CYPHERPUNK SCHOOL — SOVEREIGNTY SERIES

Your Sovereign AI Coding Agent

The Local Fallback They Can’t Revoke

Run a tool-capable coding agent on your own hardware — offline, ungated, $0/query

A practical guide to digital sovereignty

Cypherpunk School

cypherpunkschool.com

Sovereignty Series • June 2026

Contents

1. Why This Matters: The Night They Flipped the Switch

I found out the way you find out about most things that matter: by accident, in the middle of doing something else.

On **June 12, 2026**, the U.S. Commerce Department issued an export-control directive ordering Anthropic to suspend access to two of its newest frontier models — Fable 5 and Mythos 5 — for any foreign national. Inside the U.S. or outside it. There is no way to comply with that selectively without locking out a huge share of users and staff. So Anthropic did the only thing it could: **it shut the models off for everyone, worldwide,** about a day after they launched.

The stated trigger was a jailbreak. Anthropic publicly called the vulnerability "minor" and pointed out the same capability exists in shipped models elsewhere. The government ordered a takedown anyway.

Sit with the shape of that for a second. A tool millions of people were using vanished for the entire planet, overnight, by *letter*, over a bug the vendor itself calls minor.

This is not new. In the 1990s the U.S. government classified strong encryption as a *munition* — literally a weapon. They investigated Phil Zimmermann for years over PGP. The courts eventually ruled that code is speech and the export rules were unconstitutional. The math was already out; you cannot un-publish a number. Today that "munition" secures your bank login and every message you send.
What happened on June 12, 2026 is the same move, ported to a new medium. A capability the State couldn't fully control, declared a risk, restricted by nationality and fiat, overnight.

There is a second gate working alongside the government one: **price.** The best models quietly migrate behind subscription tiers that most people can't afford. Government pressure from one side, paywalls from the other — the public in the middle, losing access either way.

The lesson is the oldest one in this playbook: none of this touches you if you own the tool.

Local open-weight models don't phone home. They don't wait for permission. No letter from Commerce reaches them. This guide shows you how to build a local AI coding agent that is **revocation-proof** — a real floor you own, not a rented foundation someone else can pull out from under you.

What this guide covers. By the end you will have:

- Ollama running as a system service with a tool-capable coder model
- The two silent killers patched (wrong model, wrong context window)
- Two code-agent harnesses wired to your local model: Aider and OpenCode
- A lived proof-of-concept: a Python CLI written, tested, and self-debugged by the local agent — fully offline, $0

2. The Foundation: Ollama + a Tool-Capable Model

Ollama is a local model runtime. It serves models through an OpenAI-compatible API so any tool designed for GPT-4 or Claude can point at it instead. Models run on your GPU (or CPU if needed); nothing leaves your machine.

Install and configure Ollama

Install Ollama from ollama.com or via the one-liner for Linux:

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

By default Ollama only listens on `localhost:11434`. To expose it on your LAN (so a second machine running the harness can use the GPU host):

```
# Create a systemd override drop-in
sudo systemctl edit ollama
```

In the editor that opens, add:

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

Then reload and restart:

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

Pull a model

```
ollama pull qwen3-coder:30b
```

This is a 30B Mixture-of-Experts coder — roughly 18 GB at Q4 quantization, 256K context window. It is what was used for the live tests in this guide. Smaller variants (`qwen3-coder:8b`, `:14b`) trade capability for VRAM.

VRAM reality check. A 30B Q4 model takes roughly 18 GB of GPU memory. On a 24 GB card you run *one* big model at a time. Free VRAM before loading it — stop other GPU-resident services first.
Honest ceiling: a local 30B coder handles roughly 75–80% of daily agentic work (single-file edits, short tool chains, boilerplate generation). The hard 20% — deep multi-file refactors, subtle API-design questions, long-context reasoning — a frontier cloud model still wins. The goal here is an *owned floor*, not a replacement for everything.

The #1 lesson: verify tool support before you trust a model

Not every model that runs in Ollama can act as an agent. Agents require **tool calling** — the ability to request a function call and parse the result. Without it you get a chatbot, not an agent. Harnesses will fail in confusing ways and you will waste hours debugging the wrong thing.

Before committing to any model:

```
ollama show qwen3-coder:30b
```

Look for the `Capabilities` block in the output. You want to see `tools` listed:

```
Capabilities
    completion
    tools        <-- THIS. Required for agentic use.
```

Real cautionary tale. While building this guide, a different model was tried first — and its published specs *said* it supported tool calling. But `ollama show` told the truth: its capabilities listed only `completion, vision` — no `tools`. The harness accepted the model name without complaint, then failed at runtime with "model does not support tools." The lesson isn't "that model is bad" — it's that **a model's documentation and its actual Ollama packaging can disagree.**
Always check `ollama show` before wiring a new model into your agent stack.

