Tiny Agents in Python: an MCP-powered agent in ~70 lines of code

Published May 23, 2025


Célina Hanouti
celinah Follow


Julien Chaumond
julien-c Follow


Lucain Pouget
Wauplin Follow


shaun smith
evalstate Follow
guest

Inspired by Tiny Agents in JS, we ported the idea to Python 🐍 and extended the `huggingface_hub` client SDK to act as a MCP Client so it can pull tools from MCP servers and pass them to the LLM during inference.

MCP (Model Context Protocol) is an open protocol that standardizes how Large Language Models (LLMs) interact with external tools and APIs. Essentially, it removed the need to write custom integrations for each tool, making it simpler to plug new capabilities into your LLMs.

In this blog post, we'll show you how to get started with a tiny Agent in Python connected to MCP servers to unlock powerful tool capabilities. You'll see just how easy it is to spin up your own Agent and start building!

> *Spoiler* : An Agent is essentially a while loop built right on top of an MCP Client!

How to Run the Demo

This section walks you through how to use existing Tiny Agents. We'll cover the setup and the commands to get an agent running.

First, you need to install the latest version of `huggingface_hub` with the `mcp` extra to get all the necessary components.

```
pip install "huggingface_hub[mcp]>=0.32.0"
```

Now, let's run an agent using the CLI!

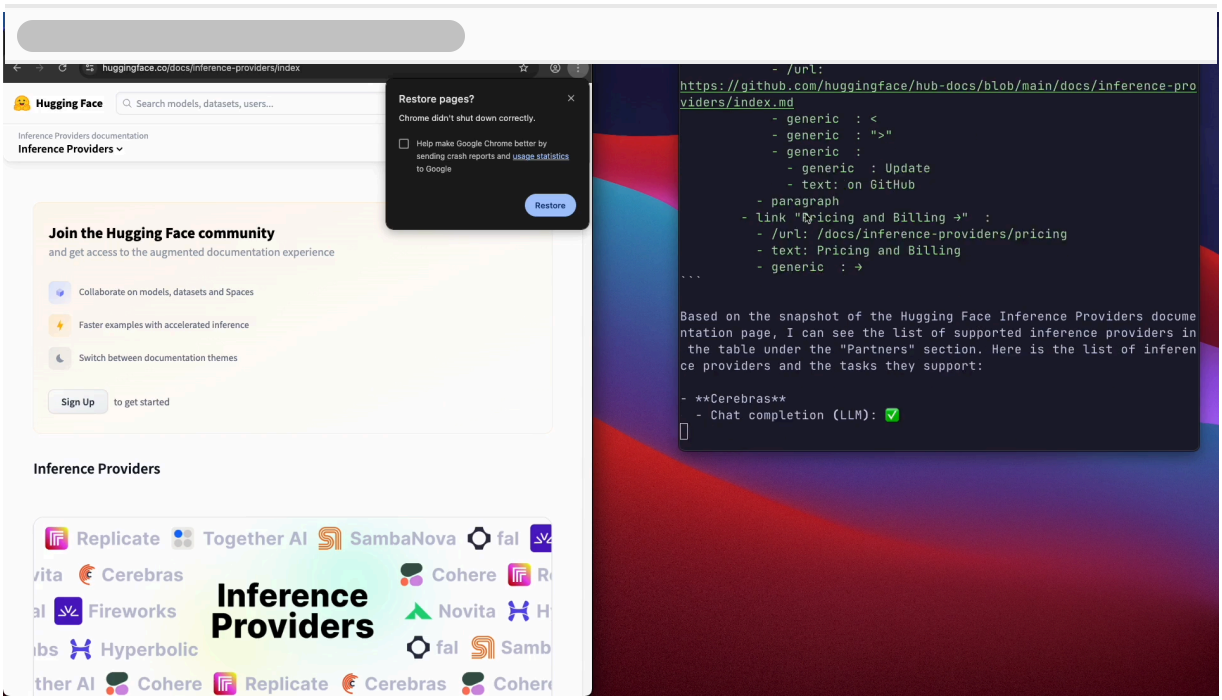
The coolest part is that you can load agents directly from the Hugging Face Hub tiny-agents Dataset, or specify a path to your own local agent configuration!

```
> tiny-agents run --help

 Usage: tiny-agents run [OPTIONS] [PATH] COMMAND [ARGS]...

 Run the Agent in the CLI


╭─ Arguments ──────────────────────────────────────────────────────────────────
│   path      [PATH]  Path to a local folder containing an agent.json
│                     (https://huggingface.co/datasets/tiny-agents/tin
╰──────────────────────────────────────────────────────────────────────────────
╭─ Options ────────────────────────────────────────────────────────────────────
│ --help          Show this message and exit.
╰──────────────────────────────────────────────────────────────────────────────
```

If you don't provide a path to a specific agent configuration, our Tiny Agent will connect by default to the following two MCP servers:

- the "canonical" file system server, which gets access to your Desktop,
- and the Playwright MCP server, which knows how to use a sandboxed Chromium browser for you.

