notebook or inject new analysis steps.
`think`	A no-effect tool for complex reasoning [24]. The AI uses this to "think" or plan without making changes. You might see it output a chain-of-thought here which isn't directly executed.
`dispatch_agent`	Launch one or more read-only sub-agents to perform tasks concurrently [23]. Useful for parallelizing work (e.g., scanning multiple files at once). The main agent can then incorporate their results.

(The above information is summarized from the Hanzo MCP documentation and README [36] [37].)

These tools cover virtually all filesystem and code-manipulation needs, essentially turning Claude or GPT-4 into a capable pair programmer with direct access to your project (under your control). You can interact with these tools through natural language prompts: for example, "Find all TODO comments in the repo" would cause the AI to use `search_content`, or "Open the configuration file and change the API endpoint URL to X" might use `edit_file` or `content_replace`. The beauty of MCP is you often don't have to explicitly call the tool; you request the task and the AI agent decides which tool to invoke. However, you **can** explicitly tell the agent to use a tool by name if needed (for instance, *"Use the* `directory_tree` *tool to show me the folder structure."*).

CLI and API Usage

Using the MCP via CLI: While most interactions happen through the AI client, you can directly invoke some MCP commands for testing. The `hanzo-mcp` CLI may support subcommands; one we know is `--install` (for Claude Desktop). Another likely one is something like `hanzo-mcp --serve` or just `hanzo-mcp` to start the server. You might run `hanzo-mcp --help` to list available CLI options. If a help menu appears, it will detail any other flags (such as specifying a config file, host/port for the server, etc.). For instance, some MCP servers allow a `--port <number>` argument. Check the docs (`docs/` directory) if you need advanced usage of the CLI.

Using the MCP via API/Clients: Once the server is running, it listens for MCP client connections. There are a few ways this occurs: - **Claude Desktop:** Select the "hanzo" server in the UI. From then on, when you ask Claude to do something involving tools, Claude will route tool usage to this MCP. E.g., you type in Claude: *"Open the file* `app.py` *and fix the function according to the error message,"* Claude's agent will call `read_files` on `app.py` via Hanzo MCP, get the content, then possibly call `edit_file` with changes, etc. The results (like file diffs or command outputs) will be sent back for Claude to summarize or confirm with you.

- **ChatGPT (GPT-4.1) via OpenAI's MCP integration:** If you have ChatGPT Enterprise or any setup where GPT-4.1 can connect to a custom MCP server, you would register this Hanzo MCP as a remote server. (OpenAI's documentation describes how to add an MCP server to ChatGPT [38].) Typically, you'd provide an endpoint URL or some identifying info so ChatGPT can reach it. Ensure the Hanzo MCP server is accessible (if ChatGPT is cloud-based, your server might need to be hosted or tunneled). Once connected, GPT-4.1 will use it similarly to Claude.
- **Cursor IDE or Other Dev Tools:** "Cursor" is an AI-assisted IDE that also uses MCP servers. You could integrate Hanzo MCP globally or per-project. For example, adding an entry in `~/.cursor/mcp.json` pointing to the `hanzo-mcp` command (or to a script that launches it) would make it available in all projects [39] [40]. The Cursor community has shared that running MCP servers in WSL and connecting the Cursor Windows app to it works (as mentioned earlier). So if you're coding in Cursor, Hanzo MCP can be your backend for power tools.

Regardless of client, **the usage pattern is**: you issue natural language requests, and behind the scenes the AI chooses from those **Tools** listed above to satisfy the request. You'll often get a response that includes tool outputs. For instance, asking for a directory listing might result in the AI responding with a markdown-formatted tree of your project (obtained via `directory_tree`), or asking to run tests could yield the test results output (from `run_command`).

Important: Always review the AI's proposed actions when it's about to write files or run commands. Hanzo MCP's permission prompts will usually halt execution and ask you (in the chat interface) to approve an action, especially if it's potentially destructive (like deleting a file or executing an unknown script). This is a critical safety feature to prevent unintended changes [25]. As the user, you should carefully read those and only approve if it looks correct. You maintain full control.

Compatibility Notes for GPT-4.1 and Claude

Hanzo MCP was **designed with Claude in mind** and is fully compatible with Anthropic's Claude models (especially through Claude Desktop) [13]. It effectively turns Claude into a coding assistant with direct file access. In practice, Claude has been used extensively with this MCP for tasks like code refactoring, documentation generation, debugging, etc., and the integration is smooth. If you have Claude (via Claude Desktop or an API), you can leverage the entire feature set described above. The README explicitly mentions *"Claude Desktop integration"* as a primary use case [41].

