Local Models and Offline Agents

Last modified: 2026-09-13 00:11:34 (PDT)

Cloud model providers offer state-of-the-art capability, but privacy constraints, HIPAA data handling, air-gapped environments, and cost considerations often require running models locally or offline. This chapter covers running coding agents without outside network calls, connecting open harnesses to local Ollama and OpenRouter endpoints, steering Claude Code with non-Anthropic models, and evaluating small local models for autonomous coding loops.

1 Running Coding Agents Offline

Some environments restrict or prohibit internet access—high-performance computing (HPC) clusters, hospital networks, or air-gapped research servers may block connections to cloud AI providers. Running a local AI model lets you use coding assistance in these settings without sending code to external servers, which also addresses data-privacy concerns when working with sensitive or confidential data.

> **Trade-offs of Local vs. Cloud Models**
>
> Local models work best with a GPU (roughly 8 GB VRAM or more for smaller models); CPU-only inference is possible but significantly slower, and hardware needs vary with model size and quantization. They are generally less capable than frontier cloud models, and may produce lower-quality results on complex tasks. For routine work in fully connected environments, cloud-based agents remain the better choice. Use local models when network access or data-privacy policies require it.

Running a Local Model with Ollama

Ollama is a common way to run open-weight AI models locally. It packages models and a simple API server into a single tool and is available for macOS, Linux, and Windows.

Install Ollama:

> **Caution**
>
> Before running any remote install script, review it first. The real risk is piping `curl` straight into `sh`/`bash` (`curl ... | sh`), which executes unreviewed code. Save the script, read it, then run it: `curl -fsSL https://ollama.com/install.sh -o install.sh && less install.sh` (paging the saved file rather than piping `curl` into `less`, which can behave oddly in some terminal emulators). Alternatively, use your system package manager (e.g., `brew install ollama` on macOS) or follow the manual installation steps on the Ollama releases page.

```
# macOS / Linux: download, review, then run (do not chain these into one command)
curl -fsSL https://ollama.com/install.sh -o install.sh
less install.sh   # review the script before running it (see caution above)
sh install.sh
```

On Windows, download the installer from https://ollama.com/download.

Pull a code-focused model:

```
# Smaller, faster; works on most machines with a modern GPU or Apple Silicon
ollama pull qwen2.5-coder:7b

# More capable; larger memory footprint
ollama pull qwen2.5-coder:32b

# Alternatively, a general-purpose model (70B variant; needs a high-memory GPU)
ollama pull llama3.3
```

The VRAM each model needs depends on its size and quantization and changes as models are re-quantized—check the Ollama model library for current requirements. As a rough guide, smaller (7B) models run on consumer GPUs with around 8 GB of VRAM, while larger (32B and 70B) models need substantially more and may not fit on a single GPU.

> **A good completion model is not necessarily a usable agent**
>
> The models above are strong at *writing code* when you ask them to. That is a different skill from **tool calling** — emitting a well-formed request to read a file or run a command, and then using the result. Inline completion and chat need only the first. Anything autonomous needs the second, because an agent that cannot call a tool cannot read your repository at all.
> The two come apart in practice, and the failure is subtler than a model simply refusing. Tested against a single-function tool schema on a 24 GB M2, `qwen2.5-coder:14b` returned

`finish_reason: stop` with an empty `tool_calls` field on four attempts out of four. It had not ignored the request: it wrote a correct tool call as ordinary prose in the `content` field,

```
{"name": "read_file", "arguments": {"path": "src/main.py"}}
```

which is the right JSON in the wrong place. A harness looks for `tool_calls`, finds nothing, and treats the turn as a plain reply, so the tool never runs. `granite4:7b-a1b-h`, at half the parameter count, returned a well-formed call in the `tool_calls` field three times out of three and completed a full multi-turn round trip.
An advertised `tools` capability is necessary but not sufficient, so do not settle the question with `ollama show`. On the same machine `qwen2.5-coder:14b` lists `tools` among its capabilities and still cannot be driven by a harness, for the reason above. Test it yourself with one request before building a loop on it:

```
RESP=$(curl -s -w '\n%{http_code}' \
  http://localhost:11434/v1/chat/completions -H 'content-type: application/json' -d '{
  "model": "granite4:7b-a1b-h", "stream": false,
  "messages": [{"role": "user", "content": "What is in src/main.py? Use the tool."}],
  "tools": [{"type": "function", "function": {"name": "read_file",
    "parameters": {"type": "object", "properties": {"path": {"type": "string"}},
    "required": ["path"]}}}]}')

CODE=$(printf '%s' "$RESP" | tail -1)
BODY=$(printf '%s' "$RESP" | sed '$d')

if [ "$CODE" != "200" ]; then
  echo "request failed (HTTP $CODE): $BODY"        # bad tag, tools unsupported, server dow
elif printf '%s' "$BODY" | grep -q '"tool_calls"'; then
  echo "usable as an agent"
else
  echo "no tool call; completion model only"
  printf '%s' "$BODY" | grep -o '"content":"[^"]*"' | head -c 200
fi
```

Check the status code separately from the result. A request that simply errored — a mistyped tag, a model the server rejects for tool use, a server that is not running — produces the same silence as a model that declined to call the tool, and only one of those is a fact about the model. Printing the `content` field on the no-call branch is what distinguishes a model that ignored the tools from one that described the call in prose instead of emitting it, which is the `qwen2.5-coder` case above.

Start the Ollama server:

```
ollama serve
```

By default the server listens at `http://localhost:11434`.

Connecting Positron Assistant to Ollama

Positron supports Ollama natively through the OpenAI-compatible API endpoint that Ollama exposes.

1. Open the Command Palette with `Cmd+Shift+P` (macOS) or `Ctrl+Shift+P` (Windows/Linux).
2. Run **Positron Assistant: Configure Language Model Providers**.
3. Select **Ollama** (or **Custom/OpenAI-compatible** if Ollama is not listed).
4. Set the base URL to `http://localhost:11434/v1` and leave the API key blank, or, if the client rejects an empty field, enter any placeholder value such as `ollama`.
5. Choose your model from the model list (e.g., `qwen2.5-coder:7b`).

Once configured, Positron Assistant will send requests to your local Ollama server instead of a cloud provider.

Connecting VS Code to Ollama via Continue

Continue is an open-source VS Code extension that supports Ollama and many other local and cloud backends.

1. Install the **Continue** extension from the VS Code marketplace.
2. Open the Continue sidebar and click the model selector.
3. Choose **Ollama** and select a model (Continue detects running Ollama models automatically).

Continue provides inline completions and a chat panel, similar to GitHub Copilot, but routed entirely to your local model.

Running an Autonomous Coding Agent with Aider

The editor integrations above provide inline completion and chat. For an autonomous agent that reads your files, proposes edits across a whole repository, and commits them to git—the local counterpart to a cloud coding agent—`aider` works directly against Ollama.

Install it in an isolated environment so its dependencies do not collide with other tools:

```
# Recommended: isolated install with pipx
pipx install aider-chat

# Or into your user environment with pip
python3 -m pip install --user aider-chat
```

Point it at Ollama and choose a model with the `ollama_chat/` prefix, which gives better results in `aider` than the plain `ollama/` prefix:

```
export OLLAMA_API_BASE=http://127.0.0.1:11434
aider --model ollama_chat/qwen2.5-coder:7b
```

! Raise the Ollama context window

Ollama's default context window is small, which silently truncates your code and makes the model look far less capable than it is. This is the single most common mistake when pairing `aider` with Ollama.
The default is not a fixed number. Ollama picks it from the memory it detects, as its own `ollama serve --help` states:

```
OLLAMA_CONTEXT_LENGTH   Context length to use unless otherwise specified
                        (default: 4k/32k/256k based on VRAM)
```

Do not assume you landed in a generous tier. A 24 GB Apple-silicon machine gets **4096 tokens**, not the 32k its total memory suggests, because only about 75% of unified memory is addressable by the GPU and the tier boundary sits above that share. Check what you actually got rather than inferring it — `ollama ps` prints the context of each loaded model:

```
ollama ps
# NAME                  SIZE     PROCESSOR    CONTEXT
# qwen2.5-coder:14b     9.5 GB   100% GPU     4096
```

There are three ways to raise it, and they differ in which clients they reach:

- **`OLLAMA_CONTEXT_LENGTH`** on the server, which sets the default for everything. Note that the macOS menu-bar app starts the server with its own environment, so exporting the variable in your shell does not reach it; this route applies when you run `ollama serve` yourself.
- **A `num_ctx` parameter sent per request**, which is what `aider` does through `~/.aider.model.settings.yml`:

```
- name: ollama_chat/qwen2.5-coder:7b
  extra_params:
    num_ctx: 32768
```

- **A Modelfile that bakes the context into a derived model**, which is the only one of the three that reaches clients that cannot send `num_ctx` themselves:

```
printf 'FROM granite4:7b-a1b-h\nPARAMETER num_ctx 32768\n' > Modelfile
ollama create granite4-32k -f Modelfile
```

  The derived model shares weight blobs with its base, so it costs no extra disk.

