Cross-Platform AI Dev Setup (macOS + WSL2) — Novice Guide & Advanced MLOps (Sept 2025)

> Two complementary documents in one: **Part A** is a hand-holding "remote-API only" guide for students/novices. **Part B** adds advanced, scalable MLOps patterns for researchers: Docker, GPU, local models (Ollama/vLLM/LM Studio), and routing across providers (LiteLLM/OpenRouter/etc.).

Part A — Novice Guide: Remote-API-Only Setup (macOS + Windows 11/WSL2 Ubuntu 22.04)

0) What you'll build

A safe, minimal dev environment that: - uses **remote LLM APIs** (Gemini, GLM, OpenAI mini, Groq, etc.), - works the same on **macOS** and in **WSL2/Ubuntu 22.04** on Windows 11, - lets you code in **VS Code** (or free alternatives) and a **terminal assistant (Aider)**, - keeps your **API keys out of git** and project-scoped with `direnv`.

> **Why this path?** You avoid native CUDA/Metal/DirectML complexity and driver drift. You still get powerful coding assistants (chat, inline edits, refactors) with the option to upgrade later (Part B).

---

1) Install the editor(s)

Option 1 (recommended): VS Code - macOS: download from code.visualstudio.com, then "Shell Command: Install `code`" from Command Palette. - Windows 11: install VS Code on Windows, then install the **Remote – WSL** extension. Work **inside** WSL folders (e.g., `~/code/...`) via "Open Folder in WSL".

Optional alternatives (free/low-cost): - **Continue** (open-source assistant) extension inside VS Code. - **Roo Code** extension (agentic coding, OpenAI-compatible endpoints, multi-model). - **Cline** (formerly Claude Dev) extension (agent + terminal tasks). - **Zed Editor** (macOS/Linux) with AI add-ons; or **VSCodium** if you prefer MS-branding-free VS Code.

> **Use one "agentic" assistant at a time** (e.g., Roo *or* Cline *or* Continue) to avoid key-binding conflicts and confusing overlapping UX.

---

2) Install the terminal assistant (CLI)

We'll standardize on **Aider** (open-source, works with Gemini, OpenRouter, OpenAI, local Ollama). It edits your repo via Git-style diffs and runs entirely in a terminal (including inside a VS Code Terminal).

```
# macOS (with Homebrew) or WSL2 (Ubuntu)
# Install Astral's uv for fast, isolated Python tools
curl -LsSf https://astral.sh/uv/install.sh | sh
exec $SHELL

# Install Aider as a standalone tool
uv tool install aider-install
# Bootstrap Aider (installs runtime deps + binary wrapper)
aider-install

# Verify
aider --version
```

> **Tip (Windows)**: run Aider inside **WSL** and open the same folder in VS Code (Remote – WSL). Keep repos under `~/code` on Linux, not under `/mnt/c/...` for performance.

---

3) Create API keys (remote, low-cost first)

Pick 1–2 providers to start; you can add more later: - **Gemini**: start with **Gemini 2.5 Flash/Pro** (fast + generous limits). Create `GEMINI_API_KEY`. - **OpenRouter** (aggregator): gives you one key that reaches many models/providers; good for trying GLM/Qwen/DeepSeek and free community models; set `OPENROUTER_API_KEY`. - **Groq** (optional): very fast Llama 3.1 family; set `GROQ_API_KEY`. - **Zhipu GLM** (optional): cost-effective Chinese/English models; set `ZHIPUAI_API_KEY`.

> You can always add OpenAI mini (e.g., `gpt-4o-mini`) later.

---

4) Keep secrets out of git with `direnv`

Install `direnv` and place per-project secrets in `.envrc` (not committed). Explicitly allow changes.

```
# macOS (brew) or Ubuntu (apt)
# macOS
brew install direnv
# Ubuntu (WSL)
sudo apt update && sudo apt install -y direnv

# Add to shell (zsh shown)
echo 'eval "$(direnv hook zsh)"' >> ~/.zshrc
exec $SHELL

# In your project folder
cd ~/code/my-first-llm-project
printf '%s\n'
```

```
  'export GEMINI_API_KEY="..."'
  'export OPENROUTER_API_KEY="..."'
  > .envrc

direnv allow  # must re-allow whenever .envrc changes
```

> **Best practices** - Never commit `.envrc`. Add `.envrc` and `.env*` to `.gitignore`. - Prefer **password managers/CLI** (1Password `op`, Bitwarden `bw`) and export into env vars during your session.

---

5) First workflow: Aider (terminal) + VS Code (editor)

1. Open your repo in VS Code.
2. In the built-in Terminal, run:

   ```
   cd ~/code/my-first-llm-project
   aider --model gemini  # or: aider --model openrouter/google/gemini-2.5-pro
   ```

3. Start small: “Add a `hello.py` that prints Hello with today’s date.”
4. Use Aider’s `--architect` or filespecs to contain edits. Review diffs before commit.

