WSL2 + Ollama on Windows: Complete Setup Guide (GPU Passthrough Included)

March 1, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Install WSL2 (wsl --install), set up .wslconfig with enough memory and sparseVhd=true, install Ollama with the one-liner (curl -fsSL https://ollama.com/install.sh | sh), and you're running. GPU passthrough works automatically with your existing NVIDIA Windows driver — don't install any Linux GPU driver inside WSL2. Performance is within 5% of native Windows Ollama for GPU-accelerated inference. The main reasons to use WSL2 over Windows native: Docker Compose with Open WebUI, Linux tooling, and a development environment that matches production servers.

📚 **Related:** WSL2 for Local AI (Full Guide) · Ollama Troubleshooting · Ollama vs LM Studio · Open WebUI Setup · Planning Tool

Windows has a native Ollama installer. It works. So why bother with WSL2?

Because the moment you want Docker Compose, Open WebUI, Python scripts that call the Ollama API, or a dev environment that matches your deployment server, you're going to want Linux. WSL2 gives you that without dual-booting, and GPU inference runs at the same speed as native Windows.

Here's everything: `wsl --install` to Ollama with GPU acceleration, Open WebUI in your browser, Docker Compose managing the stack, and the gotchas that will eat your afternoon if nobody warns you.

Install WSL2

Open an **Administrator PowerShell**:

```
wsl --install
```

This installs WSL2, the Linux kernel, and Ubuntu. It will prompt for a UNIX username and password on first launch. Restart your machine if prompted.

For a specific version:

```
wsl --install -d Ubuntu-24.04
```

Verify WSL2 is active (not WSL1):

```
wsl --list --verbose
```

If your distro shows VERSION 1, convert it:

```
wsl --set-version Ubuntu-24.04 2
```

Enable systemd

Ollama runs as a systemd service. Without systemd, you'll get `"System has not been booted with systemd as init system"` when trying to manage the service.

Inside WSL, edit `/etc/wsl.conf`:

```
[boot]
systemd=true
```

Restart from PowerShell: `wsl --shutdown`, then reopen your WSL terminal.

Configure memory

WSL2 defaults to **50% of your system RAM**. On a 32GB system, that's 16GB. Fine for 7B models, tight for anything larger.

Create or edit `C:\Users\<YourUsername>\.wslconfig`:

```
[wsl2]
memory=24GB
swap=8GB
processors=8
localhostForwarding=true
networkingMode=mirrored

[experimental]
autoMemoryReclaim=dropcache
sparseVhd=true
```

Restart WSL: `wsl --shutdown`

Setting	What It Does
`memory`	Leave 4-8GB for Windows, give the rest to WSL2
`swap`	Prevents OOM kills during model loading
`sparseVhd`	Stops your virtual disk from ballooning when you download and delete models
`autoMemoryReclaim`	`dropcache` releases RAM when WSL is idle. Don't use `gradual` — it conflicts with systemd and can freeze your shell
`networkingMode`	`mirrored` makes services accessible from your LAN. Requires Windows 11 22H2+. Windows 10 users: remove this line

GPU VRAM is not affected by .wslconfig. Your models get the full GPU memory minus ~200-500MB for the Windows desktop compositor.

GPU passthrough

This is simpler than most guides make it. Two rules:

1. **Install the NVIDIA driver on Windows only.** Your standard GeForce Game Ready or Studio driver (535+ minimum, 560+ recommended).
2. **Do NOT install any NVIDIA Linux GPU driver inside WSL2.** The Windows driver is automatically "stubbed" into WSL2 as `libcuda.so`.

Verify GPU access

Inside WSL2:

```
nvidia-smi
```

You should see your GPU model, driver version, and CUDA version. If this fails:

- Update your Windows NVIDIA driver
- Run `wsl --update` from PowerShell
- Run `wsl --shutdown` and relaunch

Install CUDA toolkit (optional)

Ollama doesn't need the CUDA Toolkit because it bundles its own CUDA runtime. But if you're also building llama.cpp or running PyTorch, install the toolkit with this rule: **install `cuda-toolkit-12-x` only, never the `cuda` or `cuda-drivers` meta-packages.** Those meta-packages install a Linux driver that overwrites the WSL2 GPU stub and breaks everything.

```
wget https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64/cuda-keyring_1.1
sudo dpkg -i cuda-keyring_1.1-1_all.deb
sudo apt-get update
sudo apt-get -y install cuda-toolkit-12-6
```

For details on CUDA setup, llama.cpp, and PyTorch in WSL2, see our full WSL2 local AI guide.