Context is not free. Raising a 14B model from 4k to 32k on a 24 GB M2 took its resident size from 9.5 GB to 15 GB, about 5.5 GB of key-value cache, so pick the largest value that still leaves the weights and the cache in GPU memory and confirm with `ollama ps` that `PROCESSOR` still reads `100% GPU`.

To avoid passing flags every time, set defaults in a config file at `~/.aider.conf.yml`:

```
model: ollama_chat/llama3.2:3b
set-env:
  - OLLAMA_API_BASE=http://127.0.0.1:11434

# Automatically load shared lab instructions and agent conventions
read:
  - ~/path/to/ai-config/AGENTS.md
```

Keep instruction files lean for small local models

While cloud frontier models handle tens of thousands of tokens of system prompt easily, small local models (1.5B–7B) lose speed and instruction fidelity when loaded with large instruction files. A multi-page documentation bundle (such as a 95 KB `CLAUDE.md`) will consume most of an SLM's active context window and cause severe prompt-ingestion delays. For local SLMs, point `read` at a concise, focused summary file (such as `AGENTS.md` or a project-specific rules snippet) rather than the full multi-tool configuration suite.

Running Aider with a Graphical User Interface (GUI)

While `aider` is primarily used from the command line, it includes a built-in browser-based GUI:

```
aider --model ollama_chat/llama3.2:3b --gui
```

This launches a local web application in your default browser with:

- A visual chat timeline and diff review panel
- Point-and-click file selectors for adding files into the active context
- Speech-to-text voice input support

Integrating Aider into VS Code

You can bring Aider directly into VS Code or Positron through three workflows:

1. **Integrated Terminal**: Run `aider` in the built-in terminal (`Ctrl+`). Edits and Git commits made by Aider immediately reflect in your editor tabs.
2. **VS Code Simple Browser**: Run `aider --gui` in the terminal, open the Command Palette (`Cmd+Shift+P`), and run **Simple Browser: Show** with `http://localhost:8501` to dock the Aider interface side-by-side with your code.
3. **VS Code Extension**: Install a community integration such as **Aider** (by MattFlower) or **Aider Composer** (by lee2py) from the marketplace for dedicated sidebar controls and editor context-menu actions.

`aider` can also split the work between two models in "architect" mode: a larger model plans the change (the architect), and a second model applies the edits (the editor).

```
aider --architect \
  --model ollama_chat/qwen2.5-coder:32b \
  --editor-model ollama_chat/qwen2.5-coder:7b
```

This can improve results on multi-step changes, but on a machine without a strong GPU it roughly doubles the time per turn, because the two models take turns and their weights are swapped in and out of memory. Reserve it for genuinely tricky changes; for small edits, a single model is faster.

> **!** Set `edit_format: whole` for a small model
>
> `aider` asks the model to express an edit either as a SEARCH/REPLACE block (`diff`, the default for most models) or by rewriting the file (`whole`). Producing an exact SEARCH/REPLACE block is a demanding format, and small models are unreliable at it. Measured on a 24 GB M2 with `granite4:7b-a1b-h` at 32k context, same prompt each time:

File	`edit_format: diff`	`edit_format: whole`
One function	3 of 3 correct	3 of 3 correct
Two functions, one to be left alone	0 of 3 correct	3 of 3 correct

The two-function failure is worth dwelling on, because it is not the failure you would expect. The model did not refuse the edit or produce a broken file. In all three `diff` runs it fixed the target function correctly **and silently deleted the other one**, then committed with a message naming only the intended fix. Nothing in the commit message, the exit status, or the model's own summary mentioned the deletion.
That is the hazard of an unattended loop in its most concrete form: a step that reports success while destroying work, leaving a wrong state as the premise for every step after it. It is also why committing after every step matters — `git` held the original, so the damage was one `git revert` away rather than lost.
Set the format per model:

```
- name: ollama_chat/granite4-32k
  edit_format: whole
```

`whole` costs more tokens per edit, since the model rewrites the whole file, which is a real cost on large files. Larger models generally handle `diff` correctly; re-measure rather than assuming either way.

Connecting Claude Code to Ollama

Ollama also serves an **Anthropic-compatible** endpoint at `/v1/messages`, alongside the OpenAI-compatible one used above. Any client that speaks the Anthropic API can therefore be pointed at a local model, including Claude Code itself, with no proxy in between. Check that the endpoint answers before wiring anything to it:

```
curl -s http://localhost:11434/v1/messages -H 'content-type: application/json' \
  -d '{"model":"granite4-32k","max_tokens":50,
       "messages":[{"role":"user","content":"Say OK only."}]}'
```

Point Claude Code at it with three environment variables:

```
export ANTHROPIC_BASE_URL=http://localhost:11434
export ANTHROPIC_AUTH_TOKEN=ollama   # any non-empty value; Ollama ignores it
export ANTHROPIC_API_KEY=""          # ensure no real key is sent to localhost
claude --model granite4-32k
```

Set these in a wrapper script rather than in your shell profile, so that plain `claude` keeps using the cloud and a separate command uses the local model.

> ⚠ A heavyweight harness can defeat a small model
>
> This works, but expect worse results than the same model gives through `aider`, and understand why before blaming the model.
> A harness spends context before your task does. Claude Code sends a long system prompt and a schema for every tool it exposes, and any Model Context Protocol (MCP) servers you have configured add their own schemas on top. Against a 32k local context that overhead is a large fraction of the budget, and a small model handles it poorly.
> Observed on a 24 GB M2 with `granite4:7b-a1b-h`, asking only that it read one file and comment on one function:
>
> - With a GitHub MCP server loaded, the model ignored the question and produced a paragraph about missing credentials.
> - With MCP disabled, it invented a task list whose contents were copied from the description of a tool it had been shown.
> - With the tool surface cut to `Read`, `Grep`, and `Glob`, it still ignored the question and asked what it should work on.
>
> The same model, same context, through `aider`, fixed a real bug and committed it in 17 seconds. Swapping in a 12B model produced no answer at all in ten minutes, because processing that much prompt at 10 tokens per second is simply slow.
> Two practical rules follow. Shrink the tool surface a local model is shown — `--strict-mcp-config --mcp-config '{"mcpServers":{}}'` loads no MCP servers, and `--allowed-tools` narrows the built-ins. And tell the harness the real context size, since Claude Code assumes a 200k window for a model it does not recognize and would let the conversation grow far past what the model can hold:
>
> ```
> export CLAUDE_CODE_MAX_CONTEXT_TOKENS=32768
> ```
>
> For autonomous work on a small local model, prefer a light harness such as `aider`. Reserve this route for using a familiar interface offline, not for getting the best out of the hardware.

Falling Back Between Cloud and Local Automatically

Air-gapped work aside, the common case is a laptop that is usually online but sometimes is not—on a plane, behind a flaky hospital network, or temporarily rate-limited by a cloud provider. You can keep a coding agent working across these gaps by putting a cloud model and a local model behind one endpoint and falling back automatically.

LiteLLM runs a small local proxy that presents a single OpenAI-compatible endpoint. You give

it a primary model and one or more fallbacks; when the primary fails with a retryable error—a rate-limit response (HTTP 429), or a connection error when you are offline—it retries the request on the next model. Pointing your agent at the proxy instead of directly at a provider makes the cloud-to-local switch automatic and invisible to the tool.

Install the proxy:

```
python3 -m pip install --user "litellm[proxy]"
```

Create a config file (for example, `~/.litellm/config.yaml`) with a cloud primary and a local fallback:

```
model_list:
  # Cloud primary --- replace with your provider and a current model id
  - model_name: coder
    litellm_params:
      model: anthropic/YOUR-MODEL-ID
      api_key: os.environ/ANTHROPIC_API_KEY
  # Local fallback, served by Ollama
  - model_name: coder-local
    litellm_params:
      model: ollama_chat/qwen2.5-coder:7b
      api_base: http://localhost:11434

litellm_settings:
  num_retries: 2
  fallbacks:
    - coder: ["coder-local"]
```

Run the proxy, which listens on `http://localhost:4000`:

```
litellm --config ~/.litellm/config.yaml --port 4000
```

Then point your agent at the proxy. Because the proxy speaks the OpenAI API, most tools accept it as a custom endpoint. For `aider`:

```
export OPENAI_API_BASE=http://localhost:4000/v1
export OPENAI_API_KEY=placeholder   # the proxy needs no key unless you set one
aider --model openai/coder
```

Requests now go to the cloud model when it is reachable and fall back to the local model on a rate limit or when you are offline.

