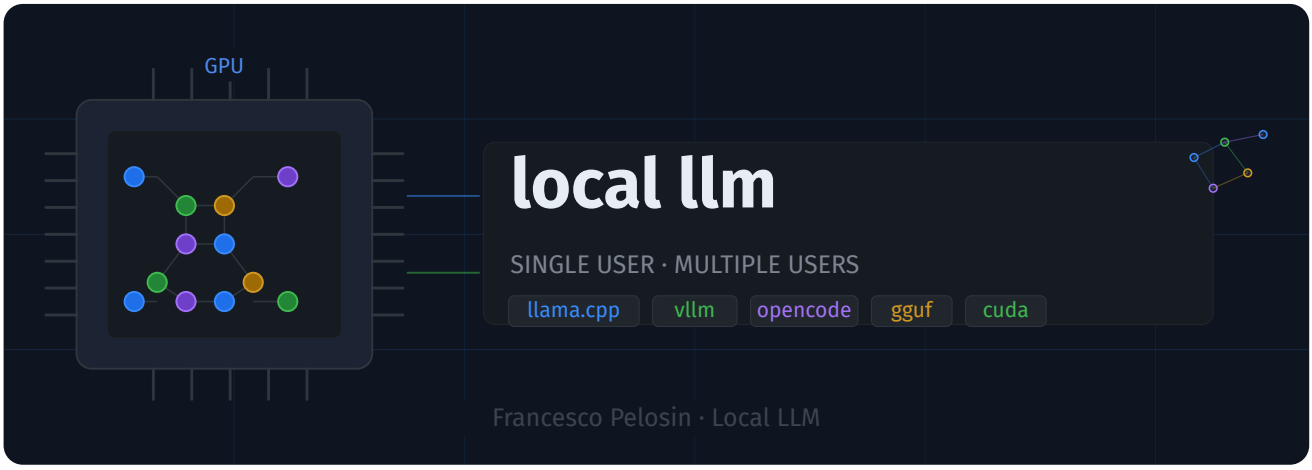
Local LLM – Single User vs Multiple Users

Francesco Pelosin June 28, 2026



1. Setup `llama.cpp` for Single User

This section covers how to set up and serve a local LLM on GPU using `llama.cpp`. This approach is best suited for a **single user** who wants low-latency access to a model, full control over serving parameters, and optional integration with agentic tools like Opencode.

> When to use llama.cpp
>
> Use `llama.cpp` when you are the only person accessing the model. It is simpler to set up and delivers lower latency than `vllm`, but handles requests sequentially — not suitable for concurrent users.

1.1. Prerequisites

CUDA Toolkit

To run the model on GPU, the CUDA toolkit must be installed. Check whether it is already available:

```
nvcc --version
```

If the command produces no output, install it with:

```
sudo apt install -y nvidia-cuda-toolkit
```

SSL Development Libraries

Next, `libssl-dev` is required to download models directly from Hugging Face via the command line:

```
sudo apt install -y libssl-dev
```

This is handy because we will later see that we can just change the name of the model and the script will download and serve it easily.

1.2. Build

Clone the `llama.cpp` repository and enter the project directory:

```
git clone https://github.com/ggml-org/llama.cpp.git
cd llama.cpp
```

Compile with CUDA and OpenSSL support to enable GPU inference and model downloads:

```
cmake -B build -DGGML_CUDA=ON -DLLAMA_OPENSSL=ON -DCMAKE_BUILD_TYPE=Release
cmake --build build --config Release -j
```

This produces the server binary at `./build/bin/llama-server`. You will serve the model with the application built inside here.

