Docker for Local AI: The Complete Setup Guide for Ollama, Open WebUI, and GPU Passthrough

March 4, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** The fastest path to a Docker-based local AI setup: install nvidia-container-toolkit (NVIDIA) or use the rocm image (AMD), then run a single docker-compose.yml with Ollama + Open WebUI. Models persist in a named volume, Open WebUI connects at http://ollama:11434 inside the compose network, and you access the chat UI at localhost:3000. Apple Silicon users should skip Docker entirely — macOS Docker doesn't pass through the GPU, so Ollama runs on CPU only. For everyone else on Linux with a GPU, Recipe 2 in this guide is the setup to beat.

📚 **Related:** Ollama Troubleshooting · Open WebUI Setup · VRAM Requirements · WSL2 for Local AI · Local RAG Guide

Running local AI on bare metal works fine until you need to reproduce your setup somewhere else. Or tear it down cleanly. Or run it on a headless server in a closet. Or let three other people use the same models.

That’s where Docker earns its keep. One compose file describes your entire stack (Ollama for inference, Open WebUI for the chat interface, maybe a vector database for RAG) and it runs identically on your laptop, your home server, and your coworker’s machine.

Problem is, most Docker-for-AI guides are outdated or half-working. GPU passthrough especially. Mac users spend hours trying to get it working before finding out Docker can’t access their GPU at all. Below: working compose files, GPU setup for every platform, and the production bits that keep it running long-term.

Why Docker for Local AI

Docker isn’t always the right answer. If you’re one person on one machine, just install Ollama natively and skip the container layer entirely. It’s simpler, has zero overhead, and the GPU works automatically.

Docker wins when:

- **Reproducibility.** The same `docker-compose.yml` works on your laptop, your NAS, and your friend’s server. No “it works on my machine.”

- **Clean isolation.** CUDA drivers, Python dependencies, model serving runtimes all stay inside containers. Your host system stays clean.
- **Easy teardown.** `docker compose down` removes everything. Models survive in a volume. Blow away the entire stack and rebuild it in seconds.
- **Multi-service stacks.** Ollama + Open WebUI + a vector database + a reverse proxy, all defined in one file, talking to each other on an internal network.
- **Remote access.** Run everything on a headless GPU server in a closet, chat from your laptop on the couch.

The tradeoff: Docker adds a layer of indirection. Debugging GPU issues means debugging both the container runtime and the GPU driver. Disk I/O for model loading is marginally slower through volumes. And on macOS, the situation is bad enough that it gets its own section below.

Decision matrix:

Scenario	Docker?	Why
Single user, single machine	No	Native Ollama is simpler
Headless GPU server	Yes	Remote access, clean management
Team / multi-user	Yes	Reproducible, isolated
NAS or homelab	Yes	Compose files are homelab native
Mac (Apple Silicon)	No	Docker can't access the GPU
Dev environment	Yes	Reproducible, teardown-friendly
WSL2 on Windows	Either	Docker Desktop works, but native WSL2 is simpler

---

Quick Start: Running in 5 Minutes

Ollama in One Command

```
docker run -d --name ollama -p 11434:11434 -v ollama_data:/root/.ollama ollama/ollama
```

That's it. Ollama is running. The `-v ollama_data:/root/.ollama` creates a named volume so your models survive container restarts.

Pull a Model and Test It

```
# Pull a model
docker exec ollama ollama pull llama3.2:3b

# Test it with curl
curl http://localhost:11434/api/generate -d '{
  "model": "llama3.2:3b",
  "prompt": "What is Docker in one sentence?",
  "stream": false
}'
```

If you get a JSON response with a `response` field, Ollama is working. But right now it's running on CPU. We'll add GPU passthrough in the next section.

Add Open WebUI

Create a `docker-compose.yml`:

```
services:
  ollama:
    image: ollama/ollama
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    ports:
      - "3000:8080"
    volumes:
      - open_webui_data:/app/backend/data
    environment:
      - OLLAMA_BASE_URL=http://ollama:11434
    depends_on:
      - ollama

volumes:
  ollama_data:
  open_webui_data:
```

```
docker compose up -d
```

Open `http://localhost:3000` in your browser. Create an account (first account becomes admin, and it's local-only, not sent anywhere). Select a model, start chatting.

That's the minimal setup. Three minutes, one file, two containers. Everything below makes it faster and more reliable.

GPU Passthrough

Every GPU vendor has a different path to get this working, and the error messages are unhelpful when it breaks.