> **A cloud API key is separate from a chat subscription**
>
> The cloud side needs an API key billed per token, issued from the provider's developer console. A chat subscription such as Claude Pro or a Copilot seat is not an API key and cannot be used here. Omit the cloud entry entirely to run local-only, or add the key later to enable the hybrid.

> **Keep the proxy on loopback**
>
> By default the proxy binds to localhost, which is what you want. Do not expose it on `0.0.0.0` on a shared or untrusted network without authentication, because anyone who can reach the port can spend your cloud API key.

Working in HPC / Cluster Environments

Many HPC clusters do not have outbound internet access on compute nodes but do allow access on login nodes.

> **Install the Ollama binary on the cluster first**
>
> Ollama must be installed on the cluster (on the host that will run `ollama serve`) before any of the steps below. On most HPC systems you do not have root access, so the `curl | sh` installer may fail or install to the wrong place. Instead, check whether your cluster already provides it (e.g., `module load ollama`), ask your HPC administrators, or download a static binary from the Ollama releases page and place it on your `PATH`.

A useful pattern:

1. **Pre-pull models** on a login node or a machine with internet access, then copy the model files to the cluster:

```
# On a machine with internet access
ollama pull qwen2.5-coder:7b
# Ollama stores models in ~/.ollama/models by default
rsync -a ~/.ollama/models/ user@cluster.example.edu:~/.ollama/models/
```

> **Watch your home-directory quota**
>
> Model files are large—`qwen2.5-coder:7b` is ~4 GB and `qwen2.5-coder:32b` is ~20 GB—and most HPC home directories have tight quotas (often 10–50 GB). Filling your home directory can break other jobs. Redirect model storage to a scratch or project filesystem with `OLLAMA_MODELS` and rsync to that path instead:

```
# On your local machine: copy the models to the cluster scratch filesystem
rsync -a ~/.ollama/models/ user@cluster.example.edu:/scratch/$USER/ollama-models/
```

```
# On the cluster: point ollama at that path (export before `ollama serve`)
export OLLAMA_MODELS=/scratch/$USER/ollama-models
```

Set the same `OLLAMA_MODELS` value before running `ollama serve` so the server finds the models.

2. **Start Ollama on a compute node** (or an interactive session) using the pre-downloaded model files—no internet required. Set `OLLAMA_HOST=0.0.0.0` so the SSH tunnel from the login node can reach the port:

```
OLLAMA_HOST=0.0.0.0:11434 ollama serve
# If you redirected model storage (see quota warning above):
# OLLAMA_HOST=0.0.0.0:11434 OLLAMA_MODELS=/scratch/$USER/ollama-models ollama serve
```

   If you are on a shared compute node, be aware that binding to `0.0.0.0` exposes the Ollama port to other users on that host. Scheduler policies vary by site and job type, so confirm whether your job has exclusive node access (request it explicitly when in doubt—e.g., `--exclusive` in SLURM), or bind only to loopback (`OLLAMA_HOST=127.0.0.1:11434`) and tunnel from the login node when the node is shared.

3. **Forward the port** to your local machine to use your editor's Ollama integration. Because Ollama is running on a compute node (e.g., `gpu-node-01`), forward through the login node to that specific host:

```
# Replace gpu-node-01 with your actual compute node hostname
ssh -L 11434:gpu-node-01:11434 user@cluster.example.edu
```

   This terminal must stay open for as long as you use the editor's Ollama integration—closing it tears down the tunnel and silently drops the connection. Alternatively, start the tunnel in the background (non-interactive) so it does not occupy a terminal:

```
ssh -N -f -L 11434:gpu-node-01:11434 user@cluster.example.edu
```

   (`-N` runs no remote command, `-f` backgrounds ssh after authenticating.) To stop the tunnel later, match the full SSH command rather than a bare port string—`pkill -f "ssh.*-N.*11434:gpu-node-01"`—so you don't accidentally kill unrelated processes whose command line happens to contain that port. Safer still, note the PID when you start it (`pgrep -f "11434:gpu-node-01"`) and `kill` that PID directly.

   Then configure your editor to use `http://localhost:11434/v1` as the base URL.

SLURM and GPU Allocation

If running Ollama on a SLURM-managed cluster, request a GPU node with enough VRAM for your chosen model and load any required CUDA modules before starting `ollama serve`. See the UCD-SERG Lab Manual's SLURM chapter for guidance on requesting GPU resources.

Privacy Considerations

Running a model locally ensures that your code and prompts never leave your machine or cluster. This is important when working with:

- Protected health information (PHI) or other HIPAA-regulated data
- Unpublished research data under data-use agreements (DUAs)
- Proprietary or commercially sensitive code

Even with local models, avoid including raw sensitive data in prompts. Work with anonymized or synthetic data wherever possible.

"Local" is not automatic—some model tags route to a cloud

Running Ollama does not by itself guarantee that a prompt stays on your machine. Ollama can serve **cloud-hosted** models alongside local ones, and those are the models too large to run on a laptop at all, which is exactly when a tag is tempting. A cloud-routed tag looks much like a local one in everyday use.
Two habits keep this honest, and they matter most in precisely the settings that motivated running locally:

- **Pull and reference explicitly local tags**, and treat a tag with no listed download size as cloud-routed until you check its own page in the Ollama model library.
- **Disable the cloud path outright** when the data is regulated, so the guarantee does not depend on remembering which tag is which:

```
OLLAMA_NO_CLOUD=1 ollama serve
```

Verify rather than trust either one. Cut the machine off the network, or block outbound traffic, and confirm the agent still completes a real task — the check described under Verifying you are genuinely offline below. A setup that quietly depended on a cloud endpoint fails that test immediately.

Verifying you are genuinely offline

A local setup that has never been tested without a network is a local setup you are guessing about. Cutting the machine off entirely is the honest test. A lighter one that does not disturb the rest of your session is to make outbound traffic fail for a single command, while leaving `localhost` reachable:

```
export HTTPS_PROXY=http://127.0.0.1:9 HTTP_PROXY=http://127.0.0.1:9
export NO_PROXY=localhost,127.0.0.1

# Confirm the block is real before trusting the result:
curl -s -m 5 -o /dev/null -w '%{http_code}\n' https://example.com  # 000 = blocked
curl -s -m 5 -o /dev/null -w '%{http_code}\n' http://localhost:11434/api/version  # 200 = lo

aider --yes --message "Fix the off-by-one error in mean()." stats.py
```

Check the block itself first, as above. A test that passes because the proxy was never applied tells you nothing, and looks exactly like success.

2 Assessing Aeris and Graft

Issue #46 asked whether two GitHub projects, Aeris and Graft, have a place in the lab's AI workflow. The notes below reflect each repository's README and metadata as read on 2026-09-09.

Aeris

Cedrick-Coto/Aeris (Coto 2026) describes itself as a deterministic cognitive simulation engine with emergent narrative, whose README describes an entity-component-system (ECS) architecture in C# (.NET 10). Its design premise is that the language model "verbalizes, never thinks": a deterministic simulation core computes perception, attention, memory, affect, and goals each tick, and the LLM only turns that state into narrative or dialogue, never modifying it. The target application is a simulated character in a Pokemon world.

Repository facts (measured 2026-09-09):

- GPL-3.0 license
- 2 stars, 1 fork, 2 contributors, 80 commits since creation on 2026-07-26
- 2 commits in the month before the reading date
- 87 Markdown files and three `.csproj` project files on `main`, but no `.cs` source files, so the "complete" engine and "210 tests" that the README roadmap reports are not present in the public repository

The project is a research design document for LLM-backed simulated agents, not a tool for working with AI coding assistants. Nothing in it targets writing, reviewing, or running code.

Useful to us? No

Aeris does not overlap with any lab workflow. Its one transferable idea, keeping the LLM out of state changes and confining it to presentation, is already how this site recommends treating agents: the responsibility-for-validation policy and the review guidance put the deterministic checks and the accountability with the human and the CI pipeline, not the model. The GPL-3.0 license would also complicate reuse of any code in lab packages, though at the reading date there was no code to reuse. No further action is warranted.

Graft

`trailhq/Graft` (NanoNets and Trail contributors 2026) (formerly `NanoNets/Graft`; the old URL redirected at the reading date) is an open-source "context layer" for coding agents. Its aim is to stop an agent re-exploring a repository at the start of every task by building a persistent map of the codebase once and feeding the relevant parts of that map into each prompt.