Inline coding inside VS Code - Install **Continue** *or* **Roo Code** *or* **Cline**. - In each extension’s settings, choose **provider: OpenAI-compatible** and set **base URL + API key** for **OpenRouter** (or direct Gemini/OpenAI). Example base URLs: - OpenRouter: `https://openrouter.ai/api/v1` - Gemini via Aider (CLI) is easiest; for in-IDE, use the extension’s native Gemini option if offered.

---

6) Profiles & model switching

Keep a per-project `.devprofile` (or VS Code **Profile**) so students can switch: - *Lightweight chat & explain:* Gemini Flash (fast, low-cost) - *Coding/refactor agent:* Roo Code with OpenRouter → choose GLM/Qwen/DeepSeek/GPT-mini - *Terminal refactor:* Aider with `--model gemini` or an OpenRouter model

> **Rule of thumb**: use **one agent extension** + **one terminal assistant** per project. Switch models via environment variables, not by reinstalling tools.

---

7) Common pitfalls & quick fixes

- **VS Code asks about Workspace Trust** → stay in **Restricted Mode** until you trust the repo; then enable when you need tasks/debugging.
- **Nothing happens in WSL** → ensure you actually opened the **WSL** folder (green corner badge in status bar) and not `C:\Users...`.

- **Aider can’t see your key** → `direnv allow` after editing `.envrc`, then `env | grep -i gemini`.
- **Rate limits/429** → switch to a different model on OpenRouter or use a fallback (see Part B LiteLLM).
- **Slow terminal I/O on Windows** → keep repos in `~/code` (ext4) inside WSL; avoid `/mnt/c`.

---

8) Upgrade path preview (what’s next)

When you’re ready: add **Docker Desktop**, try **LM Studio** or **Ollama** for local models, and place an **LLM router** (LiteLLM) in front of everything so your tools keep the same OpenAI-compatible endpoint while you swap providers behind the scenes. See Part B.

Part B — Advanced MLOps Laptop Setup (macOS + WSL2 Ubuntu 22.04)

This section layers on **containers, GPUs, local models, and a model router** so you can run the same workflows in class, on your laptop, or on a remote GPU server.

0) Goals & design patterns

- **OpenAI-compatible everywhere**: one HTTP surface for IDEs/CLIs (Continue, Roo, Cline, Aider).
- **Router in front**: **LiteLLM Proxy** locally (or server) to multiplex across providers, add fallbacks, budgets, logs.
- **Remote-first, local-optional**: default to cloud APIs; enable **Ollama/LM Studio** locally; run **vLLM** on WSL2+NVIDIA or a remote Linux GPU.
- **Immutable dev env**: use VS Code **Dev Containers** and `uv`/`fnm` for reproducible Python/Node.

---

1) System prerequisites

macOS (Apple Silicon) - Install **Homebrew**, **Docker Desktop**, **uv**, **fnm**. - Local LLMs: prefer **Ollama** or **LM Studio** (Metal-accelerated). **vLLM** is CPU-only on macOS; use a remote Linux GPU for vLLM.

Windows 11 + WSL2 (Ubuntu 22.04) - `wsl --install`, then install **Ubuntu 22.04**. - Install **Docker Desktop** and enable **WSL integration**. - **NVIDIA GPU**: install **CUDA on WSL**; verify `nvidia-smi` inside Ubuntu. Then `docker run --gpus all ...` works. - **AMD/Intel GPU**: prefer **Ollama Windows** (DirectML) or **LM Studio**; vLLM GPU is NVIDIA-first.

---

2) Language runtimes & package managers (reproducible)

Python

```
# Install uv once (macOS or Ubuntu)
curl -LsSf https://astral.sh/uv/install.sh | sh
# Per-repo workflow
echo '[project]\nname = "myproj"\n' > pyproject.toml
uv venv && uv pip install -e .  # fast, pinned envs
```

Node.js

```
# macOS
brew install fnm
# Ubuntu
curl -fsSL https://fnm.vercel.app/install | bash
exec $SHELL
fnm install --latest && fnm use --latest
```

Check in `.node-version` and `pyproject.toml` so teammates auto-sync.

---

3) Secrets & workspace isolation (beyond `.envrc`)

- Keep `.envrc` + **direnv** in dev only.
- For team/shared machines: store API keys in **1Password CLI** or **Bitwarden CLI** and export at session start.
- For risky repos: open in **VS Code Dev Containers** (mount project read-only if needed) and enable **Workspace Trust** prompts.
- Add **GitHub Secret Scanning** to repos; block pushes that include high-entropy tokens.