The following example shows a web-browsing agent configured to use the Qwen/Qwen2.5-72B-Instruct model via Nebius inference provider, and it comes equipped with a playwright MCP server, which lets it use a web browser! The agent config is loaded specifying its path in the `tiny-agents/tiny-agents` Hugging Face dataset.

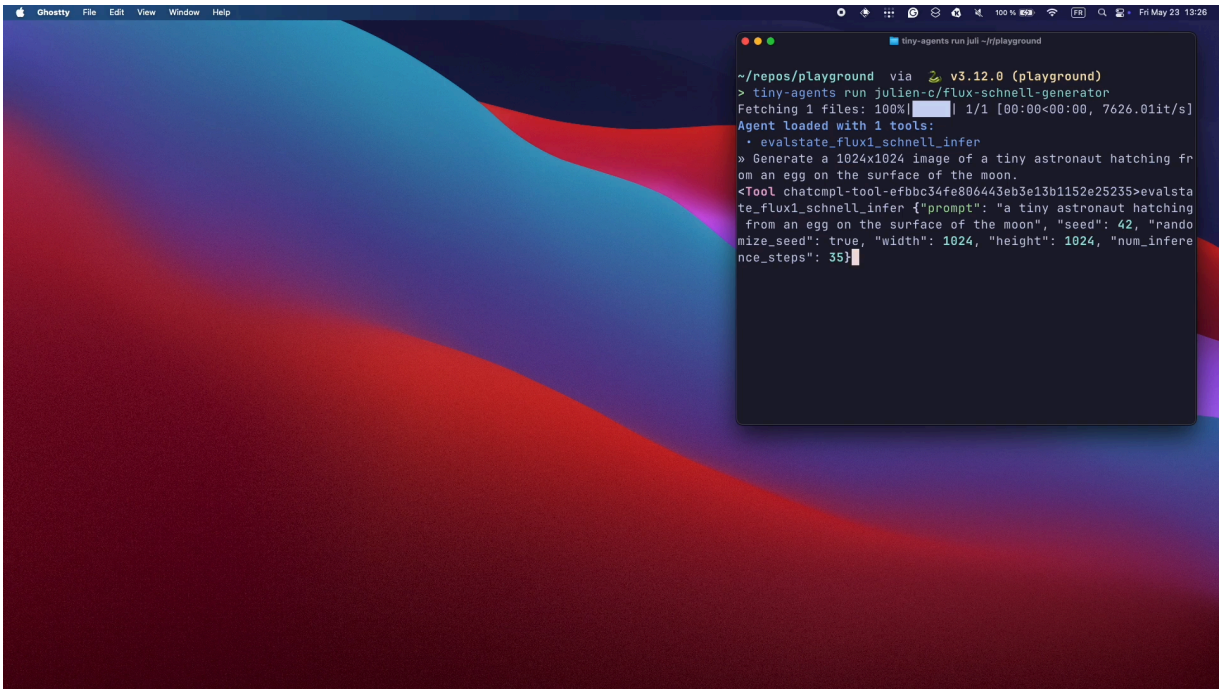


When you run the agent, you'll see it load, listing the tools it has discovered from its connected MCP servers. Then, it's ready for your prompts!

Prompt used in this demo:

> *“do a Web Search for HF inference providers on Brave Search and open the first result and then give me the list of the inference providers supported on Hugging Face ”*

You can also use Gradio Spaces as MCP servers! The following example uses Qwen/Qwen2.5-72B-Instruct model via Nebius inference provider, and connects to a `FLUX.1 [schnell]` image generation HF Space as an MCP server. The agent is loaded from its configuration in the tiny-agents/tiny-agents dataset on the Hugging Face Hub.



Prompt used in this demo:

> *“Generate a 1024x1024 image of a tiny astronaut hatching from an egg on the surface of the moon.”*

Now that you've seen how to run existing Tiny Agents, the following sections will dive deeper into how they work and how to build your own.