Repository facts (measured 2026-09-09):

- MIT license
- TypeScript, published on npm as `@nanonets/graft` (0.16.0 on npm; `package.json` on `main` reads 0.17.0), requiring Node 20 or later
- 6,788 stars, 613 forks, 34 contributors
- 174 commits in the 30 days before the reading date, and 43 open issues (excluding pull requests)

How it works, per the README:

- `graft build` parses the repository with tree-sitter into a per-symbol code graph (functions, classes, call and import edges). This layer is deterministic, runs locally, and never calls a model, so it needs no API key.
- `graft build --deep` adds an LLM layer: a short summary per file, grouped into a few dozen Markdown "nodes" (one per subsystem or concept) with typed links between them and a "crux" excerpt of the lines that carry the logic. The model runs under your own key through one of:
  - OpenAI
  - Anthropic
  - OpenRouter
  - a LiteLLM proxy
  - a local server

- `graft init` wires the graph into an agent. For Claude Code it writes a skill file, hooks, and a status line. For the other hosts it adds a fenced section to `AGENTS.md`, `GEMINI.md`, or `.github/copilot-instructions.md`:
  - Codex
  - OpenCode
  - Gemini CLI
  - Copilot
  - Cursor
  - others that read those files

  It also registers an MCP server exposing six tools (find code, file API, trace callers, regex search, repo map, freshness check).
- The graph is a git-ignored local cache, rebuilt incrementally and refreshed before each query; only the small wiring is committed.
- R is one of the "full-fidelity" languages, with support for plain functions, S3, S4, and R6 classes and methods, roxygen `@export` tags, and `library()` and `source()` imports.

The README reports benchmarks from the project's own harness (42% fewer tokens and 60% less latency with equal correctness on 162 runs across two repositories) and a 50-instance SWE-bench Verified run, Claude Code with Claude Sonnet 5 on both arms (33 of 50 resolved with Graft against 27 without, using 23% fewer tokens). These are vendor-run measurements on the vendor's chosen tasks, so treat them as an upper bound until reproduced. The package sends an anonymous batched usage ping by default; the README states the ping carries no code, paths, or queries, and that `graft telemetry disable` or `DO_NOT_TRACK=1` turns it off.

Useful to us? Yes, trial it on one R package

Graft addresses a cost the lab pays constantly: each new agent session re-reads an R package to rebuild the same picture of its structure. Three things make it a good fit for a trial:

- The structural layer is free and local, so it can run on a package without spending model tokens or sending code anywhere, which matches the constraints in Section 1.
- R is a first-class language in its parser, including S3, S4, and R6 dispatch and roxygen tags.
- It wires into the agents the lab already uses (Claude Code, Codex, OpenCode, Gemini CLI, and Copilot) through their existing instruction files and an MCP server, the mechanism MCP server setup covers, so a trial adds no new harness.

Check two things before adopting it lab-wide. First, the benchmark claims are vendor-run, so measure the token and tool-call counts on one of our own packages before and after `graft init`. Second, the `--deep` layer spends model tokens on every changed file and the `init` step for

Codex writes to user-level config outside the repository (`~/.codex/`; skip with `--no-global`), so start with the structural layer only and the Claude Code wiring only. The `graft init --dry-run` flag lists every file it would touch, which is the right first command. Graft is complementary to the memory tooling in Magic Context: that section covers remembering what happened across sessions, whereas Graft covers what the code is, and the two do not overlap.

3 Connecting OpenCode to Local Models

OpenCode is an open-source coding agent that runs in your terminal, reads your project, edits files, and runs commands. It supports local models through OpenAI-compatible providers.

This section assumes Ollama is already installed and that you have pulled a code-focused model — see Section 1 for both, including the Linux and Windows install paths.

Verify the server is running:

```
curl -s http://localhost:11434/api/version
# {"version":"0.1.x"}
```

On macOS, `brew services start ollama` registers a launchd agent so the server comes back automatically at login rather than needing a manual start each session.

Install the model-discovery plugin:

Rather than hand-coding each model into `opencode.json`, use the `opencode-local-ollama` plugin, which discovers your local Ollama models automatically on startup:

```
opencode plugin --global opencode-local-ollama
```

This writes to `~/.config/opencode/opencode.json`:

```
{
  "plugin": ["opencode-local-ollama"]
}
```

Restart OpenCode and run `/models` to see your local models listed alongside any cloud providers. The plugin reads from Ollama's `/api/tags` and `/api/show` endpoints, so newly pulled models appear on the next restart with no config edits.

> **Alternative: multi-backend local provider**
>
> If you also run LM Studio, `llama.cpp`, or vLLM alongside Ollama, the `opencode-local-provider` plugin auto-detects all of them under a single `local` provider and probes each at runtime for loaded models. Install it with `opencode plugin --global opencode-local-provider`.

A lightweight hand-written provider block in your project's `opencode.json` still works if you prefer explicit control over model names and context limits, but the plugin removes the need to keep that list in sync with `ollama pull`.

4 Connecting OpenCode to OpenRouter

OpenRouter is a gateway that exposes hundreds of hosted models — Claude, GPT, Gemini, DeepSeek, Qwen, Kimi, Llama, and more — behind a single API key and billing account. OpenCode treats it as a built-in provider, so its catalog appears in the `/models` picker alongside local models (Section 3). The catalog changes frequently; model IDs below were verified against it in August 2026.

Connect an API key:

1. Create a key at https://openrouter.ai/settings/keys and add credits at https://openrouter.ai/credits. Some models carry a `:free` ID suffix and cost nothing, at the price of tight rate limits.
2. In the OpenCode TUI, run `/connect`, select **OpenRouter**, and paste the key. The CLI command `opencode auth login` does the same thing outside the TUI. Either way the key is stored in `~/.local/share/opencode/auth.json`, never in `opencode.json`.
3. Run `/models`, filter for `openrouter`, and pick a model.

Pick a model that supports tool calls:

Coding agents drive every action — reading files, editing, running commands — through tool calls. Image-generation, speech, and embedding models have no endpoints that support tool use, so an agent session fails on them immediately with `No endpoints found that support tool use`. Prefer chat or coder variants such as `anthropic/claude-sonnet-4.5`, `deepseek/deepseek-chat`, or `qwen/qwen3-coder`.

One trap worth naming: on OpenRouter, Google lists Gemini 3 Pro only as image-output variants (`google/gemini-3-pro-image`, `google/gemini-3-pro-image-preview`) — there is no plain `google/gemini-3-pro` entry — so those image models are easy to pick by mistake. For tool-calling work, use one of the Gemini Flash chat variants instead, such as `google/gemini-3-flash-preview`.

Models are addressed as `openrouter/<vendor>/<model>`, for example `openrouter/deepseek/deepseek-chat`.

Optional configuration in `~/.config/opencode/opencode.json` (or the project-level file):

```
{
  "$schema": "https://opencode.ai/config.json",
  "model": "openrouter/deepseek/deepseek-chat",
  "small_model": "openrouter/openai/gpt-oss-20b",
  "provider": {
    "openrouter": {
      "models": {
        "moonshotai/kimi-k2": {
          "options": {
            "provider": { "order": ["baseten"], "allow_fallbacks": false }
          }
        }
      }
    }
  }
}
```

- `model` pins the session default; without it, OpenCode starts each session on its own built-in default
- `small_model` sends housekeeping tasks (session titles, summaries) to a cheap model instead of a frontier one
- entries under `provider.openrouter.models` add models that are not preloaded, or pin routing: OpenRouter load-balances across upstream hosts by default, and `order` restricts requests to named providers (provider-selection docs)

Config loads at startup, so restart OpenCode after editing it.

5 Running Claude Code with Non-Anthropic Models

Claude Code is built around the Anthropic Messages API, and its documentation describes how to point the harness at any endpoint that speaks that format (measured 2026-09-01). That mechanism is what makes it possible to run Claude Code against models Anthropic does not make, and the same documentation says plainly that doing so is unsupported. This section summarizes:

- the mechanism

- the routes people use
- what breaks
- which billing rules apply

> ⚠ A fast-moving snapshot
>
> The configuration claims in this section were checked against the Claude Code documentation on that date. The community practice and the policy history come from the research summary in issue #96, compiled from press coverage and forum discussion up to August 2026; Anthropic revised the subscription rules more than once during 2026. Re-read the current gateway documentation and the consumer terms before relying on any of it.