---

4) Local LLMs (optional but powerful)

A. Ollama (macOS/Windows/Linux)

```
# macOS
brew install ollama
# Windows (PowerShell)
winget install Ollama.Ollama
# Run a model
ollama run llama3.1:8b
# Serve to tools (OpenAI-compatible via Open WebUI or LiteLLM proxy)
```

B. LM Studio (GUI + local API) - Install app → download a model → enable the **Developer Server**. Point tools at its local endpoint.

C. vLLM (OpenAI-compatible server) - Best on **Linux + NVIDIA** (WSL2 or remote). Example:

```
# In WSL2 Ubuntu with NVIDIA
docker run --gpus all -p 8000:8000
  vllm/vllm-openai:latest
  --model meta-llama/Llama-3.1-8B-Instruct
# Now your tools can POST to http://localhost:8000/v1/...
```

- On **macOS**, use vLLM only for CPU experiments; prefer **Ollama/LM Studio** for Metal acceleration.

---

5) Put a router in front (LiteLLM Proxy)

Run a local **LiteLLM** proxy so all your IDEs/CLIs speak one API while you choose providers/models per route with fallbacks, budgets, and logs.

Quick start

```
uv tool install litellm
litellm --port 4000
  --config ~/litellm.yaml
  --num_workers 2
```

`~/litellm.yaml` **(example)**

```
model_list:
  - model_name: chat-default
    litellm_params:
      # Primary: Gemini 2.5 Pro via official SDK
      model: google/gemini-2.5-pro
      api_key: ${GEMINI_API_KEY}
  - model_name: code-fast
    litellm_params:
      # OpenRouter route to a cheap fast coder (e.g., DeepSeek Coder / Qwen / GLM)
      model: openrouter/deepseek/deepseek-coder
      api_key: ${OPENROUTER_API_KEY}
      api_base: https://openrouter.ai/api/v1
  - model_name: local-llama
    litellm_params:
      # Local LM Studio or Ollama (OpenAI-compatible endpoints)
      model: openai/llama-3.1-8b-instruct
      api_base: http://127.0.0.1:11434/v1
      api_key: dummy

router_settings:
  fallback_strategy:
    - primary: chat-default
      fallbacks: [code-fast]
  budget:
    per_user_usd_monthly: 10
```

Point **Aider / Continue / Roo / Cline** at `http://localhost:4000/v1` with an `OPENAI_API_KEY` placeholder (LiteLLM accepts it). Switch models by sending `model="chat-default"`, etc.

---

6) IDE setups (advanced)

VS Code base - Extensions: *one* of **Roo Code** / **Cline** / **Continue**, plus **GitLens**, **Docker**, **Dev Containers**, **Prettier/Black**, **Markdown All in One**. - For Roo/Cline/Continue: choose **OpenAI-compatible provider**, set base URL to **LiteLLM** (or OpenRouter). Keep actual vendor keys in `direnv`/secret vault. - Profiles: create **VS Code Profiles** per project (e.g., *Agent-Roo*, *Agent-Cline*, *No-AI*).

Cursor & Windsurf (full IDEs) - Good for long, agentic tasks and repo-wide refactors. - Still keep **Aider** handy in a Terminal for auditable diffs and model diversity. - To minimize conflicts, avoid running a VS Code agent extension *and* a full AI IDE simultaneously on the same repo.

---

7) Direct APIs vs Aggregators — guidance

- **Direct (Gemini, Groq, OpenAI, Zhipu/GLM)**
- Pros: fewer hops, consistent features, clearer SLAs/compliance.
- Cons: model lock-in; per-vendor auth/invoicing; switching models means changing client code/config.
- **Aggregators (OpenRouter) or self-hosted router (LiteLLM)**
- Pros: one client API; easy **model swaps/fallbacks**; try niche models cheaply; central metrics & rate limits.
- Cons: aggregator adds a small extra hop; occasional provider quirks; some features may lag vendor-native SDKs.

Recommended hybrid: Tools point to **LiteLLM** → LiteLLM routes to **Direct vendors** *or* **OpenRouter** depending on cost/latency/availability.

---

8) Troubleshooting playbook

- **WSL GPU not detected in containers**: update Windows NVIDIA driver, run `wsl --update`, verify `nvidia-smi` inside WSL, then `docker run --gpus all nvidia/cuda:12.4.0-base-ubuntu22.04 nvidia-smi`.