NVIDIA (Linux)

Most local AI setups are NVIDIA on Linux. This is the best-documented path.

Step 1: Install nvidia-container-toolkit

```
# Add the NVIDIA repository
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] h
  sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list

sudo apt-get update
sudo apt-get install -y nvidia-container-toolkit
```

Step 2: Configure Docker

```
sudo nvidia-ctk runtime configure --runtime=docker
sudo systemctl restart docker
```

Step 3: Update your compose file

Add the `deploy` section to the Ollama service:

```
services:
  ollama:
    image: ollama/ollama
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
```

Step 4: Verify

```
docker compose up -d

# Check GPU is visible inside the container
docker exec ollama nvidia-smi

# Pull a model and check it's using GPU
docker exec ollama ollama pull llama3.2:3b
docker exec ollama ollama ps
```

The `ollama ps` output should show layers loaded on the GPU, not CPU. If `nvidia-smi` shows “command not found” inside the container, the toolkit isn’t configured correctly. Restart Docker after running `nvidia-ctk`.

AMD ROCm (Linux)

AMD requires a different Docker image and device passthrough.

```
services:
  ollama:
    image: ollama/ollama:rocm
    container_name: ollama
    ports:
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama
    devices:
      - /dev/kfd:/dev/kfd
      - /dev/dri:/dev/dri
    group_add:
      - video
      - render
```

What's different from NVIDIA:

- Use the `ollama/ollama:rocm` image, not the default
- Pass through `/dev/kfd` (kernel fusion driver) and `/dev/dri` (direct rendering) as devices
- Add the `video` and `render` groups so the container can access the GPU
- No separate toolkit install needed. ROCm support is built into the image.

If you hit "permission denied on /dev/kfd," your host user needs to be in the `render` and `video` groups:

```
sudo usermod -aG render,video $USER
# Log out and back in, then try again
```

For more on AMD GPU troubleshooting, see our ROCm GPU detection guide.

Apple Silicon: Don't Use Docker

Read this twice if you're on a Mac: **Docker Desktop for Mac does NOT pass through the GPU to containers.** Ollama inside Docker on a Mac runs on CPU only.

Your M1/M2/M3/M4 has a unified memory GPU that Ollama uses well when running natively. But Docker Desktop runs a Linux VM under the hood, and that VM has zero access to the Apple GPU. There's no workaround. No flag to set. It just doesn't work.

If you're on Mac:

1. Install Ollama natively from ollama.com
2. Install Open WebUI via pip (`pip install open-webui && open-webui serve`) or Docker (Open WebUI itself doesn't need a GPU)
3. Point Open WebUI at `http://localhost:11434`

You can still use Docker for Open WebUI and other services that don't need a GPU. Just run Ollama outside Docker. See our Mac M-Series guide for the full native setup.

Intel Arc

Intel Arc GPU support in Docker is experimental. The IPEX-LLM project provides container images with Intel GPU acceleration, but Ollama's own Intel Arc support is still limited.

If you're on Intel Arc, try running Ollama natively with the `--gpu intel` flag (added in recent builds) or use IPEX-LLM containers directly. As of early 2026, I wouldn't call this production-ready for Docker-based setups.

Verify the GPU Is Working

Whatever GPU you're running, always confirm it's actually being used:

```
# NVIDIA: check nvidia-smi inside the container
docker exec ollama nvidia-smi

# Any GPU: pull a model and check where layers loaded
docker exec ollama ollama pull llama3.2:3b
docker exec ollama ollama ps
```

The `ollama ps` output shows something like:

```
NAME              ID            SIZE    PROCESSOR    UNTIL
llama3.2:3b       a80c4f17acd5 2.0 GB 100% GPU     4 minutes from now
```

If it says "100% CPU" instead of "100% GPU," the GPU isn't being passed through. Go back and check your toolkit installation and compose configuration.

Docker Compose Recipes

Five copy-paste-ready configurations. Pick the one that matches your setup.