The official position: documented mechanism, unsupported use

Anthropic's gateway documentation is written for organizations that run their own gateway in front of **Claude** models, so that credentials, usage tracking, cost controls, and audit logging live in one place. It states that "any gateway that exposes a supported API format works," and in the same paragraph that Anthropic "doesn't endorse, maintain, or audit third-party gateway products, and doesn't support routing Claude Code to non-Claude models through any gateway."

The supported non-default backends are all Claude models on other clouds:

- Amazon Bedrock (`CLAUDE_CODE_USE_BEDROCK=1`)
- Google Cloud's Agent Platform, formerly Vertex AI (`CLAUDE_CODE_USE_VERTEX=1`)
- Microsoft Foundry (`CLAUDE_CODE_USE_FOUNDRY=1`)

Routing to a different model family is the unsupported case. It works because the gateway mechanism is provider-agnostic, not because Anthropic tests it.

The configuration surface

The variables that matter, from the connection guide, the model configuration page, and the environment-variable reference:

- `ANTHROPIC_BASE_URL` points the harness at a gateway. The documentation is explicit that it "changes where requests are sent, not which model answers them."
- `ANTHROPIC_AUTH_TOKEN` (sent as `Authorization: Bearer`) or `ANTHROPIC_API_KEY` (sent as `x-api-key`) carries the credential. `ANTHROPIC_AUTH_TOKEN` takes precedence over a saved claude.ai login immediately; `ANTHROPIC_API_KEY` takes over after a one-time approval prompt in interactive mode.

- `ANTHROPIC_MODEL`, and the `ANTHROPIC_DEFAULT_SONNET_MODEL`, `ANTHROPIC_DEFAULT_OPUS_MODEL`, `ANTHROPIC_DEFAULT_HAIKU_MODEL`, and `ANTHROPIC_DEFAULT_FABLE_MODEL` family, map the built-in aliases to whatever model IDs the backend accepts.
- `CLAUDE_CODE_ENABLE_GATEWAY_MODEL_DISCOVERY=1` makes Claude Code query the gateway's `/v1/models` endpoint at startup and add the results to the `/model` picker.
- `ANTHROPIC_CUSTOM_MODEL_OPTION` (with optional `_NAME` and `_DESCRIPTION`) adds a single custom row to the picker; Claude Code skips validation for that ID, so any string the endpoint accepts works.
- `CLAUDE_CODE_DISABLE_EXPERIMENTAL_BETAS=1` stops Claude Code from sending pre-release request fields and beta headers. The connection guide lists it as the fix for `400` errors naming `context_management` or `Extra inputs are not permitted`, which is what a non-Anthropic upstream returns when it rejects Claude-specific fields.
- `CLAUDE_CODE_MAX_CONTEXT_TOKENS` declares the context window for a model ID Claude Code does not recognize; Section 1 shows it in use for a local model.

Claude Code reads these at startup, either from the shell or from the `env` block of `~/.claude/settings.json`.

The routes people use

The forum reports collected in the tracking issue fall into a few shapes, from fewest moving parts to most:

- **A model vendor's Anthropic-compatible endpoint.** Several labs, including Z.ai (GLM), Moonshot (Kimi), and MiniMax, publish endpoints that speak the Messages format directly, so the whole setup is `ANTHROPIC_BASE_URL` plus the vendor's key. It is the route those reports describe most often, and the one bundled with the vendors' "coding plan" subscriptions.
- **A hosted aggregator.** OpenRouter exposes an Anthropic-format endpoint, so Claude Code can talk to it with no local proxy; Section 4 covers the same aggregator from the OpenCode side.
- **A self-hosted gateway.** LiteLLM and Kong are the examples that Anthropic's own environment-variable reference names for gateway model discovery. LiteLLM translates Messages-format requests to many providers and adds virtual keys, fallbacks, and cost tracking; Section 1 shows a LiteLLM configuration for local models.
- **A community router.** Claude Code Router intercepts each request and routes it by type (background, thinking, long-context, default) to a different provider or model. It has the most features of the five, and the tracking issue quotes its own issue tracker describing it as unstable and hard to configure.
- **A local model server.** vLLM and Ollama both serve the Messages API directly, so Claude Code can point at either with no proxy; Section 1 covers the mechanics, and Section 6 covers which local models can sustain an autonomous loop.

Treat any self-hosted gateway or community router as infrastructure: pin its version, and keep the official `claude` launcher as a fallback. The tracking issue records a 2026 incident in which malicious LiteLLM releases were briefly published to PyPI, which is the concrete reason for pinning.

What degrades

The forum reports in the tracking issue agree on three seams:

- **The harness is tuned for Claude.** Tool-call conventions, context compaction, and the prompt structure behind plan mode and subagents were built against Claude's behavior. Other models, reached through a translating gateway, show worse tool-call reliability (a model describing an edit instead of making it is the failure reported most often), occasional compaction failures, and behavior that is hard to debug because two layers are involved.
- **Prompt caching depends on the gateway.** Claude Code re-sends the system prompt, tool schemas, and history on every turn and relies on prompt caching to make that cheap. The gateway protocol reference lists caching among the features a gateway has to pass through; a gateway that does not, or an upstream with no equivalent, bills every turn at full input price, which erodes much of the per-token saving from a cheaper model.
- **Claude-specific request fields are rejected.** The `400` errors and the `CLAUDE_CODE_DISABLE_EXPERIMENTAL_BETAS` remedy in the configuration list above are the documented form of this, and the model configuration page adds that the context window Claude Code assumes for an unrecognized model ID may not match the real one.

The summary also notes that benchmark gaps between the strongest open-weight coding models and Claude narrowed during 2026, while cautioning that the open-weight figures it saw were largely the vendors' own launch numbers. Harness-level reliability is a separate question from benchmark score in any case.

Billing and policy

Two rules from the gateway documentation settle most questions:

- While a gateway credential variable or `apiKeyHelper` is active, "a developer's claude.ai subscription isn't used … and the subscription's usage limits don't apply." That traffic is billed per token to whoever owns the credential the gateway forwards.
- Setting `ANTHROPIC_BASE_URL` alone, without a gateway credential, does not replace the subscription; the saved login stays the active credential and its limits apply. The documented case for that is an organization's own gateway forwarding to Anthropic, not a third-party harness serving other models.

The policy history in the tracking issue concerns the other direction, using a Claude **subscription** login to drive third-party harnesses or proxies. As that summary records it, Anthropic announced in April 2026 that subscriptions would stop covering usage in third-party tools from 2026-04-04, with such usage drawing on separately purchased extra usage or an API key instead, while provisioning your own API keys or provider credentials, billed to the key owner, stayed allowed. The summary also notes that the terms were reworded afterwards, so read it as a pointer to what to check rather than as the current text.

Recommendations

Table 1: Choosing a route for running Claude Code with other models

Goal	Suggested route
Model flexibility is the priority	A model-agnostic harness (OpenCode, Aider, Cline), which is built for it; see Section 3 and Section 4
Claude Code's skills and plugins with one cheaper model	The vendor's Anthropic-compatible endpoint with your own key
Several models with team governance	A self-hosted gateway such as LiteLLM, or an enterprise gateway product, pinned as infrastructure
Per-request-type cost routing	Claude Code Router, pinned as infrastructure
Claude models at subscription prices	The official Claude Code CLI signed in with the subscription, which remains fully supported

Whichever route you pick:

- Authenticate non-Claude traffic with your own API key or provider credential rather than a subscription login.
- Assume prompt caching is off until the gateway proves otherwise.
- Measure cost per completed task on your own work rather than comparing token prices; if the alternative is not cheaper on that measure, or tool-call failures disrupt sessions, go back to a supported configuration.

6 Small, Local Models for Autonomous Agentic Coding

Section 1 covers the mechanics of running a model on your own hardware: installing Ollama, wiring up an editor, and driving `aider` against a local endpoint. This section is about a narrower and harder question sitting on top of that setup: which local model to pick, and how

to let it work **autonomously** — making a sequence of edits, commits, and tool calls with no human approving each step — without the loop quietly going wrong.

> ⚠ A fast-moving, opinionated snapshot
>
> As of August 2026, the open-weight coding-model landscape changes monthly: new releases, new quantizations, and new benchmark numbers appear faster than any static page can track. The model names, sizes, and figures below were verified against each model's own listing at the time this section was written, not against benchmark round-ups, and they will drift. Re-check the source links before choosing a model for a new project, and re-benchmark on your own tasks rather than trusting a published score — your repository's mix of languages and idioms is not the benchmark's.

The honest catch

Small models make more per-step mistakes than frontier cloud models: a slightly wrong function signature, a hallucinated package, a test edited to pass instead of a bug fixed. A human working alongside a small model catches most of these immediately. An **autonomous** loop does not have that human in it, so a small error on step 3 becomes the premise for steps 4 through 40, and the mistakes compound rather than cancel out.

