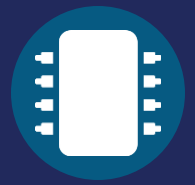


Running Local LLMs with MCP

Giving Large Language Models Real Tool Access

Adel Kabiri
Aug 2026

Agenda

Local LLM

- What Is a Local LLM?
- Why Run Models Locally?
- Setup: Ollama vs LM Studio
- Local Models Examples
- Enterprise Use
- Naming & GGUF Explained

MCP

- What is MCP?
- API vs. MCP
- MCP's Capabilities
- Pair MCP With a Local LLM
- MCP Flow
- Enterprise Local LLM + MCP
- Why Python?
- LM Studio as an MCP Host

In Practice

LIVE DEMO IN LM STUDIO

What Is a Local LLM?

A large language model that runs entirely on your own local machine.

Runs on your hardware

CPU/GPU on your laptop or desktop

Open-weight models

e.g. Gemma, Qwen, Llama, …, download and run offline

Managed through an app

LM Studio or Ollama provide the model browser, chat UI, and local API server

EXAMPLE MODEL USED

Gemma 4 E2B

4.3 GB

on-disk size

Why Run Models Locally?

Local LLMs trade cloud-scale compute and convenience for greater privacy, lower cost, and more control.

Privacy

Files, prompts, and outputs never leave your machine.

Cost

No per-token billing.

Offline access

Works with no internet connection.

Full control

Choose the exact model and context for your hardware.

Local LLM Setup: Two Ways

Both run the same open-weight models, the difference is the interface.

Ollama

Command-line

Pull and run a model:

```
$ ollama pull gemma3:4b
$ ollama run gemma3:4b
```

Call it from Python:

```
import ollama

resp = ollama.chat(
  model='gemma3:4b',
  messages=[{'role': 'user',
             'content': 'Hi'}]
)
print(resp['message']['content'])
```

LM Studio

Graphical app

1. Download & install the desktop app
2. Browse the built-in models and download a GGUF
3. Chat, or start a local server
4. Connect MCP servers directly

Local Models Examples

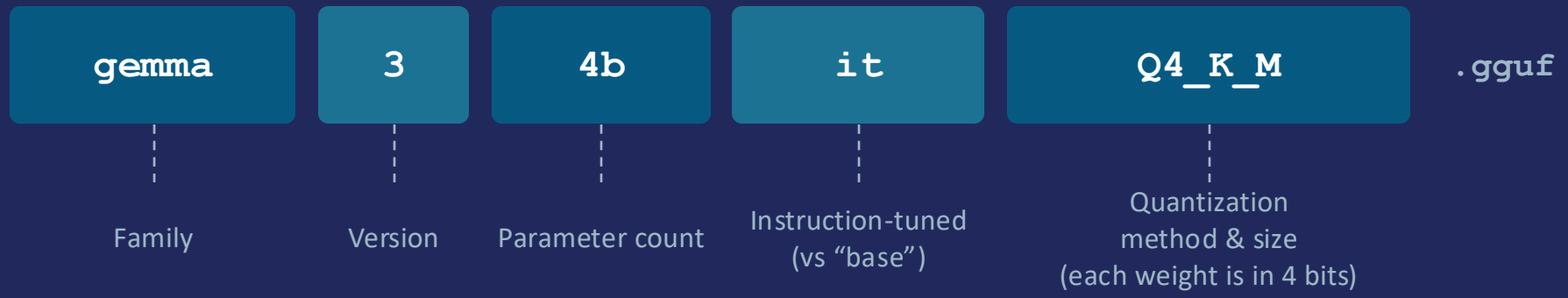
Examples of open-weight models, all available via Ollama or LM Studio.

Model	Company	Year	Number of Parameters	Quantization	Size	CPU speed* (tokens/second)
Qwen2.5-Coder 0.5B Instruct	Alibaba (Qwen)	2024	0.5B	Q4_K_M	~0.4 GB	~40–60
Llama 3.2 1B Instruct	Meta	2024	1B	Q4_K_M	~0.8 GB	~35–50
Phi-4-mini-reasoning	Microsoft	2025	3.8B	Q4_K_M	~2.3 GB	~15–20
Gemma 4 E2B IT	Google DeepMind	2026	2B	Q4 (QAT)	~4.3 GB	~6–8
Qwen3 7B Instruct	Alibaba (Qwen)	2025	7B	Q4_K_M	~4.4 GB	~18–25
Gemma 4 22B Instruct QAT	Google DeepMind	2026	22B	Q4 (QAT)	~13 GB	~4–6

**CPU-only estimates on modest consumer hardware*

Decoding Model Names & the GGUF File

Every filename in a model catalog acts as a spec sheet.



