WSL2 for Local AI: The Complete Windows Setup Guide

February 23, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** WSL2 gives you a full Linux environment on Windows with near-native GPU performance for LLM inference. One command to install (wsl –install), GPU passthrough works automatically with your existing NVIDIA driver, and Ollama/llama.cpp run at 90-100% of native Linux speed. Key setup: allocate enough memory in .wslconfig (default is only 50% of RAM), install cuda-toolkit-12-x (NOT the cuda meta-package — it breaks the driver stub), and keep all AI files inside the WSL filesystem (not /mnt/c, which is 3-5x slower). For most Windows users who need Python, CUDA, Docker, and Ollama, WSL2 is the fastest path to a working local AI setup.

📚 **Related:** Ollama Troubleshooting Guide · llama.cpp vs Ollama vs vLLM · Local AI Troubleshooting · Run Your First Local LLM · Planning Tool

Most AI tools are Linux-first. The best guides assume Ubuntu. The Docker images target Linux. And if you're on Windows, you're either dual-booting or fighting compatibility issues.

WSL2 fixes this. It runs a real Linux kernel inside Windows with GPU passthrough that delivers 90-100% of native inference performance. You get Ubuntu's package manager, Docker, CUDA, and every Linux AI tool — without leaving Windows.

This guide covers the complete setup from `wsl --install` to running Ollama and llama.cpp with full GPU acceleration. Including the gotchas that will waste your afternoon if you don't know about them.

Install WSL2

Open an **Administrator PowerShell**:

```
wsl --install
```

This installs the WSL platform, the Linux kernel, and Ubuntu (default). It will prompt for a UNIX username and password on first launch.

For a specific Ubuntu version:

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

Ubuntu 24.04 LTS is recommended — well-tested with CUDA, Ollama, and Docker.

Verify WSL2 Is Active

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

If your distro shows VERSION 1, convert it:

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

Enable systemd

Needed for Docker and services. Inside WSL, edit `/etc/wsl.conf`:

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

Then restart: `wsl --shutdown` from PowerShell.

GPU Passthrough

This is the part most guides overcomplicate. The truth: **if you have a recent NVIDIA driver on Windows, GPU passthrough just works.**

How It Works

WSL2 runs a lightweight Hyper-V VM. NVIDIA's Windows driver is "stubbed" into WSL2 as `libcuda.so`. This means:

1. **Install the NVIDIA driver on Windows only** — use your normal GeForce Game Ready or Studio driver (535+ recommended, 560+ ideal)
2. **Do NOT install any NVIDIA Linux GPU driver inside WSL2** — the Windows driver handles everything
3. **Do install the CUDA Toolkit inside WSL2** — but using a special package that excludes the driver

Install CUDA Toolkit in WSL2

The critical rule: **install `cuda-toolkit-12-x` only, never the `cuda` or `cuda-drivers` meta-packages.** The full `cuda` package tries to install a Linux driver that breaks the WSL2 GPU stub.

```
# Add NVIDIA's WSL-Ubuntu repository
wget https://developer.download.nvidia.com/compute/cuda/repos/wsl-ubuntu/x86_64/cuda-wsl-ubuntu
sudo mv cuda-wsl-ubuntu.pin /etc/apt/preferences.d/cuda-repository-pin-600

# Download and install the repo package (check nvidia.com for latest version)
wget https://developer.download.nvidia.com/compute/cuda/12.6.3/local_installers/cuda-repo-wsl-u
sudo dpkg -i cuda-repo-wsl-ubuntu-12-6-local_12.6.3-1_amd64.deb
sudo cp /var/cuda-repo-wsl-ubuntu-12-6-local/cuda-*-keyring.gpg /usr/share/keyrings/
sudo apt-get update

# CRITICAL: install toolkit only, NOT the full cuda package
sudo apt-get -y install cuda-toolkit-12-6
```

Add to your `~/.bashrc`:

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

Verify GPU Access

```
nvidia-smi
```

You should see your Windows GPU from inside WSL2. If you don't, update your Windows NVIDIA driver.

Memory Configuration

This is the gotcha that catches everyone. WSL2 defaults to **50% of your system RAM**. On a 32GB system, that's only 16GB — not enough for larger models.

Configure .wslconfig

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`

Recommended Settings

Setting	Recommendation	Why
`memory`	Leave 4-8GB for Windows, give the rest	Models load into RAM before GPU offload
`swap`	8GB minimum	Prevents OOM kills during model loading

Setting	Recommendation	Why
`autoMemoryReclaim`	`dropCache`	Reclaims RAM when WSL is idle
`sparseVhd`	`true`	Prevents disk bloat when deleting models
`networkingMode`	`mirrored`	Services accessible from LAN, localhost works bidirectionally

GPU VRAM is not limited by .wslconfig. Your models get the full GPU memory. Windows uses 200-500MB of VRAM for the desktop compositor, so expect slightly less available VRAM than a headless Linux system.

Install the AI Stack

Ollama

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

Verify GPU detection:

```
ollama run llama3.2 "hello"
journalctl -u ollama --no-pager | grep -i gpu
```

You should see `NVIDIA GPU detected` with your GPU model and VRAM.

llama.cpp with CUDA

```
sudo apt update
sudo apt install -y build-essential cmake git

git clone https://github.com/ggml-org/llama.cpp.git
cd llama.cpp
cmake -B build -DGGML_CUDA=ON
cmake --build build --config Release -j$(nproc)
```

WSL2-specific fixes:

If cmake can't find CUDA, ensure `/usr/local/cuda/bin` is in your PATH (see bashrc export above).