This is the reason **small + local + fully autonomous** is the hardest combination to run safely, and the reason this section spends most of its length on structure rather than on model selection. The fix is not a bigger local model — that only raises the error rate at which the same compounding problem starts to bite. The fix is bounding the loop so that a compounding error is caught and stopped early, covered under Guardrails for autonomy below.

Model landscape

Prefer a model explicitly trained for **tool calling and agentic use** over a general chat or plain code-completion model: an agentic loop depends on the model reliably emitting well-formed tool calls and stopping when it has finished a step, not only on writing plausible code. A handful of open-weight families currently fit that description well enough to run an autonomous loop against:

Family	Sizes worth running locally	License	Best for
Llama 3.2	3B (2.0 GB), 1B (1.3 GB) dense	Llama 3.2 Community License	High-speed local subagents (< 1s action turnaround); native structured tool calling

Family	Sizes worth running locally	License	Best for
Qwen3-Coder	30B-A3B mixture-of-experts (smallest tag, 19 GB)	Apache 2.0	General-purpose agentic coding across languages
Qwen2.5-Coder	1.5B, 3B, 7B, 14B, 32B dense	Apache 2.0	Fast code generation (3B at $\sim$ 45 tok/s). Verify tool-call formatting in your harness
Phi-4-mini	3.8B (2.4 GB) dense	MIT	Strong multi-step reasoning in a compact footprint
Granite 4	7B-A1B mixture-of-experts (4.2 GB), 32B-A9B	Apache 2.0	A small, fast tool-caller that fits where the tiers above do not
Granite 3.2	2B (1.5 GB) dense	Apache 2.0	Ultra-lightweight IBM tool-calling model for single-task triage
Devstral Small	24B	Apache 2.0	Purpose-built for coding agents (multi-file edits, tool use)
Codestral	22B	Mistral AI Non-Production License	Fill-in-the-middle completion, not redistribution in a product
DeepSeek-Coder-V2	16B (Lite) mixture-of-experts	DeepSeek Model License (commercial use permitted, own terms)	A capable, low-VRAM mixture-of-experts option
GLM-4.x	Flagship models are large MoE (over 100B total parameters)	MIT	Strong agentic benchmarks, but sized for a workstation or rented GPU, not a laptop

Qwen3-Coder's 30B-A3B tag is a mixture-of-experts model: 30B total parameters, but only about 3.3B active per token. VRAM at rest is set by the total, not the active count — every

expert has to stay resident in memory even though only a fraction fires on any given token — which is why the tag still needs roughly 19 GB at 4-bit quantization, in line with its 30B total rather than its 3.3B active count. What the small active count buys is speed: inference runs closer to a 3–4B model's pace despite the larger memory footprint. Codestral's license is worth reading before you rely on it: Mistral's Non-Production License permits local evaluation but not production or commercial deployment — fine for trying it out, not fine for a lab pipeline that runs unattended.

As a practical floor, treat the 24–32B tier at 4-bit quantization as the smallest size that holds up across a multi-step autonomous loop without frequent tool-call errors. Below that, a model is still useful as an assistant you supervise turn by turn (Section 1 covers exactly that setup), but it is not yet a safe choice to leave unattended.

! Parameter count is the wrong first question

Treat that floor as a statement about *sustained* multi-step loops, not as a filter to apply before anything else. Size predicts tool-calling ability poorly enough that checking it first will mislead you.

Measured on a 24 GB M2 against a single-function tool schema, `qwen2.5-coder:14b` returned an empty `tool_calls` field on four attempts out of four, with `finish_reason: stop` each time. It wrote a correct tool call as prose in the `content` field instead, which no harness will act on. The 4.2 GB `granite4:7b-a1b-h`, at half the parameter count, put a well-formed call in `tool_calls` three times out of three and completed a multi-turn round trip using the result. The smaller model was usable as an agent where the larger one was not, and no amount of context or prompting fixes a model whose calls never reach the field a harness reads.

The advertised capability list does not settle it either. `ollama show qwen2.5-coder:14b` lists `tools`, and the model still cannot be driven by a harness, so treat the tag as necessary rather than sufficient.

So order the questions this way:

1. **Does it emit well-formed tool calls?** One request answers this. A model that fails here cannot be an agent at any size.
2. **Does it fit, with context?** Weights plus key-value cache, verified with `ollama ps`.
3. **Is it big enough to sustain a long loop?** This is where the 24–32B floor applies.

A small model that clears the first two is worth measuring on your own tasks before concluding it cannot be left unattended, because the guardrails below, not the parameter count, are what actually bound the damage from a bad step.

Hardware tiers

VRAM (or unified memory)	Model tier	Autonomy
~8 GB	7–8B	Assistant only — keep a human reviewing every step
~12–16 GB	14–24B	Entry point for a bounded autonomous loop
~24 GB+	30–32B, with context headroom	Comfortable autonomy at the practical floor above
Apple-silicon unified memory (32 GB+)	Same tiers as above, generally slower per token	Well suited to an overnight batch job where wall-clock time matters less

These are rough guides, not guarantees: VRAM headroom for context length matters as much as VRAM for the weights themselves, and a long-running agentic loop accumulates a long conversation history that eats into that headroom as it runs. Check the current requirements on the model's own listing (the Ollama model library states them per tag) rather than a rule of thumb, since quantization schemes change.

⚠ Apple unified memory does not map onto the VRAM column

Read the unified-memory row as its own scale rather than as the VRAM figures with a speed penalty attached. Two deductions come off the headline number before any model loads:

- **Only about 75% of unified memory is addressable by the GPU** by default, so a 24 GB machine has roughly 18 GB to work with, not 24.
- **Context is charged on top of the weights.** Measured on a 24 GB M2, raising a 14B model from 4k to 32k context moved it from 9.5 GB resident to 15 GB.

Together those rule out the 30–32B tier on a 24 GB Mac, even though the headline number matches the VRAM column. The smallest `qwen3-coder` tag is 19 GB, which exceeds the addressable ceiling on its own, leaving nothing for context. There is no smaller variant of it to fall back to.
The practical ceiling on 24 GB of unified memory is a **12–14B dense model at 32k context**, or a mixture-of-experts model of similar footprint. Confirm with `ollama ps` after loading: `PROCESSOR` reading `100% GPU` means it fits, and anything less means part of the model is on the CPU and the loop will be far slower than the tier table suggests.

Action latency and memory bandwidth

When designing an interactive agent or subagent workflow, **turn turnaround time** dictates whether the tool feels responsive or unusable. In an autonomous or semi-autonomous loop,

latency per action is governed by two phases:

$$\text{Action Latency} = \text{Time to First Token (Prompt Ingestion)} + \text{Tool-Call Generation (Decode)}$$

On consumer unified memory hardware (such as Apple M-series chips with ~ 100 GB/s memory bandwidth), the memory bus sets a hard theoretical ceiling on token generation speeds:

- **14B Dense Models (4-bit, ~ 9 GB weights)**: Decode is physically capped at $\sim$ 10–11 tokens/s. Generating a modest 60-token tool call takes 6 seconds on decode alone. Combined with multi-turn prompt evaluation (3–8 seconds on long conversation histories), the total turn latency exceeds 10–16 + **seconds per action**, which is too sluggish for tight iterative tool loops.
- **7B–8B Models (4-bit, ~ 4.5 GB weights)**: Decode reaches $\sim$ 20–25 tokens/s, producing action turnarounds in the 4–7 **second** range.
- **1.5B–4B Small Language Models (4-bit, $\sim$ 1–2.5 GB weights)**: Decode runs at 40–90 + **tokens/s**, and prompt ingestion finishes in < 500 ms with prefix cache reuse. Total action turnaround drops to 1–3 **seconds**, making fast, real-time subagent action loops practical on laptop hardware.

> **Prompt caching preserves latency across turns**
>
> In a multi-turn agent loop, the system prompt and accumulated history are resent on every iteration. Ensuring the inference server keeps the model resident in memory (`keep_alive: -1` in Ollama) and maintains prompt KV-cache reuse drops Time-To-First-Token on subsequent turns from several seconds to under 50 ms.

An interactive chat REPL is not an agent harness

A common stumbling block when running local models is typing agent instructions directly into `ollama run`:

```
ollama run llama3.2:3b
>>> grab an issue from github and write a PR to fix it
```

In plain `ollama run`, the model is running in an isolated conversational REPL with **no tool schemas, no file access, and no shell or Git access**. Because it cannot actually query GitHub or inspect your repository, it will fabricate a fictional issue (e.g. `cpython/issues/1234`), write fictional code in prose, and describe a non-existent commit. When asked *"did you push the PR?"*, it will correctly admit that it is a text-only assistant without execution capabilities.