What is GGUF?

- A single-file format for storing a model's weights, tokenizer, and metadata together
- Built for llama.cpp
- Ships quantized (e.g., Q4_K_M, Q5_K_M, Q8_0), smaller number = smaller file, faster
- If you see .ggml, it's outdated, look for a .gguf re-upload

Local LLMs in the Enterprise

Models run on company-owned or private-cloud hardware. Employees and internal apps call it like any other internal API, but nothing leaves the network.

Benefits

- ✓ Data stays in-house
- ✓ Predictable cost
- ✓ Fine-tune based on internal data

Challenges

- ! Upfront infrastructure investment
- ! Ongoing model updates
- ! Capability gap on complex reasoning tasks

What Is MCP?

Model Context Protocol (MCP) is an open standard for connecting AI models to external tools, data sources, and workflows. It was introduced by Anthropic in November 2024.

One protocol, many tools

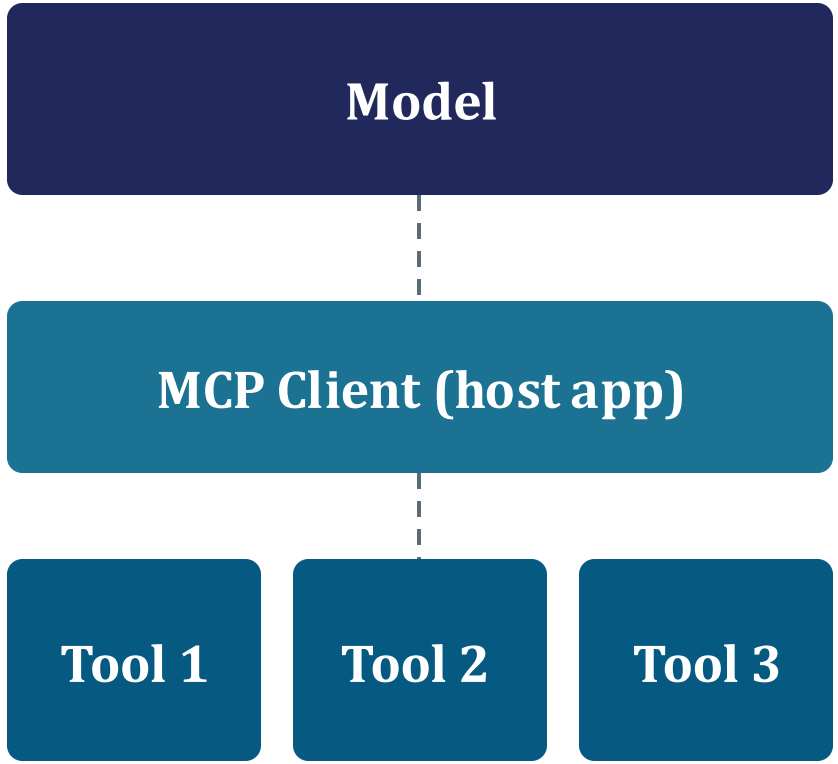
Any MCP-compatible app can plug into any MCP server

Tools, not just text

The model can call functions

You control what's exposed

Each server defines exactly which actions and folders are reachable



MCP Servers (tools)

API vs. MCP

Both move data to a model but the difference is *who's in the driver's seat*.

- **With API:** the developer writes code that calls the API and manually integrates the response into the model's prompt. The model has zero awareness that the API even exists.
- **With MCP:** the tool describes itself and its functionality. The model then decides whether it needs to call that tool, and the MCP client handles actually executing the call and feeding the result back in.

Traditional API

- Custom integration code for each service
- The model doesn't know what's available, your app should be directed when to call what
- Every new tool, a new client to write and maintain

MCP

- Tools describe their own capabilities, the model discovers them at runtime
- One client (the MCP host) works with any compliant server
- New tool = drop in a server; nothing else changes

MCP's Capabilities

MCP’s three core capabilities are: Tools, Resources, and Prompts.

Tools

Functions the model can call to implement

```
read_latest_email()
receipt_reader()
```

Resources

Data or files can be made available to read.

```
A file listing, a live
doc, a DB schema
```

Prompts

Reusable prompt templates

```
“summarize-doc”
“receipt-summary”
```

Our two demos (**email reader, receipt reader**) are tools. Resources and prompts are what make MCP a full context protocol, not just a function-calling protocol.