Recipe 1: Solo Ollama

When to use: You just want the API, no web UI. Good for development or when you're connecting from another app.

```
# docker-compose.yml — Solo Ollama with NVIDIA GPU
services:
  ollama:
    image: ollama/ollama
    container_name: ollama
    restart: unless-stopped
    ports:
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]

volumes:
  ollama_data:
```

Recipe 2: Ollama + Open WebUI (Recommended)

When to use: The standard setup. Chat interface with full model management. This is what most people should run.

```
# docker-compose.yml — Ollama + Open WebUI (NVIDIA GPU)
services:
  ollama:
    image: ollama/ollama
    container_name: ollama
    restart: unless-stopped
    ports:
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    restart: unless-stopped
    ports:
      - "3000:8080"
    volumes:
      - open_webui_data:/app/backend/data
    environment:
      - OLLAMA_BASE_URL=http://ollama:11434
    depends_on:
      - ollama

volumes:
  ollama_data:
  open_webui_data:
```

Recipe 3: RAG-Ready Stack (Ollama + Open WebUI + Qdrant)

When to use: You want to search your own documents with AI. Qdrant stores the vector embeddings, Open WebUI has built-in RAG support that connects to it.

```
# docker-compose.yml — RAG stack with Qdrant
services:
  ollama:
    image: ollama/ollama
    container_name: ollama
    restart: unless-stopped
    ports:
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]

  qdrant:
    image: qdrant/qdrant
    container_name: qdrant
    restart: unless-stopped
    ports:
      - "6333:6333"
    volumes:
      - qdrant_data:/qdrant/storage

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    restart: unless-stopped
    ports:
      - "3000:8080"
    volumes:
      - open_webui_data:/app/backend/data
    environment:
      - OLLAMA_BASE_URL=http://ollama:11434
      - RAG_EMBEDDING_ENGINE=ollama
      - RAG_EMBEDDING_MODEL=nomic-embed-text
      - VECTOR_DB=qdrant
      - QDRANT_URI=http://qdrant:6333
    depends_on:
      - ollama
      - qdrant

volumes:
  ollama_data:
  open_webui_data:
  qdrant_data:
```

After starting, pull the embedding model: `docker exec ollama ollama pull nomic-embed-text`. Then upload documents in Open WebUI's workspace panel. For a deeper dive on RAG, see our Local RAG guide.

Recipe 4: Multi-GPU (Two Ollama Instances Behind nginx)

When to use: You have two GPUs and want to run different models simultaneously, or load-balance the same model across instances. One Ollama instance per GPU, nginx routes between them.

```
# docker-compose.yml — Multi-GPU with nginx load balancer
services:
  ollama-gpu0:
    image: ollama/ollama
    container_name: ollama-gpu0
    restart: unless-stopped
    volumes:
      - ollama_data_gpu0:/root/.ollama
    environment:
      - CUDA_VISIBLE_DEVICES=0
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              device_ids: ["0"]
              capabilities: [gpu]

  ollama-gpu1:
    image: ollama/ollama
    container_name: ollama-gpu1
    restart: unless-stopped
    volumes:
      - ollama_data_gpu1:/root/.ollama
    environment:
      - CUDA_VISIBLE_DEVICES=0
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              device_ids: ["1"]
              capabilities: [gpu]

  nginx:
    image: nginx:alpine
    container_name: ollama-lb
    restart: unless-stopped
    ports:
      - "11434:11434"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - ollama-gpu0
      - ollama-gpu1

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    restart: unless-stopped
    ports:
      - "3000:8080"
    volumes:
      - open_webui_data:/app/backend/data
    environment:
      - OLLAMA_BASE_URL=http://nginx:11434
    depends_on:
      - nginx

volumes:
  ollama_data_gpu0:
  ollama_data_gpu1:
  open_webui_data:
```

Create `nginx.conf` alongside your compose file:

```
events { worker_connections 1024; }

http {
    upstream ollama {
        server ollama-gpu0:11434;
        server ollama-gpu1:11434;
    }

    server {
        listen 11434;
        location / {
            proxy_pass http://ollama;
            proxy_set_header Host $host;
            proxy_read_timeout 300s;
        }
    }
}
```

Each Ollama instance sees only its assigned GPU (via `device_ids`). The `CUDA_VISIBLE_DEVICES=0` inside the container is correct. From each container's perspective, it only has one GPU, and it's device 0. For more on multi-GPU setups, see our multi-GPU guide.