1.3. Serve

In a separate terminal, launch the server. The example below targets `GPU0` (an A6000). The server will automatically download the specified model if not already present, and exposes two services:

- A **web UI** at `localhost:8034`, usable like a standard chat interface.
- An **OpenAI-compatible API endpoint** for local development and agentic tool integration.

```
CUDA_VISIBLE_DEVICES=0 ./build/bin/llama-server \
  -hf unsloth/Qwen3.6-35B-A3B-GGUF:UD-Q5_K_XL \
  --jinja \
```

```
  -c 65536 \
  --port 8034 \
  -ngl 999 \
  --flash-attn on

# Alternatively, use a smaller model if VRAM is limited:
# ./build/bin/llama-server \
#   -hf unsloth/gemma-4-E4B-it-GGUF:UD-Q4_K_XL \
#   --jinja -c 32768 --port 8033
```

Key parameters:

Flag	Description
`-hf MODEL`	Downloads and loads the model from Hugging Face. Set `HF_TOKEN` as an environment variable for gated models, or use the `hf` CLI with `hf auth` for interactive login.
`-ngl N`	Number of model layers to offload to the GPU. Use `999` to offload all layers.
`-c N`	Context window size in tokens. Larger values consume more VRAM. Opencode requires at least `65536` to accommodate its hidden system prompts for agentic behaviour.
`--jinja`	Enables Jinja-based chat template rendering, required for tool calling.
`--flash-attn on`	Enables Flash Attention for faster and more memory-efficient inference.
`--port N`	Port on which the server listens.

HF Token: Some models on Hugging Face require accepting a license before download. Set `HF_TOKEN=your_token` in your environment, or run `hf auth` with the `hf` CLI tool before launching the server.

1.4. Models Tested

The following models have been installed and tested on `server15`:

Model	Notes
`unsloth/Qwen3.6-35B-A3B-GGUF:UD-Q5_K_XL`	Best overall. Reasoning quality is on par with Claude from a few months ago. **Recommended default.**
`mradermacher/Qwen3.5-27B-Claude-4.6-Opus-Reasoning-Distilled-i1-GGUF:Q6_K`	Very capable, but Qwen 3.6 consistently outperforms it on agentic tasks.
`unsloth/gemma-4-E4B-it-GGUF:UD-Q4_K_XL`	Not reliable in agentic/tool-use scenarios. Works acceptably for conversational use via the web UI.

2. Opencode

Opencode is a terminal-based AI coding agent. Once `llama.cpp` is serving a model, Opencode can be pointed at the local API endpoint to use it as its backend. Add the following configuration to `~/.config/opencode/opencode.json`:

```
{
  "$schema": "https://opencode.ai/config.json",
  "model": "qwen3.6-local/unsloth/Qwen3.6-35B-A3B-GGUF:UD-Q5_K_XL",
  "provider": {
    "gemma-local": {
      "name": "llama.cpp Gemma",
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://127.0.0.1:8033/v1"
      },
      "models": {
        "unsloth/gemma-4-E4B-it-GGUF:UD-Q4_K_XL": {
          "name": "Gemma 4 E4B local",
          "tool_call": true
        }
      }
    },
    "qwen3.6-local": {
      "name": "llama.cpp Qwen",
      "npm": "@ai-sdk/openai-compatible",
      "options": {
        "baseURL": "http://127.0.0.1:8034/v1"
      },
      "models": {
        "unsloth/Qwen3.6-35B-A3B-GGUF:UD-Q5_K_XL": {
          "name": "Qwen 3.6 local",
          "tool_call": true
        }
      }
    }
  }
}
```

Configuration notes

- Each entry under `"provider"` maps to a running `llama-server` instance.
- The `"baseURL"` must match the host and port used when the server was started.
- `"tool_call": true` tells Opencode that the model supports function/tool calling — required for agentic behaviour.

3. Setup `vllm` for Multiple Users

When multiple users need concurrent access to a model, `llama.cpp` is not the right tool — it handles requests sequentially and is optimised for single-user use. `vllm` is designed for **high-throughput multi-user inference**, using techniques like continuous batching and PagedAttention to serve many requests efficiently.