Install Ollama

```
curl -fsSL https://ollama.com/install.sh | sh
```

Same one-liner as native Linux. The install script auto-detects your GPU through the WSL2 CUDA stub.

Verify GPU detection

Pull a model and check:

```
ollama pull qwen2.5:14b-instruct-q4_K_M
ollama run qwen2.5:14b-instruct-q4_K_M "hello"
```

While the model is running, open a second WSL terminal:

```
ollama ps
```

The `PROCESSOR` column tells you where inference is happening. You want to see `100% GPU` or a GPU percentage. If it says `100% CPU`, something is wrong.

More ways to verify:

```
# Check Ollama's logs for GPU detection
journalctl -u ollama --no-pager | grep -i gpu

# Watch GPU utilization in real time
watch -n 1 nvidia-smi
```

You should see `ollama_llama_server` in the nvidia-smi process list with VRAM allocated. A 14B model at Q4 should use roughly 8-9GB of VRAM and generate 30-80+ tokens per second depending on your GPU.

If the GPU isn't detected

Check these in order:

Symptom	Fix
`nvidia-smi` fails entirely	Update Windows NVIDIA driver, run `wsl --update`, restart WSL
`nvidia-smi` works but Ollama says CPU	Restart Ollama: `sudo systemctl restart ollama`

Symptom	Fix
`"CUDA error 100"` in logs	You probably installed the `cuda` package. Purge it: `sudo apt remove --purge cuda cuda-drivers && sudo apt autoremove`
Very slow (5-15 tok/s on a modern GPU)	Model is running on CPU. Check `ollama ps` — Processor should show GPU

See our Ollama troubleshooting guide for more fixes.

Set up Open WebUI

Open WebUI gives you a ChatGPT-like browser interface for Ollama. The easiest way to run it in WSL2 is Docker.

Install Docker in WSL2

```
sudo apt update && sudo apt install -y docker.io
sudo systemctl enable --now docker
sudo usermod -aG docker $USER
```

Log out and back in (or run `newgrp docker`) for the group change to take effect.

Run it

The simplest approach is host networking, so Open WebUI can reach Ollama at localhost:

```
docker run -d \
  --network=host \
  -v open-webui:/app/backend/data \
  --name open-webui \
  --restart always \
  ghcr.io/open-webui/open-webui:main
```

Open `http://localhost:8080` in your Windows browser. Create an account (local only, not sent anywhere), and you'll see your Ollama models.

If host networking doesn't work (some Docker Desktop configurations):

```
docker run -d \
  -p 3000:8080 \
  --add-host=host.docker.internal:host-gateway \
  -e OLLAMA_BASE_URL=http://host.docker.internal:11434 \
  -v open-webui:/app/backend/data \
  --name open-webui \
  --restart always \
  ghcr.io/open-webui/open-webui:main
```

Then access at `http://localhost:3000`.

If Open WebUI can't connect to Ollama

Make sure Ollama is listening on all interfaces, not just loopback:

```
sudo systemctl edit ollama.service
```

Add:

```
[Service]
Environment="OLLAMA_HOST=0.0.0.0:11434"
```

Then:

```
sudo systemctl daemon-reload
sudo systemctl restart ollama
```

Verify it's listening:

```
curl http://localhost:11434
```

You should see `Ollama is running`.

Docker Compose setup

If you want Ollama and Open WebUI managed together, Docker Compose handles that.