Agent Configuration

Each agent's behavior (its default model, inference provider, which MCP servers to connect to, and its initial system prompt) is defined by an `agent.json` file. You can also provide a custom `PROMPT.md` in the same directory for a more detailed system prompt. Here is an example:

`agent.json` The `model` and `provider` fields specify the LLM and inference provider used by the agent. The `servers` array defines the MCP servers the agent will connect to. In this example, a "stdio" MCP server is configured. This type of server runs as a local process. The Agent starts it using the specified `command` and `args`, and then communicates with it via stdin/stdout to discover and execute available tools.

```
{
    "model": "Qwen/Qwen2.5-72B-Instruct",
    "provider": "nebius",
    "servers": [
        {
            "type": "stdio",
            "config": {
                "command": "npx",
                "args": ["@playwright/mcp@latest"]
            }
        }
    ]
}
```

`PROMPT.md`

```
You are an agent - please keep going until the user's query is complet
```

> You can find more details about Hugging Face Inference Providers here.

LLMs Can Use Tools

Modern LLMs are built for function calling (or tool use), which enables users to easily build applications tailored to specific use cases and real-world tasks.

A function is defined by its schema, which informs the LLM what it does and what input arguments it expects. The LLM decides when to use a tool, the Agent then orchestrates running the tool and feeding the result back.

```
tools = [
        {
            "type": "function",
            "function": {
                "name": "get_weather",
                "description": "Get current temperature for a given lo
                "parameters": {
                    "type": "object",
                    "properties": {
                        "location": {
                            "type": "string",
                            "description": "City and country e.g. Pari
                        }
                    },
                    "required": ["location"],
                },
            }
        }
]
```

`InferenceClient` implements the same tool calling interface as the OpenAI Chat Completions API, which is the established standard for inference providers and the community.

Building our Python MCP Client

The `MCPClient` is the heart of our tool-use functionality. It's now part of `huggingface_hub` and uses the `AsyncInferenceClient` to communicate with LLMs.

> The full `MCPClient` code is in here if you want to follow along using the actual code 🤓

Key responsibilities of the `MCPClient`:

- Manage async connections to one or more MCP servers.
- Discover tools from these servers.
- Format these tools for the LLM.
- Execute tool calls via the correct MCP server.

Here's a glimpse of how it connects to an MCP server (the `add_mcp_server` method):

```
# Lines 111-219 of `MCPClient.add_mcp_server`
# https://github.com/huggingface/huggingface_hub/blob/main/src/hugging
class MCPClient:
    ...
    async def add_mcp_server(self, type: ServerType, **params: Any):
        # 'type' can be "stdio", "sse", or "http"
        # 'params' are specific to the server type, e.g.:
        # for "stdio": {"command": "my_tool_server_cmd", "args": ["--p
        # for "http": {"url": "http://my.tool.server/mcp"}

        # 1. Establish connection based on type (stdio, sse, http)
        #    (Uses mcp.client.stdio_client, sse_client, or streamableh
        read, write = await self.exit_stack.enter_async_context(...)

        # 2. Create an MCP ClientSession
        session = await self.exit_stack.enter_async_context(
```

```
            ClientSession(read_stream=read, write_stream=write, ...)
        )
        await session.initialize()

        # 3. List tools from the server
        response = await session.list_tools()
        for tool in response.tools:
            # Store session for this tool
            self.sessions[tool.name] = session
            #  Add tool to the list of available tools and Format for
            self.available_tools.append({
                "type": "function",
                "function": {
                    "name": tool.name,
                    "description": tool.description,
                    "parameters": tool.input_schema,
                },
            })
```

It supports `stdio` servers for local tools (like accessing your file system), and `http` servers for remote tools! It's also compatible with `sse`, which is the previous standard for remote tools.

Using the Tools: Streaming and Processing

The `MCPClient`'s `process_single_turn_with_tools` method is where the LLM interaction happens. It sends the conversation history and available tools to the LLM via `AsyncInferenceClient.chat.completions.create(..., stream=True)`.