For **OpenAI's GPT-4.1**, compatibility is achieved through the Model Context Protocol standard. OpenAI introduced MCP support to allow ChatGPT (especially the GPT-4 model) to use tools in a fashion similar to plug-ins. Hanzo MCP implements the standard, so GPT-4 can interface with it just as Claude does. In practical terms: - If you are using ChatGPT with an MCP connector (available in certain versions of ChatGPT or via the OpenAI API with function calling), you can register the MCP server. Once linked, GPT-4 will know what tools are available (the server advertises its tool list and schemas to the client on connect). The set of tools (read_files, etc.) will appear to GPT-4, and it can invoke them. - Ensure the **API keys/env for OpenAI** are set, as described earlier, so that the Hanzo server itself can use GPT-4 as the underlying model for reasoning. If you are using ChatGPT's brain to do the reasoning, then the Hanzo MCP doesn't need OpenAI credentials (ChatGPT is doing the thinking, MCP is just the executor). However, if you use Hanzo's internal agent capabilities (it mentions possible use of any LiteLLM model), you might also configure GPT-4 API for that scenario. Clarify which role GPT-4 is playing: *client (brain) or server (execution)* – typically, GPT-4 (ChatGPT) is the brain and Hanzo MCP is just executing. In Claude's case, Claude is the brain, MCP executes. - In summary, **GPT-4.1 works with Hanzo MCP** either as a driving agent or as a model behind the scenes. The interoperability is a core goal of MCP – it's model-

agnostic. Users have reported success hooking it up such that GPT-4 can perform the same kind of code-assistance tasks as Claude using this server.

One caveat: if both Claude and GPT-4 are used simultaneously with the same server, be mindful of any concurrency issues or state. For instance, if Claude Desktop and ChatGPT are both pointed at your one Hanzo MCP instance, they might conflict (both trying to use the tools at once). It's possible to run multiple instances (with different ports or names) to keep them separate. Or simply use them one at a time.

Known Issues, Bugs, and Platform-Specific Considerations

Finally, let's cover some known issues and important considerations, especially related to running on Windows, using WSL, and working with GPU acceleration:

- **Windows and WSL:** Hanzo MCP is cross-platform (*"Operating System :: OS Independent"* per PyPI [42]), but in practice, certain tools (particularly shell commands and file paths) assume a POSIX environment. **Using WSL on Windows is highly recommended.** Directly running the MCP on Windows may cause issues with path formats, permissions, or the ability of the AI to execute certain commands. In fact, the Cursor community discovered that *"Cursor can't run Claude's coding abilities on bare Windows – the solution is to run the MCP server inside WSL and let the Windows client talk to it"* [1]. This setup avoids many headaches: the MCP thinks it's on Linux (smooth operation), and your Windows AI app connects over localhost networking. If you use Claude Desktop on Windows, you can configure the MCP entry to call `wsl.exe` and launch `hanzo-mcp` in WSL automatically [43]. There are guides and even YouTube videos on connecting WSL MCP servers to Claude Desktop [44].
  Common issues addressed by WSL: file path translation (Windows vs Unix paths), case sensitivity, and availability of typical shell utilities. Also, certain security features (like Unix file permissions) exist in WSL but not on plain Windows. So WSL provides a more consistent environment for the server.

- **Docker as an Isolation Layer:** Some users opt to run MCP servers in Docker containers for added security (as noted in a blog post [45] [46]). The idea is to bind-mount only specific directories into the container, and even run as a non-root user, so that even if something went wrong, the AI's reach is contained. If you are on Windows, you can combine this with WSL (Docker via WSL2 backend) to good effect. The official instructions mention a Docker setup for a filesystem MCP; with Hanzo MCP, you'd have to create your own Dockerfile (though the repository might include one for development). This is an advanced setup, but worth mentioning for enterprise or highly secure use cases.

- **Permission Prompts and IDE interference:** Sometimes the AI might appear to hang – often it's waiting for you to permit an action (especially in Cursor or Claude Desktop, a popup or dialog might ask for confirmation). If you encounter a stall after the AI says "I will now apply the changes," check for a prompt. In Cursor, for example, a little overlay may appear asking for approval.

- **Large Outputs:** Tools like `directory_tree` on a big project, or reading a large file, can produce a lot of output. Some clients have trouble handling very long messages. Claude is pretty good at large context, but ChatGPT may truncate. Consider scoping requests (e.g., ask for a subtree of the directory, or a specific section of a file) to avoid hitting length limits. This is not a bug per se in MCP, but a reality of the clients.

- **DeepSpeed and Local Model Performance:** If you use a hefty local model (for example, a 30B or 70B parameter model) with this setup, you might face slow responses or out-of-memory errors. **DeepSpeed** can alleviate some of this by partitioning the model across GPUs and using optimizations (e.g., ZeRO redundancy) [3] . Installing Hanzo MCP with the `performance` extra (if provided) could pull in DeepSpeed or related libraries. Keep in mind that using DeepSpeed also requires a compatible PyTorch installation and matching CUDA toolkit. Ensure your GPU drivers, CUDA, and PyTorch are all aligned (e.g., for CUDA 11.x). Also note that DeepSpeed mainly helps with *inference/training* of models; it doesn't directly impact the MCP's own functions except insofar as it enables the AI model to run smoother. If your usage is through OpenAI/Anthropic APIs, DeepSpeed is not applicable.