The #2 lesson: the silent context killer (`num_ctx`)

This one is nastier because it produces no error at all. Ollama's **default context window is approximately 4,096 tokens** — fine for a quick chat, but fatal for agentic loops. An agent accumulates a growing conversation: instructions, tool calls, file contents, test output. At 4K tokens the context silently truncates. The model appears to forget what it just did, contradicts itself, loops, or generates nonsense. There is no warning. The logs show nothing. It just looks dumb.

Fix it with a systemd drop-in:

```
sudo mkdir -p /etc/systemd/system/ollama.service.d/
```

Create `/etc/systemd/system/ollama.service.d/context.conf`:

```
[Service]
Environment="OLLAMA_CONTEXT_LENGTH=32768"
```

Reload and restart:

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

Verify the variable is live:

```
systemctl show ollama -p Environment | tr ' ' '\n' | grep CONTEXT
```

You should see `OLLAMA_CONTEXT_LENGTH=32768`.

> **Why 32768?** It is a safe middle ground. 4K is too small for any real agentic loop. 128K–256K is possible but requires proportionally more VRAM and slows inference significantly. 32K fits a substantial coding session without hammering the GPU.
> For the `qwen3-coder:30b` model's full 256K context you would need to set `OLLAMA_CONTEXT_LENGTH=131072` and have headroom in VRAM. Start at 32K; increase if you regularly hit limits.

3. Harness A: Aider (Tight Pair-Programmer)

Aider is a terminal-based AI pair programmer. It has deep git integration: every edit is shown as a diff before it lands, commits can be automatic or manual, and there is a first-class undo command. It is *tight* — you stay in control of which files are in context and you see exactly what changed. If you want a precise co-author rather than an autonomous executor, this is the harness.

Install

```
pipx install aider-chat
```

> `pipx` installs aider-chat into its own isolated Python environment and puts the `aider` binary on your PATH. If you do not have `pipx`: `pip install --user pipx && pipx ensurepath`.

Point Aider at Ollama

Aider needs two environment variables and a model flag. Set them in your shell or a `.env` file in the project root:

```
export OLLAMA_API_BASE=http://localhost:11434
export OLLAMA_CONTEXT_LENGTH=32768
```

Launch with:

```
aider --model ollama_chat/qwen3-coder:30b <files>
```

Or for the GPU on another machine on your LAN:

```
export OLLAMA_API_BASE=http://<GPU_HOST_LAN_IP>:11434
aider --model ollama_chat/qwen3-coder:30b <files>
```

Persistent configuration with `.aider.conf.yml`

Drop a config file at `~/.aider.conf.yml` (global) or in the project root (per-project):

```
model: ollama_chat/qwen3-coder:30b
auto-commits: true
read:
  - CONVENTIONS.md
```

The `read` key tells Aider to inject `CONVENTIONS.md` into every session as a read-only context file. This is your persistent rules layer — naming conventions, patterns you always want followed, architectural constraints. The model reads it at every turn without being told to.

Key Aider commands

Command	What it does
`/add <file>`	Add a file to the editable context
`/drop <file>`	Remove a file from context
`/ask <question>`	Ask without editing (read-only answer)
`/architect <task>`	Use a smarter/different model for planning
`/run <command>`	Run a shell command and feed output to the model
`/test`	Run your test suite; feed failures to the model for fixing
`/model <name>`	Swap models mid-session
`/diff`	Show the current uncommitted diff
`/undo`	Undo the last commit

The edit–run–fix loop. The core Aider workflow: `/add` the file you're working on, describe the change, inspect the diff, accept it, then `/run` or `/test` to validate. If tests fail, Aider reads the output automatically and proposes a fix. One loop. No copy-paste.
`/model` lets you swap in a general-purpose model for architecture questions and back to the coder for implementation — without leaving the session.

4. Harness B: OpenCode (Autonomous Loop)

OpenCode is a more autonomous TUI agent. Where Aider acts as a precise pair where you confirm each edit, OpenCode runs multi-step loops: it writes code, executes it, reads the output, fixes errors, re-runs, and keeps going until the task is done or it gets stuck. Think of it as the difference between a co-pilot and a junior engineer you've briefed and set loose.

Install OpenCode

```
npm install -g opencode-ai
```

Or via the official install script if available at opencode.ai.