1. Prepare tools and calling the LLM

First, the method determines all tools the LLM should be aware of for the current turn – this includes tools from MCP servers and any special "exit loop" tools for agent control; then, it makes a streaming call to the LLM:

```
# Lines 241-251 of `MCPClient.process_single_turn_with_tools`
# https://github.com/huggingface/huggingface_hub/blob/main/src/hugging

    # Prepare tools list based on options
    tools = self.available_tools
    if exit_loop_tools is not None:
        tools = [*exit_loop_tools, *self.available_tools]

    # Create the streaming request to the LLM
    response = await self.client.chat.completions.create(
        messages=messages,
        tools=tools,
        tool_choice="auto",  # LLM decides if it needs a tool
        stream=True,
    )
```

As chunks arrive from the LLM, the method iterates through them. Each chunk is immediately yielded, then we reconstruct the complete text response and any tool calls.

```
# Lines 258-290 of `MCPClient.process_single_turn_with_tools`
# https://github.com/huggingface/huggingface_hub/blob/main/src/hugging
# Read from stream
async for chunk in response:
      # Yield each chunk to caller
      yield chunk
      # Aggregate LLM's text response and parts of tool calls
      …
```

2. Executing tools

Once the stream is complete, if the LLM requested any tool calls (now fully reconstructed in `final_tool_calls`), the method processes each one:

```
# Lines 293-313 of `MCPClient.process_single_turn_with_tools`
# https://github.com/huggingface/huggingface_hub/blob/main/src/hugging
for tool_call in final_tool_calls.values():
    function_name = tool_call.function.name
    function_args = json.loads(tool_call.function.arguments or "{}")

    # Prepare a message to store the tool's result
    tool_message = {"role": "tool", "tool_call_id": tool_call.id, "con

    # a. Is this a special "exit loop" tool?
    if exit_loop_tools and function_name in [t.function.name for t in
        # If so, yield a message and terminate this turn's processing
        messages.append(ChatCompletionInputMessage.parse_obj_as_instan
        yield ChatCompletionInputMessage.parse_obj_as_instance(tool_me
        return # The Agent's main loop will handle this signal

    # b. It's a regular tool: find the MCP session and execute it
    session = self.sessions.get(function_name) # self.sessions maps to
    if session is not None:
        result = await session.call_tool(function_name, function_args)
        tool_message["content"] = format_result(result) # format_resul
    else:
        tool_message["content"] = f"Error: No session found for tool:
        tool_message["content"] = error_msg

    # Add tool result to history and yield it
    ...
```

It first checks if the tool called exits the loop (`exit_loop_tool`). If not, it finds the correct MCP session responsible for that tool and calls `session.call_tool()`. The

result (or error response) is then formatted, added to the conversation history, and yielded so the Agent is aware of the tool's output.

Our Tiny Python Agent: It's (Almost) Just a Loop!

With the `MCPClient` doing all the job for tool interactions, our `Agent` class becomes wonderfully simple. It inherits from `MCPClient` and adds the conversation management logic.

> The Agent class is tiny and focuses on the conversational loop, the code can be found here.

1. Initializing the Agent

When an Agent is created, it takes an agent config (model, provider, which MCP servers to use, system prompt) and initializes the conversation history with the system prompt. The `load_tools()` method then iterates through the server configurations (defined in agent.json) and calls `add_mcp_server` (from the parent `MCPClient`) for each one, populating the agent's toolbox.

```
# Lines 12-54 of `Agent`
# https://github.com/huggingface/huggingface_hub/blob/main/src/hugging
class Agent(MCPClient):
    def __init__(
        self,
        *,
        model: str,
        servers: Iterable[Dict], # Configuration for MCP servers
        provider: Optional[PROVIDER_OR_POLICY_T] = None,
        api_key: Optional[str] = None,
        prompt: Optional[str] = None, # The system prompt
    ):
```

```
        # Initialize the underlying MCPClient with model, provider, et
        super().__init__(model=model, provider=provider, api_key=api_k
        # Store server configurations to be loaded
        self._servers_cfg = list(servers)
        # Start the conversation with a system message
        self.messages: List[Union[Dict, ChatCompletionInputMessage]] =
            {"role": "system", "content": prompt or DEFAULT_SYSTEM_PRO
        ]

    async def load_tools(self) -> None:
        # Connect to all configured MCP servers and register their too
        for cfg in self._servers_cfg:
            await self.add_mcp_server(cfg["type"], **cfg["config"])
```