An agent requires a **harness** (such as `aider` or a programmatic tool runner) that:

1. Translates available capabilities into structured tool schemas (`tools` parameter).
2. Intercepts the model's structured `tool_calls` payloads.
3. Executes the corresponding commands or file edits on the local machine.
4. Feeds the command outputs back into the conversation context as tool messages.

Baking deterministic agent presets with Modelfiles

Default local model tags are configured for open-ended conversation (temperature 0.8, small 4k context). For agentic tool use, bake a dedicated model tag via a `Modelfile` to enforce deterministic schema compliance:

```
# Modelfile.llama3.2-agent
FROM llama3.2:3b
PARAMETER temperature 0.0
PARAMETER num_ctx 16384
SYSTEM You are a fast, concise autonomous coding agent. Always execute tasks directly using a
```

Create the derived model:

```
ollama create llama3.2-agent -f Modelfile.llama3.2-agent
```

This ensures:

- **Zero temperature (`0.0`)**: Prevents hallucinated JSON keys or invalid tool parameters.
- **Expanded context (`16384`)**: Accommodates multi-turn tool outputs and file snippets without silent truncation.
- **Direct system persona**: Suppresses conversational preamble (*"Sure, I'd be happy to help with that…"*) in favor of immediate tool invocation.

Routed architectures: a planner and an executor

Section 1 already shows the mechanics of splitting a task between two local models with `aider --architect`: a larger model plans the change, and a smaller one applies the edits. The same split has a name in the research literature and a stronger motivating argument than "it's faster": Belcak and NVIDIA's small-language-model research group argue that most of what an agent does in a loop is "a small number of specialized tasks repetitively and with little variation" — reading a diff, running a test, formatting a commit message — and that a small model is "sufficiently powerful, inherently more suitable, and necessarily more economical" for that work (Belcak et al. 2025). A large model earns its cost only on the steps that genuinely need broad, general reasoning: deciding *what* to change and why.

Two shapes of this pattern are worth knowing:

- **All-local**: a single strong local model (30–32B) does both planning and execution, which is simplest to set up and is the right default for a laptop or workstation with one GPU.
- **Local planner, local executor**: a 30–32B planner drafts each step and a 7–8B executor applies it, trading some plan quality for throughput — worthwhile mainly on hardware that cannot comfortably hold two copies of a 32B model at once.
- **Cloud planner, local executor**: a frontier cloud model plans and a local model executes, which keeps the bulk of file contents on your own machine while still using strong reasoning for the decisions that matter most. This is a hybrid rather than a fully local setup — see the LiteLLM fallback pattern in Section 1 for one way to wire a cloud-with-local-fallback endpoint, which composes with this split.

The same split is the main cost lever in a public field report from a non-programmer ("Vibe coded this game in four months", r/ClaudeCode, 2026-08-22; summarized in issue #98), who built a browser racing game over four months with coding agents. The report names three levers:

- plan with the strongest model available
- implement with cheaper ones
- keep the scope to what can realistically ship

The reported total was roughly $200 in project-specific subscriptions over the four months, on top of a general-purpose subscription the author already held. The lab's machine-facing configuration, `Morrison-Lab/ai-config`, states the same routing rule for agents.

None of these routing choices substitutes for the guardrails below. A well-chosen planner still hands off to an executor that can make a per-step mistake, and the loop still needs a way to catch that.

Guardrails for autonomy

> **!** Errors compound in an unattended loop
>
> A frontier cloud model makes fewer per-step mistakes than a small local one, but the risk that matters here is not the per-step error rate on its own — it is that **autonomous** mode removes the human who would otherwise catch a mistake before it becomes the premise for the next ten steps. Structure the loop so that a mistake is caught by something other than a human watching in real time, or do not leave a small model fully unattended.

The mitigation for a higher per-step error rate is not a better model; it is a loop that cannot silently drift far from a known-good state. Five patterns do most of the work, and each is deliberately mechanical rather than judgment-based — the whole point is that they do not depend on the model noticing its own mistake:

1. **Verification gates after every step.** Run something that actually exercises the change — the test suite, `quarto render`, a linter, `R CMD check` — after each edit, and treat a non-zero exit as a hard stop for that step, not a suggestion. The environment's pass/fail signal is the judge, never the model's own claim that it "should work now."
2. **Capped blast radius.** Run the loop in an isolated Git worktree (see the `worktree` feature), and commit after every step that passes its gate — never after a batch of several. A commit-per-green-step history means the worst outcome of a bad step is one commit to roll back, not an unreviewable pile of changes.
3. **Bounded loops.** Cap the total number of iterations and, separately, the number of *consecutive* failed gates. Hitting either cap should stop the loop and report what it tried, not retry indefinitely — a model that fails the same gate three times in a row is not going to succeed on the fourth attempt without a different approach, and a different approach is a decision for a human to make.
4. **Decomposition into specified, testable units, done up front.** Break the work into steps that each have a checkable definition of done before the loop starts, rather than handing the model one large, open-ended goal. A step with a clear pass/fail test is exactly the shape a small model handles well; an open-ended goal is exactly the shape that invites drift.
5. **Full logging.** Keep every prompt, tool call, and gate result the loop produced, so a run that stopped (or that a human later distrusts) can be reviewed after the fact rather than re-run blind.
6. **A tool surface small enough for the model.** Every tool the harness exposes costs context before the task starts, because its schema is sent with the prompt, and a configured Model Context Protocol (MCP) server can add thousands of tokens of definitions on its own. Against a 32k local context that overhead competes directly with the work, and a small model loses: asked to read one file, a 7B model with a full MCP surface loaded produced a paragraph about missing credentials instead, and with MCP disabled invented a task list copied from the description of a tool it had been shown. Expose the smallest set of tools the step actually needs, and prefer a light harness over a heavyweight one — the same model that failed those attempts fixed a real bug and committed it in 17 seconds through `aider`.

These are the guardrails as a reader-facing rationale. The concrete gate wiring — an `ai-config` skill that launches a capped, worktree-isolated local loop, and a `gha` reusable workflow that runs one against a pull request using this repository's own lint, spellcheck, and render checks as its gates — is tracked separately; see Companion work below.

Stack-specific notes

This site's own stack — R, Python, Quarto, Julia, GitHub Actions YAML, and Markdown — is largely about producing *verifiable* artifacts: a script that runs, a document that renders, a workflow that passes. That is exactly the property that makes a narrow, checkable sub-task safe for an autonomous small-model loop, but the safety margin is not the same across languages:

Language / format	Autonomy dial	Notes
Python, Markdown, GitHub Actions YAML	Loosest leash	Well represented in training data; a syntax or lint check is a strong gate on its own
R	Tighter gates	Watch for non-tidyverse idioms and unfamiliar use of S4 or Reference (R5) classes; a model trained mostly on Python code can default to non-idiomatic R
Quarto (`.qmd`)	`quarto render` as the pass/fail judge	Have the model edit a known-good `_quarto.yml` rather than authoring one from scratch — a render failure is a strong, cheap gate
Julia	Shortest leash, strongest model, tightest test gate	The weakest training coverage of this stack's languages, so treat any unattended Julia change as higher risk by default

None of this changes the guardrails above; it changes how tightly you set them — a smaller step size, a lower consecutive-failure cap, or simply keeping a human in the loop for Julia while letting a Markdown fix run unattended.

Fine-tuning: closing the idiom gap

A local model's non-idiomatic R or Julia, noted in the stack-specific table above, is a training-data problem rather than a capability problem: the model has seen far less R and Julia than Python, not that it is incapable of writing either. Two lighter options are worth trying before fine-tuning anything:

- **Retrieval**, giving the model your own package's existing R or Julia code as context so it has concrete idiom to imitate.
- **Prompting**, stating the conventions directly — this repository's own `CLAUDE.md` and `.github/copilot-instructions.md` are examples of exactly that.

When those are not enough, **LoRA** (Low-Rank Adaptation) and its 4-bit variant **QLoRA** are the standard way to close an idiom gap without retraining a whole model. Both freeze the pretrained weights and train a small set of additional low-rank matrices on top, which cuts the trainable parameter count by orders of magnitude compared to full fine-tuning (Hu et al. 2021). QLoRA adds 4-bit quantization of the frozen weights on top of that, which is what actually

shrinks the memory footprint enough to fine-tune a mid-sized model on a single consumer GPU (Dettmers et al. 2023). Hugging Face's PEFT library is the common tooling entry point; the specifics of running it against this lab's own repositories belong in `ai-config`, not here.