Install Docker Compose plugin

```
sudo apt install -y docker-compose-v2
```

GPU support: install NVIDIA Container Toolkit

This lets Docker containers access your GPU:

```
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | \
  sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg

curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list
  sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] h
  sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

sudo apt-get update && sudo apt-get install -y nvidia-container-toolkit
sudo nvidia-ctk runtime configure --runtime=docker
sudo systemctl restart docker
```

Test GPU access in Docker:

```
docker run --rm --gpus all nvidia/cuda:12.6.0-runtime-ubuntu24.04 nvidia-smi
```

The docker-compose.yml

Create `~/ollama-stack/docker-compose.yml`:

```
services:
  ollama:
    image: ollama/ollama:latest
    container_name: ollama
    ports:
```

```
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama
    environment:
      - OLLAMA_HOST=0.0.0.0:11434
      - OLLAMA_FLASH_ATTENTION=1
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
    restart: unless-stopped

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    ports:
      - "8080:8080"
    volumes:
      - open_webui_data:/app/backend/data
    environment:
      - OLLAMA_BASE_URL=http://ollama:11434
    depends_on:
      - ollama
    restart: unless-stopped

volumes:
  ollama_data:
  open_webui_data:
```

Start it:

```
cd ~/ollama-stack
docker compose up -d
```

Pull a model:

```
docker exec ollama ollama pull qwen2.5:14b-instruct-q4_K_M
```

Open `http://localhost:8080` in your Windows browser.

Docker Compose vs bare-metal Ollama in WSL2

	Bare-metal Ollama + Docker Open WebUI	Full Docker Compose
Performance	Slightly faster (no container overhead on Ollama)	~1-2% slower for Ollama
Management	Two things to manage separately	`docker compose up -d` starts everything
Updates	`ollama update` + `docker pull` separately	`docker compose pull && docker compose up -d`
Model storage	`~/.ollama/models/`	Docker volume (harder to inspect)
GPU access	Automatic	Requires nvidia-container-toolkit

Bare-metal Ollama + Docker Open WebUI is the simplest path for most people. Docker Compose is better if you want one command to start everything or you're adding more services later.

The gotchas

1. Port conflict: Windows Ollama vs WSL2 Ollama

If you have Ollama installed on both Windows and WSL2, both try to bind port 11434. You'll get `"bind: address already in use"` or the Windows app will silently report `"another instance of ollama is running"`.

Fix: Pick one. If you're running Ollama in WSL2, uninstall the Windows version (or at least quit the tray icon). Alternatively, run the WSL2 instance on a different port:

```
export OLLAMA_HOST=0.0.0.0:11435
ollama serve
```

2. File system performance

Accessing files through `/mnt/c/` (your Windows drives) is 3-5x slower than WSL2's native filesystem. This affects model loading if your models are stored on the Windows side.

Rule: Keep everything inside WSL2's filesystem. Models go in `~/.ollama/models/` by default, which is already on the Linux filesystem. Don't move them to `/mnt/c/`.

Access WSL2 files from Windows Explorer via `\\wsl.localhost\Ubuntu-24.04\home\youruser\`.

3. VPN kills WSL2 networking

Corporate VPNs (Cisco AnyConnect, GlobalProtect, Pulse Secure) commonly break WSL2 internet access. The VPN changes the routing table and DNS in ways that disconnect the WSL2 VM.

Fix for Windows 11: Set `networkingMode=mirrored` in `.wslconfig`. This makes WSL2 share the host network stack, including VPN connections.

Fix for Windows 10 (or if mirrored doesn't work): Adjust the VPN interface metric so WSL2 traffic routes correctly:

```
# Run in PowerShell after connecting to VPN
Get-NetAdapter | Where-Object {$_.InterfaceDescription -Match "Cisco AnyConnect"} | Set-NetIPInt
```

You may also need to manually set DNS in WSL2:

```
sudo bash -c 'echo "nameserver 8.8.8.8" > /etc/resolv.conf'
```

This needs to be re-run each time you connect to the VPN.

4. Disk bloat

WSL2 stores its filesystem in a VHDX file that grows when you download models but doesn't shrink when you delete them. After cycling through several large models, you can lose tens of GB of disk space.

Prevention: Set `sparseVhd=true` in `.wslconfig` (stops future bloat).

Fix existing bloat:

```
# Inside WSL2, release deleted blocks
sudo fstrim /
```

```
# From PowerShell after wsl --shutdown
wsl --manage Ubuntu-24.04 --resize-vhd
```

5. OOM kills during model loading

If WSL2 runs out of its allocated memory, the Linux OOM killer terminates processes. Usually mid-model-load, with no useful error message. Ollama just crashes.

Fix: Increase `memory` and `swap` in `.wslconfig`. A 70B model at Q4 needs ~40GB of RAM for initial loading even though it runs in VRAM afterward.