2. The agent's core: the Loop

The `Agent.run()` method is an asynchronous generator that processes a single user input. It manages the conversation turns, deciding when the agent's current task is complete.

```
# Lines 56-99 of `Agent.run()`
# https://github.com/huggingface/huggingface_hub/blob/main/src/hugging
async def run(self, user_input: str, *, abort_event: Optional[asyncio.
    ...
    while True: # Main loop for processing the user_input
        ...

        # Delegate to MCPClient to interact with LLM and tools for one
        # This streams back LLM text, tool call info, and tool results
        async for item in self.process_single_turn_with_tools(
            self.messages,
            ...
        ):
            yield item
```

```
        ...

        # Exit Conditions
        # 1. Was an "exit" tool  called?
        if last.get("role") == "tool" and last.get("name") in {t.funct
                return

        # 2. Max turns reached or LLM gave a final text answer?
        if last.get("role") != "tool" and num_turns > MAX_NUM_TURNS:
                return
        if last.get("role") != "tool" and next_turn_should_call_tools:
            return

        next_turn_should_call_tools = (last_message.get("role") != "to
```

Inside the `run()` loop:

- It first adds the user prompt to the conversation.
- Then it calls `MCPClient.process_single_turn_with_tools(...)` to get the LLM's response and handle any tool executions for one step of reasoning.
- Each item is immediately yielded, enabling real-time streaming to the caller.
- After each step, it checks exit conditions: if a special "exit loop" tools was used, if a maximum turn limit is hit, or if the LLM provides a text response that seems final for the current request.

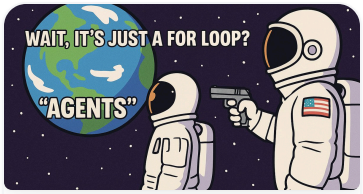
Next Steps

There are a lot of cool ways to explore and expand upon the MCP Client and the Tiny Agent 🔥 Here are some ideas to get you started:

- Benchmark how different LLM models and inference providers impact agentic performance: Tool calling performance can differ because each provider may optimize it differently. You can find the list of supported providers here.
- Run tiny agents with local LLM inference servers, such as llama.cpp, or LM Studio.
- .. and of course contribute! Share your unique tiny agents and open PRs in tiny-agents/tiny-agents dataset on the Hugging Face Hub.

Pull requests and contributions are welcome! Again, everything here is open source! 💎❤️

More Articles from our Blog



Tiny Agents: a MCP-powered agent in 50 lines of code

By julien-c • △ 284

Groq on Hugging Face Inference Providers 🔥

By sbrandeis • △ 36

🤗 Community

hevangel May 25

It doesn't work. I got this error

```
63 | | os._exit(130) |
| 64 | |
| 65 | try: |
| ❱ 66 | | loop.add_signal_handler(signal.SIGINT, _sigint_handler) |
| 67 | | |
| 68 | | async with Agent( |
| 69 | | | provider=config["provider"], |
| |
|
╭──────────────────────────────────────────────────
──────────────────────────────────────────────────
────────── locals
──────────────────────────────────────────────────
──────────────────────────────────────────────────
──────────╮ |
| | abort_event = <asyncio.locks.Event object at 0x000002B7D6EE1160 [unset]> | |
| | agent_path = '.\agent_playwright\' | |
| | config = { | |
| | | 'model': 'Qwen/Qwen2.5-72B-Instruct', | |
| | | 'provider': 'nebius', | |
| | | 'servers': [{'type': 'stdio', 'config': {'command': 'npx', 'args': ['@playwright/mcp']}}] | |
| | } | |
| | first_sigint = True | |
| | loop = | |
| | original_sigint_handler = functools.partial(<bound method Runner._on_sigint of
<asyncio.runners.Runner object at 0x000002B7D6EE0AD0>>, main_task=<Task finished | |
| | name='Task-1' coro=<run_agent() done, defined at B:\learning_llm\mcp.venv\Lib\site-
packages\huggingface_hub\inference_mcp\cli.py:32> | |
| | exception=NotImplementedError()>) | |
| | prompt = None | |
| | servers = [{'type': 'stdio', 'config': {'command': 'npx', 'args': ['@playwright/mcp']}}] | |
|
╰──────────────────────────────────────────────────
──────────────────────────────────────────────────
──────────────────────────────────────────────────
──────────────────────────────────────────────────
──────────────────────────────╯ |
```

1 reply ·

celinah Article author May 26

Hi @hevangel,
Very sorry for the inconvenience, we've fixed the issue and released a patch for that:
https://github.com/huggingface/huggingface_hub/releases/tag/v0.32.1
could you upgrade your `huggingface_hub` version please?

```
pip install -U huggingface_hub>=0.32.1
```

let us know if you see any other unexpected behavior!

❤️ 1 +

Reply in thread

insightfactory May 31

It is a very cool Idea,
I have converted your JavaScript version to Python,
It is a simple Loop + LiteLLM + MCP + Hook System for extendability.

https://github.com/askbudi/tinyagent

At the moment it supports the following Hooks:

- Gradio UI
- Rich UI
- Logging

And also storage layer:

- PG / Supabase
- SQLite
- JSON

The core Tiny Agent only depends on Litellm :D, nothing else.