Configure a local Ollama provider

OpenCode's config lives at `~/.config/opencode/opencode.jsonc`. By default it only shows cloud providers in the model picker. You need to declare your local Ollama instance as a custom provider:

```
{
  "provider": {
    "ollama-local": {
      "npm": "@ai-sdk/openai-compatible",
      "name": "Ollama (Local)",
      "options": {
        "baseURL": "http://localhost:11434/v1"
      },
      "models": {
        "qwen3-coder:30b": {
          "name": "Qwen3-Coder 30B",
          "tools": true,
          "temperature": 0
        }
      }
    }
  }
}
```

For the GPU on a separate machine:

```
"baseURL": "http://<GPU_HOST_LAN_IP>:11434/v1"
```

Gotcha: the model picker. Before you add this config, OpenCode's model dropdown only shows cloud providers (OpenAI, Anthropic, etc.). The local provider does not appear in the UI until it is declared in `opencode.jsonc`. Once declared, restart OpenCode and your Ollama models will appear in the picker. If the model is not listed: check the config path, check the `baseURL`, and confirm Ollama is running (`systemctl status ollama`).

Run it

Launch `opencode` from a project directory. Select your local model in the picker. Describe the task. OpenCode autonomously runs the edit–run–fix loop: it writes code, executes it, reads the output, patches failures, re-tests, and repeats.

Proof it works: a fully autonomous build session.
`qwen3-coder:30b` was given a single task: "Write a `diceware.py` CLI that generates passphrases from a wordlist."
It completed the task in **8 autonomous steps** over approximately 41 seconds:

1. Wrote the initial implementation with `argparse`, wordlist loading, and a `--selftest` flag
2. Ran `python diceware.py --selftest` — passed
3. Tested `--words 6` flag — passed
4. Tested `--capitalize` flag — **found a bug** (no output produced)
5. Self-diagnosed: capitalization logic was applied after joining words, not per-word
6. Patched the bug autonomously
7. Re-ran `--capitalize` — passed
8. Ran full flag matrix (`--words`, `--separator`, `--capitalize` in combination) — all passed

Critically: it **used Python's `secrets` module**, not `random` — the cryptographically secure choice, unprompted. Fully local, $0 per query, no API key, no network connection required.

5. Aider vs. OpenCode: Which to Use

	Aider	OpenCode
Control	High — you see every diff before it lands	Lower — runs autonomously
Best for	Pair editing, targeted changes, code review	Multi-step tasks, build-run-fix loops
Git integration	First-class (`/undo`, auto-commits)	Handles execution context
Conventions	`CONVENTIONS.md` injected every turn	Prompt-level instructions
Feel	Precise co-author	Junior engineer you've briefed

Both run 100% locally. Both cost $0 per query. Both work with the same Ollama backend. Use Aider when you want to *stay in the loop* on every edit. Use OpenCode when you want to *describe a task and come back to results.*

6. Quick Reference

Setup checklist

1. **Install Ollama** and pull `qwen3-coder:30b`
2. **Check tool support:** run `ollama show qwen3-coder:30b` — confirm `tools` appears under Capabilities
3. **Fix context:** create the systemd drop-in at `/etc/systemd/system/ollama.service.d/context.conf` with `OLLAMA_CONTEXT_LENGTH=32768`; reload; verify
4. **Install Aider:** `pipx install aider-chat`; set `OLLAMA_API_BASE` (see §3); launch with the model flag shown in §3
5. **Install OpenCode:** `npm install -g opencode-ai`; declare the `ollama-local` provider in the config file at `~/.config/opencode/opencode.jsonc` (see §4); restart; select the model in the picker

Common commands

Task	Command
Check Ollama status	`systemctl status ollama`
List loaded models	`ollama list`
Verify capabilities	`ollama show <model>`
Check context env	`systemctl show ollama -p Environment | tr ' ' '\n' | grep CONTEXT`
Start Aider (local)	`aider --model ollama_chat/qwen3-coder:30b <files>`
Start Aider (remote GPU)	`OLLAMA_API_BASE=http://<GPU_HOST_LAN_IP>:11434 aider ...`
Start OpenCode	`opencode` (from project dir)

7. You Own This One

The night Fable 5 disappeared, it was gone for *everyone* — not just people in restricted jurisdictions, not just people who had violated terms. Everyone. No warning, no appeal.

That's the clearest demonstration yet of the structural risk in building your work on tools you don't control. Treat centralized AI as a **convenience, never a foundation**. For anything you can't afford to have taken away: run local, own your stack, keep a fallback.

What you have built here is that fallback. A coding agent running on your own hardware, serving your own requests, answering to no API key and no export-control directive. It handles the 75–80% of daily work where a frontier cloud model would have been overkill anyway. It is ready when the cloud is not.

Sovereignty isn't paranoia. It's just refusing to build your house on land you're only renting.

Want more?

More guides on self-hosting, privacy, local AI, and digital sovereignty:

cypherpunkschool.com