Whatever you fine-tune on, keep a held-out evaluation set of real tasks from your own codebase that the training data never touched, and re-check it after every fine-tuning run — a model that has memorized its training examples will look better on paper than it performs on the next genuinely new task.

Companion work

This page explains the reasoning; it does not implement a launcher or a CI gate. Two companion issues carry the runnable parts, each linking back here for rationale:

- **`ai-config` #1292**: a skill that configures and launches a local autonomous loop — model choice, an Ollama or `llama.cpp` endpoint, and the guardrail caps above.
- **`gha` #436**: a reusable workflow, a sibling to this repository's own `claude.yml`, that runs a small/self-hosted-model agent against a pull request, wiring this site's existing checks as the loop's verification gates:
  - spellcheck
  - link check
  - non-standard-characters
  - bibliography DOIs

7 Thinking Machines Lab's Inkling Models

Inkling (Thinking Machines Lab 2026a) is a pair of open-weight, multimodal foundation models from Thinking Machines Lab (measured 2026-09-09). This is a model release, not a coding agent or a chat product: the weights are downloadable, and the company's own hosted surfaces for it are a fine-tuning API and a playground rather than an end-user assistant. This section summarizes the release and asks whether it matters for our workflow.

Who Thinking Machines Lab is

Thinking Machines Lab is a San Francisco AI startup founded in February 2025 by Mira Murati, formerly OpenAI's chief technology officer, with John Schulman, an OpenAI co-founder, as chief scientist (Wikipedia contributors 2026). It raised $2 billion at a $12 billion valuation in July 2025, and two founding members left for OpenAI in January 2026 (Wikipedia contributors 2026). Its first product, Tinker (Thinking Machines Lab 2026f), released in October 2025, is a LoRA fine-tuning API for open-weight models where the customer writes the training loop in

Python and Thinking Machines runs the GPUs (Wikipedia contributors 2026). Inkling is the company's first in-house model, released on 2026-07-15 (Thinking Machines Lab 2026c).

What Inkling is

Two models share the name (measured 2026-09-09) (Thinking Machines Lab 2026a):

- **Inkling**: a mixture-of-experts transformer with 975 billion total parameters, 41 billion of them active per token, and a context window of up to one million tokens (Thinking Machines Lab 2026c).
- **Inkling-Small**: 276 billion total parameters, 12 billion active, released 2026-07-30 and post-trained by on-policy distillation with Inkling as the teacher (Thinking Machines Lab 2026e). The product page says it matches the larger model on many benchmarks at a quarter of the size (Thinking Machines Lab 2026a).

Both accept text, image, and audio input and produce text only (Thinking Machines Lab 2026b, 2026d). Both are released under the Apache 2.0 license (Thinking Machines Lab 2026b, 2026d), and both expose an effort setting that trades reasoning tokens for score; the reported evaluations are run at `effort=0.99` (Thinking Machines Lab 2026c).

The announcement positions Inkling as a broad base for customization rather than a leaderboard winner (Thinking Machines Lab 2026c). Its claims, and the evidence offered for them:

- **Breadth**: reported scores of 77.6% on SWE-Bench Verified, 63.8% on Terminal Bench 2.1, 87.2% on GPQA Diamond, and 73.5% on MMMU Pro, all at the highest effort setting (Thinking Machines Lab 2026c).
- **Efficiency**: the company reports that Inkling matches NVIDIA's Nemotron 3 Ultra on Terminal Bench 2.1 using roughly a third of the tokens (Thinking Machines Lab 2026c).
- **Training**: 45 trillion tokens of text, image, audio, and video, then supervised fine-tuning on synthetic data and reinforcement learning with more than 30 million rollouts (Thinking Machines Lab 2026c).

These are the vendor's own numbers on the vendor's own runs; we have not reproduced any of them. Inkling-Small's announcement reports 80.2% on SWE-Bench Verified and 31.6% on Humanity's Last Exam, against Inkling's 77.6% and 29.7% (Thinking Machines Lab 2026e).

How it is accessed and priced

Access routes (measured 2026-09-09):

- **Weights**: on Hugging Face, in BF16 and `NVFP4` checkpoints (Thinking Machines Lab 2026b, 2026d). Even the smaller model has 276 billion parameters, so neither fits any hardware the lab owns.

- **Tinker**: LoRA fine-tuning at 64K or 256K context, and a beta serverless inference endpoint, both billed per million tokens (Thinking Machines Lab 2026g). Under a limited-time 50% discount, fine-tuning Inkling at 64K context costs $1.87 per million prefill tokens, $4.68 per million sampled tokens, and $5.61 per million training tokens, roughly doubling at 256K; Inkling-Small is about a third of that. Serverless inference is $1.00 input and $4.05 output per million tokens for Inkling, and $0.30 and $1.20 for Inkling-Small. Checkpoint storage adds $0.10 per GB per month (Thinking Machines Lab 2026f).
- **Third-party hosts**: the announcement lists Together AI, Fireworks, Modal, Databricks, and Baseten as hosts, and vLLM, SGLang, and `llama.cpp` among supported inference engines (Thinking Machines Lab 2026c). Our quota-aware defaults for the Databricks endpoint are already in

    1.

What it is for

Thinking Machines' stated bet is that organizations will want to fine-tune a capable open model on their own data rather than rent a closed frontier model, and Inkling exists to give Tinker a strong first-party base (Thinking Machines Lab 2026c). Tinker's own pitch lists specialized agents, forecasting, continual learning, and AI research as the intended uses (Thinking Machines Lab 2026f). The audio and image inputs make it a candidate for transcription-plus-reasoning pipelines as well as coding.

Useful to us? Marginally, and only through a host

As a coding-agent backend, Inkling is one more open-weight option alongside the models in the agent catalog, reachable through Databricks (custom model endpoints) or, when a host lists it there, OpenRouter (Section 4). Its reported SWE-Bench and Terminal Bench scores are competitive but not ahead of the closed models we already pay for, and the 8,192-token output cap on the Databricks endpoint is a real constraint for agentic edits. Self-hosting is out: the lab has no machine within an order of magnitude of the memory required.

The more interesting angle is research. Tinker lets a student run a fine-tuning or reinforcement-learning experiment on a frontier-class open model with a Python training loop and no cluster administration, and universities can ask for wider access (Thinking Machines Lab 2026f). If a project ever needs to adapt a model to epidemiological text or lab-specific coding conventions, Tinker with Inkling-Small is a cheap place to try it. Until such a project appears, no action is needed beyond keeping the Databricks endpoint in our defaults table.

References

Belcak, Peter, Greg Heinrich, Shizhe Diao, et al. 2025. *Small Language Models Are the Future of Agentic AI*. NVIDIA Research; arXiv preprint. https://arxiv.org/abs/2506.02153.

Coto, Cedrick. 2026. *Aeris: Deterministic Cognitive Simulation Engine with Emergent Narrative*. Software. https://github.com/Cedrick-Coto/Aeris.

Dettmers, Tim, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. *QLoRA: Efficient Finetuning of Quantized LLMs*. arXiv preprint. https://arxiv.org/abs/2305.14314.

Hu, Edward J., Yelong Shen, Phillip Wallis, et al. 2021. *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv preprint. https://arxiv.org/abs/2106.09685.

NanoNets and Trail contributors. 2026. *Graft: Open-Source Context Layer for Coding Agents*. Software. https://github.com/trailhq/Graft.

Thinking Machines Lab. 2026a. *Inkling*. Product page. https://thinkingmachines.ai/inkling/.

Thinking Machines Lab. 2026b. *Inkling Model Card*. Hugging Face model card. https://huggingface.co/thinkingmachines/Inkling.

Thinking Machines Lab. 2026c. *Inkling: Our Open-Weights Model*. News post. https://thinkingmachines.ai/news/introducing-inkling/.

Thinking Machines Lab. 2026d. *Inkling-Small Model Card*. Hugging Face model card. https://huggingface.co/thinkingmachines/Inkling-Small.

Thinking Machines Lab. 2026e. *Introducing Inkling-Small*. News post. https://thinkingmachines.ai/news/inkling-small/.

Thinking Machines Lab. 2026f. *Tinker*. Product page. https://thinkingmachines.ai/tinker/.

Thinking Machines Lab. 2026g. *Tinker Models and Pricing*. Documentation. https://tinker-docs.thinkingmachines.ai/tinker/models/.

Wikipedia contributors. 2026. *Thinking Machines Lab*. Wikipedia. https://en.wikipedia.org/wiki/Thinking_Machines_Lab.