- **macOS + vLLM is slow**: expected (CPU only). Use **Ollama/LM Studio** or a remote Linux GPU.
- **Router 429/5xx**: enable LiteLLM fallbacks (`fallback_strategy`) and caching; add multiple routes for redundancy.
- **IDE conflicts**: disable extra AI extensions; keep a **clean profile** per tool.
- **Dependency drift**: rebuild Dev Container; `uv pip compile` lockfiles; pin Node version via `.node-version`.
- **OpenRouter 401/403**: confirm `OPENROUTER_API_KEY` is set in the same shell as your tool; some apps need restart.
- **direnv not loading**: after editing `.envrc` you must run `direnv allow` again; ensure your shell has the `direnv hook` line.

---

9) Scaling the workflow (teams, courses, labs)

- **Templates**: provide a course repo with:
- `.devcontainer/` (CUDA on WSL template; non-GPU macOS template)
- `pyproject.toml`, `.node-version`, `.editorconfig`
- `Makefile` targets: `make setup`, `make test`, `make run` (students copy/paste less)
- **Central router**: host **LiteLLM** in the lab (reverse proxy + rate limits + per-student budgets). Students set a single base URL.
- **Secret hygiene**: use organization GitHub **secret scanning + push protection**; provide a canned `.gitignore` with `.envrc`.
- **Cost controls**: default to low-cost models (Gemini Flash, GLM/Qwen/DeepSeek via OpenRouter). Reserve higher-end models per-assignment.

---

10) Real-world workflow recipes

A. Classroom starter (no GPU, 60-min boot-up) 1) Students install VS Code + one assistant (Continue), `direnv`, and Aider. 2) Instructor gives an `OPENROUTER_API_KEY` or lab LiteLLM URL. 3) Assignments: prompt engineering in README; small refactors via Aider diffs.

B. Researcher portable lab (laptop + optional remote GPU) 1) Dev Container with Python (uv) + Node (fnm). 2) Local router (LiteLLM) with routes: `chat-default`→Gemini; `code-fast`→OpenRouter (GLM/Qwen/DeepSeek); `local-llama`→Ollama. 3) For heavy evals: spin a remote GPU with **vLLM**; point LiteLLM route to that IP.

C. Multi-agent experiments - Keep agents modular (CLI Aider for patching; IDE agent for planning; local LLM for offline deterministic tests). Log each agent’s prompts/responses; route all through LiteLLM for uniform telemetry.

Appendices

Appendix A — Minimal checklists (copy/paste)

macOS novice (15 min)

```
# Tools
brew install --quiet direnv fnm ollama || true
curl -LsSf https://astral.sh/uv/install.sh | sh

# Project skeleton
mkdir -p ~/code/hello && cd $_
python3 - <<'PY'
print("hello")
PY

# Secrets
printf '%s\n' 'export GEMINI_API_KEY=...' > .envrc
printf '%s\n' 'export OPENROUTER_API_KEY=...' >> .envrc
direnv allow

# Aider
uv tool install aider-install && aider-install
```

Windows 11 (WSL2 Ubuntu) novice

```
wsl --install  # if not already
# In Ubuntu terminal
sudo apt update && sudo apt install -y direnv
curl -LsSf https://astral.sh/uv/install.sh | sh
exec $SHELL
mkdir -p ~/code/hello && cd $_
printf '%s\n' 'export OPENROUTER_API_KEY=...' > .envrc && direnv allow
uv tool install aider-install && aider-install
```

Appendix B — VS Code extension settings (OpenAI-compatible)

- **Continue**: Settings → Provider → OpenAI-compatible; Base URL = your LiteLLM or OpenRouter endpoint; API key = env var reference.
- **Roo Code**: Settings → Provider = Custom (OpenAI-compatible). Set Base URL + API key; optionally define workspace rules/modes in `.roo/`.
- **Cline**: Settings → Model Provider = Custom / OpenAI-compatible; Base URL + API key; enable Terminal Access with caution on untrusted repos.

Appendix C — Example `.envrc`

```
# Remote vendors
export GEMINI_API_KEY="..."      # Google Gemini
export OPENROUTER_API_KEY="..."  # OpenRouter aggregator
export GROQ_API_KEY="..."        # Groq (fast Llama)
# Local servers (no keys needed, placeholders ok)
export OPENAI_API_KEY="local"
# For OpenAI-compatible routers that require a var
```

Appendix D — Model selection tips (cost-savvy)

- Default: **Gemini 2.5 Flash/Pro** for speed/price.
- Code edits/refactors: try **DeepSeek Coder/Qwen/GLM** via **OpenRouter**.
- Long-context doc Q&A: **Gemini** or **GLM/Qwen long-context** variants.
- Offline/air-gapped: **Ollama** with **Llama 3.1 8B/70B** GGUF.

You now have two lanes: a simple, student-friendly remote-only setup (Part A) and a production-ready laptop MLOps stack (Part B) that scales from local to remote GPUs without rewriting your tools.