For specific GPU architectures, set `CMAKE_CUDA_ARCHITECTURES`: 86 for RTX 30-series, 89 for RTX 40-series:

```
cmake -B build -DGGML_CUDA=ON -DCMAKE_CUDA_ARCHITECTURES="86"
```

Python + PyTorch with CUDA

```
# Install miniconda
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh -b
eval "$($HOME/miniconda3/bin/conda shell.bash hook)"
conda init

conda create -n ai python=3.11 -y
conda activate ai
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu126
```

Verify:

```
import torch
print(torch.cuda.is_available())       # True
print(torch.cuda.get_device_name(0))   # Your GPU
```

Do NOT install `nvidia-cuda-toolkit` via apt — it can overwrite the WSL2 `libcuda.so` stub. Use the WSL-specific package from NVIDIA or let PyTorch's bundled CUDA runtime handle it.

Docker with GPU Support

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

```
# Install NVIDIA Container Toolkit
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.lis
  sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg]
  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 in Docker:

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

Performance: WSL2 vs Native

Workload	WSL2 vs Native Linux
Ollama LLM inference (GPU)	90-100% — near-identical tok/s
llama.cpp with CUDA	90-100% — GPU is the bottleneck, not the OS
CUDA kernel launch	Higher latency per launch, but LLM kernels are long-running
CPU-bound tasks	85-95% — Hyper-V virtualization tax
Disk I/O (native ext4)	80-95%
Disk I/O via /mnt/c	**30-50%** — 3-5x slower, 9P protocol

The critical insight: LLM inference runs large, sustained GPU kernels. WSL2’s overhead is in kernel launch latency, which is negligible for inference. For the workloads that matter — running Ollama, llama.cpp with CUDA — WSL2 is essentially as fast as native Linux.

Ollama on Windows native vs WSL2 shows only a 10-13% difference in tok/s, and with GPU offloading the gap shrinks to near-zero because the bottleneck is GPU memory bandwidth, not the OS layer.

The Gotchas

1. File System Performance

The single biggest pitfall. Accessing files through `/mnt/c/` (your Windows drives) uses the 9P protocol and is 3-5x slower than the native ext4 filesystem.

Rule: Keep all models, datasets, and code inside WSL2's filesystem (e.g., `~/projects/`). Never run AI workloads from `/mnt/c/`.

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

2. Disk Bloat

WSL2 stores its filesystem in a VHDX file that grows but doesn't automatically shrink. After downloading and deleting large models, your disk balloons.

Fix: Enable `sparseVhd=true` in `.wslconfig` (prevents future bloat). To compact existing:

```
wsl --shutdown
# In an admin PowerShell:
diskpart
# select vdisk file="C:\Users\<you>\AppData\Local\Packages\CanonicalGroupLimited.Ubuntu24.04LTS_
# compact vdisk
# exit
```

Run `sudo fstrim /` inside WSL before compacting.

3. Networking

Default NAT mode: WSL2 ports forward to `localhost` on Windows, but aren't accessible from LAN.

Mirrored mode (recommended): Add `networkingMode=mirrored` to `.wslconfig`. Services become LAN-accessible, `localhost` works bidirectionally. Requires Windows 11 22H2+.

4. OOM Kills

If WSL2 runs out of its allocated memory, the Linux OOM killer terminates processes — usually mid-model-load. Solutions: increase `memory` and `swap` in `.wslconfig`, or use smaller quantizations.

5. The cuda Package Trap

Installing `sudo apt install cuda` inside WSL2 installs a Linux NVIDIA driver that overwrites the WSL2 GPU stub and breaks everything. Always install `cuda-toolkit-12-x` specifically.

WSL2 vs Alternatives

Scenario	Best Option
“Just want to run Ollama and chat”	Native Windows Ollama or WSL2 — both work
“I develop AI apps and need Python/CUDA/Docker”	**WSL2**
“I train models and need max GPU throughput”	Dual boot / dedicated Linux
“I need reproducible deployments”	Docker on WSL2
“Building a dedicated inference server”	Native Linux

WSL2 is the right choice for most Windows users who want to run local AI. The performance is close enough to native that the convenience of staying in Windows outweighs the 5-15% overhead.

Dual boot only makes sense if you’re training models (up to 33% faster for GPU-intensive training workloads) or need every last percentage of performance.

Quick Start Checklist

1. `wsl --install` from Admin PowerShell
2. Configure `.wslconfig` with adequate memory and `sparseVhd=true`
3. Install NVIDIA driver on Windows (535+)
4. Install `cuda-toolkit-12-6` inside WSL2 (NOT the `cuda` package)
5. Verify: `nvidia-smi` shows your GPU

6. Install Ollama: `curl -fsSL https://ollama.com/install.sh | sh`
7. Pull a model: `ollama pull qwen2.5:14b-instruct-q4_K_M`
8. Run: `ollama run qwen2.5:14b-instruct-q4_K_M`

Total time from fresh Windows install to running your first local LLM: about 20 minutes.

Source: https://insiderllm.com/guides/wsl2-local-ai-windows-guide/