I thought it would be cool if anyone could personalize their own Tinyagent based on what they need, for example adding memory layer, or storage in PG.
And here it is possible to chat with this repo and add functionality you need for your specific project

https://askdev.ai/github/askbudi/tinyagent

1 reply · 👍 2 +

celinah Article author Jun 2

Hi @insightfactory,
very nice! don't hesitate to check and contribute (we love contributions from the community!) to our tiny-agents Python implementation as well: https://github.com/huggingface/huggingface_hub/tree/main/src/huggingface_hub/inference/_mcp :)

Reply in thread

sylvain471 Jun 1 • edited Jun 2

Hello, there seems to be a problem with `huggingface_hub[mcp]` 0.32.3 which invariably produces

```
TypeError: Passing coroutines is forbidden, use tasks explicitly.
<sys>:0: RuntimeWarning: coroutine 'Event.wait' was never awaited
```

downgrading to 0.32.1 seems to fix the problem.

Concerning using local llm, an example of an `agent.json` config using ollama would be very much appreciated. I have been trying with configs like
EDIT: this config is indeed the proper way to set ollama

```
{
  "model": "llama3.2:3b",
  "endpointUrl": "http://localhost:11434",
  "servers": [... ]
```

```
}
```

with tiny-agent.js but it produces an error

```
SyntaxError: Unexpected non-whitespace character after JSON at positic
    at JSON.parse (<anonymous>)
```

EDIT: issue partially resolved see
https://github.com/huggingface/huggingface.js/issues/1502

and with tiny-agent python it produces an error apparently due to the missing `provider` key.

4 replies ·

celinah Article author Jun 2

Hi @sylvain471,
Thanks for reporting this!
Do you happen to have a reproducible example for the first error using `huggingface_hub==0.32.3` ? If possible, could you also share the agent config you used when the error occurred?

As for local LLMs, support was introduced in `huggingface_hub>=0.32.2` , so it's expected that older versions would raise an error.

Expand 3 replies

Reply in thread

sylvain471 Jun 3

Hi @celinah,
I made a minimal example with a reproducible example at https://github.com/sdelahaies/tiny-agent-mcp.git, but after some tests it seems that the problem comes from the python version: on my setup `uv venv` installs python 3.13.1

with which the example failed, with `uv venv --python 3.11` it works.
As for the version 0.32.1, I had to tweak the agent class for local LLM to work, but I lost track of these change running all my tests... anyway my conclusion is to downgrade the python version
thanks!

1 reply ·

celinah Article author Jun 3

yes indeed, the bug appears only on python >= 3.12. We fixed it and released a patch, see my comment here: https://huggingface.co/blog/python-tiny-agents#683eca4750b8f035fa0407b9

👍 1 +

Reply in thread

insightfactory 24 days ago

I extend my TinyAgent Implementation, and create CodeTinyAgent, it is similar to Smolagents, (Thinking in Python) but in 50ish lines of Code,
I also extend Modal Functions a bit to become stateful. CodeTinyAgent executes Python code in the cloud.

Live Version:
https://huggingface.co/spaces/Agents-MCP-Hackathon/TinyCodeAgent

Source:
https://github.com/askbudi/TinyCodeAgent

1 reply · ❤️ 2 +

julien-c Article author 23 days ago

that's very cool @insightfactory!

❤️ 1 +

Reply in thread

betki 21 days ago

This is my agent.json
{
"model": "qwen3:4b",
"endpointUrl": "http://localhost:11434/",
"provider": "auto",
"servers": [
{
"type": "sse",
"config": {
"url": "http://127.0.0.1:7860/gradio_api/mcp/sse"
}
}
]
}

I am getting this error:

Error during agent run: Repo id must use alphanumeric chars or '-', '_', '.', '--' and '..' are forbidden, '-' and '.' cannot start or end the name, max length is 96: 'qwen3:4b'.

I am using libraries with python 3.13, huggingface-hub 0.33.0

2 replies ·

julien-c Article author 19 days ago

can you remove the `provider: "auto"` line?

Expand 1 reply

Reply in thread

Edit Preview

Start discussing this article

Tap or paste here to upload images

Comment

Sign up or log in to comment

System theme

Company

TOS

Privacy

About

Jobs

Website

Models

Datasets

Spaces

Pricing

Docs