Recipe 5: CPU-Only (NAS / Old Server)

When to use: No GPU available. Running on a Synology NAS, a Raspberry Pi, or an old server. Slower but functional for small models.

```
# docker-compose.yml — CPU-only (no GPU)
services:
  ollama:
    image: ollama/ollama
    container_name: ollama
    restart: unless-stopped
    ports:
      - "11434:11434"
    volumes:
      - ollama_data:/root/.ollama
    environment:
      - OLLAMA_NUM_PARALLEL=1

  open-webui:
    image: ghcr.io/open-webui/open-webui:main
    container_name: open-webui
    restart: unless-stopped
    ports:
      - "3000:8080"
    volumes:
      - open_webui_data:/app/backend/data
    environment:
      - OLLAMA_BASE_URL=http://ollama:11434
    depends_on:
      - ollama

volumes:
  ollama_data:
  open_webui_data:
```

No `deploy` section, no GPU devices. Stick to small models: `llama3.2:1b`, `phi3:mini`, or `qwen2.5:1.5b`. Expect 2-5 tok/s on a modern CPU. Check our CPU-only guide for which models actually work.

Model Persistence and Management

Named Volumes vs Bind Mounts

The recipes above use named volumes (`ollama_data:/root/.ollama`). This is the right default. Docker manages the storage, it survives `docker compose down`, and it works everywhere.

Bind mounts (`./ollama_models:/root/.ollama`) let you control exactly where models live on disk. Use this when:

- You want models on a specific drive (NVMe vs HDD)
- You need to share models between Docker and a native Ollama install
- You want to back up models by just copying a directory

```
volumes:
  - /path/to/your/models:/root/.ollama
```

Disk Space Expectations

Model Size	Quantization	Disk Usage
3B	Q4_K_M	~2 GB
7-8B	Q4_K_M	~4.5 GB
14B	Q4_K_M	~8 GB
32B	Q4_K_M	~18 GB
70B	Q4_K_M	~40 GB

All models go into the volume under `/root/.ollama/models/` . A few models add up fast, so keep an eye on disk usage.

Pre-Pulling Models on Container Start

If you want specific models ready when the container starts (useful for automated deployments), create an entrypoint script:

```
#!/bin/bash
# pull-models.sh
ollama serve &
sleep 5

ollama pull llama3.2:3b
ollama pull nomic-embed-text

# Keep the server running
wait
```

Add it to your compose file:

```
services:
  ollama:
    image: ollama/ollama
    volumes:
      - ollama_data:/root/.ollama
      - ./pull-models.sh:/pull-models.sh
    entrypoint: ["/bin/bash", "/pull-models.sh"]
```

The models only download once. If they're already in the volume, `ollama pull` skips them.

Backing Up Your Models

```
# With named volumes
docker run --rm -v ollama_data:/data -v $(pwd):/backup alpine \
  tar czf /backup/ollama-models-backup.tar.gz -C /data .

# With bind mounts, just copy the directory
cp -r /path/to/your/models /path/to/backup/
```

Networking and Remote Access

Container-to-Container

Inside a compose network, services reach each other by name. Open WebUI connects to Ollama at `http://ollama:11434`. That's why the `OLLAMA_BASE_URL` environment variable uses the service name, not `localhost`.

Host Access

Port mapping (`11434:11434`) exposes Ollama on `localhost:11434`. Any app on the host machine can hit it.

LAN Access

By default, Ollama only listens on localhost inside the container. Docker's port mapping handles external access. If you've mapped the port, other machines on your LAN can reach it at `http://your-server-ip:11434`.

If you're running Ollama natively (not in Docker) and want LAN access, set `OLLAMA_HOST=0.0.0.0` in the environment. Inside Docker, this isn't needed since Docker's port forwarding handles it.

Reverse Proxy with Caddy

For HTTPS access (especially over the internet or tailnet):

```
# Add to your docker-compose.yml
  caddy:
    image: caddy:alpine
    restart: unless-stopped
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./Caddyfile:/etc/caddy/Caddyfile
      - caddy_data:/data
```

```
# Caddyfile
ai.yourdomain.com {
    reverse_proxy open-webui:8080
}

ollama.yourdomain.com {
    reverse_proxy ollama:11434
}
```

Caddy handles TLS certificates automatically. If you're using Tailscale, you can use `tailscale cert` instead and skip port 80/443 entirely.

Security Warning

Do not expose port 11434 directly to the internet. Ollama's API has no authentication. Anyone who can reach the port can run models, pull models, and delete models. Put it behind a reverse proxy with auth, or limit access to your local network / VPN.

Production Hardening

For setups that need to stay up: home servers, team infrastructure, always-on services.

Health Checks

```
services:
  ollama:
    image: ollama/ollama
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:11434/api/tags"]
      interval: 30s
      timeout: 10s
      retries: 3
      start_period: 30s
```

Open WebUI can use Ollama's health check as a dependency:

```
  open-webui:
    depends_on:
      ollama:
        condition: service_healthy
```

Restart Policies

`restart: unless-stopped` covers most cases. Use `restart: always` if you want the containers to survive a reboot (combined with Docker's system service being enabled).