6. systemd + autoMemoryReclaim=gradual = frozen shell

Using `autoMemoryReclaim=gradual` with systemd enabled can cause shell commands like `ls` and `apt update` to hang indefinitely. This is a known WSL2 bug.

Fix: Use `autoMemoryReclaim=dropcache` instead. It releases memory immediately rather than waiting for idle CPU, and doesn't conflict with systemd.

Performance: Windows native vs WSL2

Windows Central tested Ollama across multiple models (deepseek-r1:14b, gemma3:27b, and others) on an RTX 5080 and found the tokens-per-second numbers were "as near as makes no difference, identical" between native Windows and WSL2.

Platform	Performance	Notes
Native Linux	Baseline (fastest)	No virtualization overhead
WSL2	95-100% of native	WSL2 overhead is negligible for GPU-bound inference
Windows native	95-100% of native	WDDM driver overhead; varies by Ollama version

Windows native vs WSL2 Ollama: under 5% difference for GPU inference. The GPU is the bottleneck, not the OS layer.

Where you will see a gap: CPU-only inference (Linux is consistently faster), model loading from `/mnt/c/` (3-5x slower than loading from WSL2's native filesystem), and context window size, which matters far more than platform choice. One test showed 86 tok/s at 4K context vs 9 tok/s at 64K on the same model.

When to use Windows native Ollama

- You just want to chat with models and don't need Docker, Python, or Linux tools
- You're on Windows 10 and don't want to deal with WSL2 networking
- You want the simplest possible setup

When to use WSL2 Ollama

- You need Docker Compose (Ollama + Open WebUI + other services)
- You develop AI apps and need Python, CUDA, and Linux tooling
- You want a setup that matches Linux deployment servers
- You're already using WSL2 for development

Network access from other devices

If you want to access Ollama from your phone, tablet, or another computer on your network:

With mirrored networking (Windows 11 only)

If you set `networkingMode=mirrored` in `.wslconfig`, Ollama is already accessible on your LAN at your computer's IP address. You just need:

1. Ollama listening on all interfaces: `OLLAMA_HOST=0.0.0.0:11434`
2. A Windows Firewall rule:

```
New-NetFirewallRule -DisplayName "Ollama" -Direction Inbound -LocalPort 11434 -Protocol TCP -Act
```

Then access Ollama from any device at `http://<your-pc-ip>:11434`.

For Open WebUI, add a rule for port 8080 too.

Without mirrored networking (Windows 10 or NAT mode)

You need manual port forwarding. WSL2's IP changes on every reboot, so use a script:

```
# Save as forward-ollama.ps1, run as Administrator
$wslIp = (wsl hostname -I).Trim().Split(" ")[0]
netsh interface portproxy delete v4tov4 listenport=11434 listenaddress=0.0.0.0
netsh interface portproxy add v4tov4 listenport=11434 listenaddress=0.0.0.0 connectport=11434 co
netsh interface portproxy delete v4tov4 listenport=8080 listenaddress=0.0.0.0
netsh interface portproxy add v4tov4 listenport=8080 listenaddress=0.0.0.0 connectport=8080 conn
Write-Host "Forwarding to WSL2 at $wslIp"
```

Add a firewall rule for both ports, and run this script after each reboot.

Quick start checklist

1. `wsl --install` from Admin PowerShell, restart
2. Enable systemd in `/etc/wsl.conf`
3. Create `.wslconfig` with adequate memory, `sparseVhd=true`, and `autoMemoryReclaim=dropcache`
4. `wsl --shutdown` and relaunch
5. Verify GPU: `nvidia-smi` inside WSL2
6. Install Ollama: `curl -fsSL https://ollama.com/install.sh | sh`
7. Set `OLLAMA_HOST=0.0.0.0:11434` in the systemd service
8. Pull a model: `ollama pull qwen2.5:14b-instruct-q4_K_M`
9. Run it: `ollama run qwen2.5:14b-instruct-q4_K_M`
10. (Optional) Install Docker and Open WebUI for a browser interface

Total time from fresh Windows install to chatting with a local model: about 20 minutes, plus model download time.

Get notified when we publish new guides.

Subscribe — free, no spam

Source: https://insiderllm.com/guides/wsl2-ollama-windows-setup-guide/

Free guides for running AI locally