Why Pair MCP With a Local LLM?

MCP gives a local LLM hands, turning a model that can only talk into one that can ***do*** something. Neither piece is that exciting alone; together, they close the loop.

Turns a chatbot into an agent

A local model that can read files, run scripts, and act as a tool.

Keeps the entire loop private

Reasoning and tool execution both stay in-house (on-device), no data ever transfers to external sources.

One standard

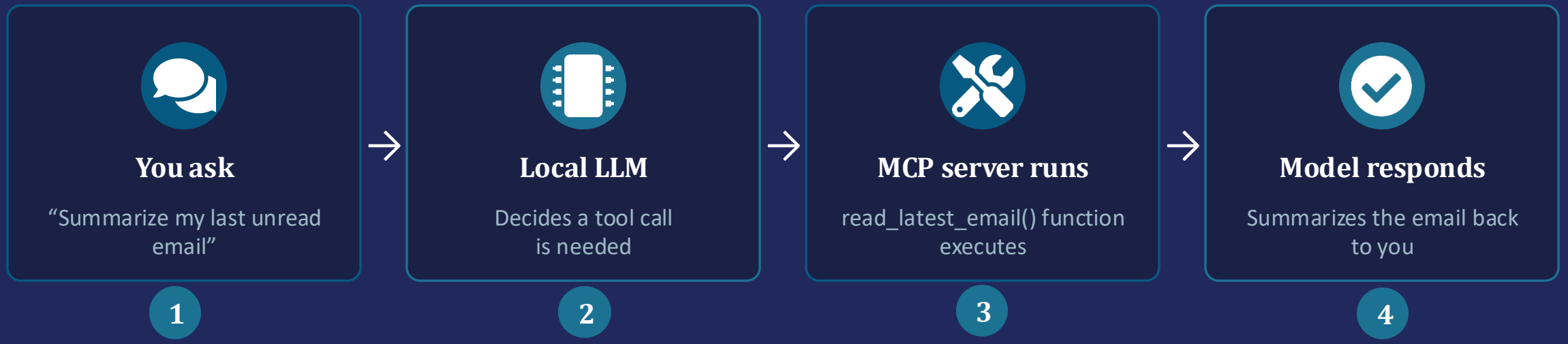
Write an MCP server once; it works with any MCP-compatible host, local or cloud.

Works fully offline

Local tool servers + a local model means the whole pipeline won't need internet connection.

MCP Flow

End-to-end flow, from the question to the performance of the task.



- Steps 3 and 4 never send your email contents to a remote server, no external data transfer
- read_latest_email() function is registered as an MCP tool using @mcp.tool() decorator in Python, therefore the model can discover and call it as a tool

Enterprise Local LLM + MCP

Local (in-house) LLMs pairing with MCP servers can be used in enterprise to perform tasks with full control over privacy and costs.

Sensitive data never leaves the network

Internal sensitive data can be processed without an external API in the loop.

Custom internal tools

Custom internal tools can be developed in-house.

Predictable cost

No per-token bill

Simpler compliance audits

The entire pipeline runs on infrastructure you control.

Why Python for Local LLM + MCP?

Python is the most common/most supported language for local LLM + MCP work. MCP itself is language-agnostic by design (Python just happens to have the most mature tooling).

- ✓ MCP's official SDK (FastMCP) was released in Python with the strong support of decorators
- ✓ Hugging Face's entire ecosystem is Python-native
- ✓ Fast iteration, suits the experimental nature of prompting, tool-calling, and ...

Python's large ecosystem and libraries make MCP implementation easy. The tools we built used the official MCP Python SDK. All you need is: “pip install mcp”.

LM Studio as an MCP Host

LM Studio can connect to MCP servers directly.

1. **Install dependencies**

   pip install mcp

2. **Register the server**

   Add it to mcp.json

3. **Load a tool-capable model**

   Look for the (hammer icon), confirms the model supports tool calling

4. **Enable & chat**

   Integrate and ask for the task

mcp.json

```
{
  "mcpServers": {
    "pdf-folder-reader": {
      "command": "python",
      "args": [
        "./pdf_folder_mcp.py"
      ]
    }
  }
}
```

▶ **UP NEXT: LIVE DEMO IN LM STUDIO**

Thank You