When to use vllm

Use `vllm` when several users need to query the model simultaneously. However, it is more complex to set up than `llama.cpp` but scales well across multiple GPUs and handles concurrent requests without queuing them sequentially.

⚠️⚠️⚠️ ***I tried so hard to run Qwen 3.6 but I failed, therefore I suggest to just use `llama.cpp`. In this part of the tutorial I use a small model which I know it works. You should always check CUDA and gpu compatibilities before running these models (spoiler, it is a madness).***

3.1. Install

First, check your CUDA driver version, as the correct `vllm` wheels depend on it.

CUDA 13 (cu130):

```
uv pip install vllm \
  --extra-index-url https://wheels.vllm.ai/0.19.1/cu130 \
  --extra-index-url https://download.pytorch.org/whl/cu130 \
  --index-strategy unsafe-best-match
```

CUDA 12.x and below:

```
uv pip install vllm
```

When in doubt, consult the official installation guide at https://docs.vllm.ai for the correct wheels for your environment and driver version.

3.2. Serve

Start the vllm server with the desired model and configuration:

```
uv run vllm serve Qwen/Qwen2.5-7B-Instruct \
  --host 127.0.0.1 \
  --port 8000 \
  --max-model-len 32768 \
  --tensor-parallel-size 4
```

Key parameters:

Flag	Description
`--host`	Binds the server to a specific interface. `127.0.0.1` restricts access to localhost; use `0.0.0.0` to allow external connections.
`--port`	Port on which the API is served. Default is `8000`.
`--max-model-len`	Maximum context length in tokens. Reduce this if the model does not fit in VRAM at its default context size.

`--tensor-parallel-size`	Number of GPUs to shard the model across. Must divide evenly into the model's number of attention heads.

The server exposes an OpenAI-compatible API at `http://127.0.0.1:8000/v1`.

3.3. Web UI

To provide a ChatGPT-style interface for end users, deploy Open WebUI via Docker:

```
docker run -d --network=host \
  -e OPENAI_API_BASE_URL=http://127.0.0.1:8000/v1 \
  -e OPENAI_API_KEY=sk-no-key \
  --name open-webui \
  ghcr.io/open-webui/open-webui:main
```

The `--network=host` flag allows the container to reach the `vllm` server on the host machine. The interface is then available at `localhost:8080`.

Since the server is remote, forward port `8080` over SSH to access the UI from your local machine:

```
ssh -L 8080:127.0.0.1:8080 user@server15
```

Then open `http://localhost:8080` in your browser. On first login you will be prompted to create an admin account.

3.4. Agentic Provider

To connect the `vllm`-served model to Opencode, add the following block to your `opencode.json` provider list:

```
"qwen-vllm": {
  "name": "vLLM Qwen",
  "npm": "@ai-sdk/openai-compatible",
  "options": {
    "baseURL": "http://127.0.0.1:8000/v1",
    "apiKey": "sk-no-key"
  },
  "models": {
    "Qwen/Qwen2.5-7B-Instruct": {
      "name": "Qwen 2.5 (vLLM)",
      "tool_call": true
    }
  }
}
```

> The `"apiKey"` field is required by the OpenAI-compatible client library, but `vllm` does not enforce authentication by default — any non-empty string works.

4. Summary: llama.cpp vs vllm

Aspect	llama.cpp	vllm
Best for	Single user, local dev	Multiple concurrent users
Throughput	Low — sequential	High — continuous batching
Setup complexity	Low	Moderate
GPU support	Single GPU	Multi-GPU (tensor parallel)
Model format	GGUF (quantised)	Full-precision HF checkpoint
Web UI	Built-in (basic)	Open WebUI via Docker
Agentic (Opencode)	✓ Yes	✓ Yes
API compatibility	OpenAI-compatible	OpenAI-compatible

💡 Recommendation

Start with `llama.cpp` + Qwen 3.6 for personal development. Move to `vllm` when you need to share access with teammates or run the model as a shared service on the server, keep in mind that it is very difficult to set up.