- **CUDA Toolkit on Windows:** If for some reason you are running a local model on Windows (perhaps via WSL2 GPU passthrough or Windows native with PyTorch DirectML), be aware of compatibility issues. Many ML frameworks have less support on Windows. WSL2 with NVIDIA's CUDA integration is generally effective, but some extra setup (drivers on Windows side + the `cuda` package in Ubuntu) is needed. This is beyond Hanzo MCP itself, but affects it if the model backend is local.

- **Bugs and Updates:** At the time of writing, the Hanzo MCP project is active (the release history shows frequent updates in April-June 2025 [47]). Keep your installation updated to benefit from bug fixes. You can upgrade via pip (`pip install -U hanzo-mcp`). Known issues tend to be discussed in the project's GitHub issues (there were none open as of now [48]) or in community forums (like the MCP subreddit or Cursor forum). For example, if a particular tool isn't working as expected (say, `edit_notebook` misordering cells), check for updates or raised issues.

- **Anthropic/Claude Rate Limits:** If using Claude via Claude Desktop, you're limited by whatever allowance that app has (likely the desktop app manages that). If using Claude via API, note the context length (Claude 2 supports very large contexts ~100K tokens, which is great for big codebases). GPT-4 currently has smaller context (8K or 32K variants). This means Claude might handle a `read_files` of an entire large codebase better than GPT-4 can. Be mindful of these differences. It's not an MCP bug, just a model constraint. A best practice is to allow the AI to summarize or focus rather than always reading huge files in full.

- **Claude vs GPT Behavior:** In use, you might notice Claude is more laissez-faire (it might attempt larger operations in one go) whereas GPT-4 might be more step-by-step. This is just their style; the MCP will accommodate either. From a compatibility standpoint, both are supported equally by the MCP server.

- **Model Cost and Keys:** Remember that when the AI reads files or executes commands, it may send a lot of text back to the model (especially file content for analysis). If you're using API keys (OpenAI/Anthropic), those count towards your usage. Large files = large prompts. Use the tools wisely to avoid excessive token usage. Also ensure your keys are kept secret – do not inadvertently share logs that contain them.

In summary, Hanzo MCP is a powerful bridge between AI and your development environment. By following this guide, you should have it installed in your `hanzo-mcp` directory with a working virtual environment, configured to work with your chosen AI models. You have an understanding of its capabilities and how to invoke them via your AI assistant. Keep security in mind, use WSL for Windows, and update configuration as needed for your setup. With GPT-4.1 or Claude harnessed through Hanzo MCP, you can dramatically boost your productivity in managing and improving code. Happy coding!

Sources:

- Hanzo MCP GitHub README and documentation [13] [17] [49] [31] [24] (features and tools overview)
- Hanzo MCP PyPI information [2] [26] [35] (requirements and tool descriptions)
- Claude Desktop integration steps from README [41] [50]
- Playbooks summary of Hanzo MCP features [14] [51] [18] [25]
- Community discussions on Windows/WSL usage [1] [46] [43] (Windows compatibility and WSL solution)
- DeepSpeed usage note from Hugging Face blog [3] (for large model support)
- Overlord project example (Hanzo's related project) for API key config [12] (Anthropic API key usage).

---

[1] I figured out how to run claude code natively on Windows as Cursor ...
https://www.reddit.com/r/ClaudeAI/comments/1lc6u5k/i_figured_out_how_to_run_claude_code_natively_on/

[2] [4] [5] [11] [26] [27] [28] [29] [35] [42] [47] hanzo-mcp·PyPI
https://pypi.org/project/hanzo-mcp/

[3] DeepSpeed - Hugging Face
https://huggingface.co/docs/accelerate/v0.16.0/en/usage_guides/deepspeed

[6] [7] [8] [14] [15] [16] [18] [21] [25] [39] [40] [51] Hanzo MCP server for AI agents
https://playbooks.com/mcp/hanzo-platform

[9] [10] [13] [17] [19] [20] [22] [23] [24] [30] [31] [32] [33] [34] [36] [37] [41] [48] [49] [50] GitHub - hanzoai/mcp: Hanzo MCP: AI + Hanzo Dev exposed via MCP. Enables standardized context exchange across Dev tools, agents, and models. Core to Hanzo's interoperable AI orchestration framework.
https://github.com/hanzoai/mcp

[12] GitHub - hanzoai/overlord: AI Overlord, managing your disparate agents through local computer use.
https://github.com/hanzoai/overlord

[38] Hanzo API
https://api.hanzo.ai/

[43] [45] [46] How to use the filesystem mcp with WSL and Docker
https://browniantech.com/blog/post/How-to-use-the-filesystem-mcp-with-WSL-and-Docker

[44] Connect WSL MCP Servers To Claude Desktop on Windows
https://www.youtube.com/watch?v=oNc8WsQLphY