Resource Limits

Prevent Ollama from eating all your RAM:

```
services:
  ollama:
    deploy:
      resources:
        limits:
          memory: 32g
        reservations:
          devices:
            - driver: nvidia
              count: all
              capabilities: [gpu]
```

Log Management

Docker logs grow unbounded by default. Add this to each service:

```
    logging:
      driver: json-file
      options:
        max-size: "10m"
        max-file: "3"
```

Or set it globally in `/etc/docker/daemon.json`:

```
{
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "10m",
    "max-file": "3"
  }
}
```

Auto-Updates with Watchtower

Watchtower can auto-update your containers when new images are pushed:

```
  watchtower:
    image: containrrr/watchtower
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      - WATCHTOWER_CLEANUP=true
      - WATCHTOWER_SCHEDULE=0 0 4 * * *  # 4 AM daily
```

Fair warning: Ollama image updates won't break your models (they're in the volume), but a major version jump could change model compatibility or API behavior. If stability matters more than staying current, skip Watchtower and update manually.

Common Problems and Fixes

"GPU not detected" or Running on CPU

Symptoms: `ollama ps` shows "100% CPU," `nvidia-smi` not found inside container.

Fix (NVIDIA):

1. Install nvidia-container-toolkit (see GPU section above)
2. Run `sudo nvidia-ctk runtime configure --runtime=docker`
3. Restart Docker: `sudo systemctl restart docker`
4. Make sure the `deploy.resources.reservations` block is in your compose file. The old `--gpus all` flag doesn't work in compose files.

Models Downloading Every Restart

Symptoms: `ollama pull` runs again on every `docker compose up`.

Fix: You're missing the volume mount. Make sure `ollama_data:/root/.ollama` is in both the `volumes:` section of the service and the top-level `volumes:` declaration. Also check that you're using `docker compose down` and not `docker compose down -v` (the `-v` flag deletes volumes).

Open WebUI Can't Connect to Ollama

Symptoms: "Ollama connection error" in the Open WebUI interface.

Fix: The `OLLAMA_BASE_URL` must use the Docker service name, not `localhost`. Inside the compose network, Ollama is at `http://ollama:11434`, not `http://localhost:11434`. If you're running Ollama outside Docker, use `http://host.docker.internal:11434` (Mac/Windows) or `--network=host` (Linux).

Slow on Mac

Why: Docker Desktop for Mac runs a Linux VM. That VM can't access the Apple GPU. Ollama runs on CPU inside the VM. There is no workaround.

Fix: Run Ollama natively. Use Docker only for non-GPU services like Open WebUI. See the Apple Silicon section above.

Port Conflicts

Symptoms: "port is already allocated" when starting containers.

Fix: Either stop whatever is using the port, or remap. Change `"11434:11434"` to `"11435:11434"` to use port 11435 on the host. Same for Open WebUI: `"3001:8080"` instead of `"3000:8080"`.

Permission Denied on /dev/kfd (AMD)

Symptoms: ROCm can't access the GPU, permission errors in logs.

Fix:

```
sudo usermod -aG render,video $USER
```

Log out completely and log back in. Also make sure `group_add: [video, render]` is in your compose file. See our ROCm troubleshooting guide for more.

Our Take

Docker is the right tool for local AI when you're past the "just trying it out" phase. The sweet spot is Recipe 2 (Ollama + Open WebUI) on a Linux box with an NVIDIA GPU. One compose file, two containers, full GPU acceleration, and a chat interface your non-technical family members can use.

Use Docker when you need: a headless GPU server, a team setup, homelab deployment, reproducible environments, or remote access to inference.

Skip Docker when: you're a single user on a Mac (no GPU passthrough), you just want to chat with a model (native Ollama is simpler), or you need maximum performance with zero overhead.

Mac users: install Ollama natively and save yourself the headache. Linux with an NVIDIA card: Recipe 2, bookmark it.

Image: Terminal showing docker compose up with Ollama and Open WebUI starting successfully

Image: Open WebUI chat interface running in browser with a local model selected

Image: nvidia-smi output inside a Docker container showing GPU utilization

Image: Diagram showing Docker compose stack — Ollama, Open WebUI, Qdrant containers with volume mounts and port mappings