Running Modern Edge LLMs on an RTX PRO 4000 Blackwell

June 10, 2026 / Gavin Jackson

ai llm local-ai ollama gguf nvidia blackwell gemma qwen hermes vscode continue

I have been spending more time than is probably healthy thinking about local AI.

Part of that is curiosity. Part of it is cost. If LLM pricing keeps moving upward, or if the useful models increasingly sit behind subscriptions, enterprise plans, and usage tiers, then relying entirely on cloud inference starts to feel fragile.

The other part is network reality. Not every environment can connect directly to the Internet. Some developer networks should not have broad outbound access. Some codebases should not leave the building. Some organisations need AI assistance inside a controlled workstation environment, not hanging off whatever API a developer happened to sign up for last Tuesday night.

So I bought an NVIDIA RTX PRO 4000 Blackwell card, installed it in an Ubuntu 26.04 workstation, and started testing what modern edge LLMs look like on hardware that is expensive, but not absurd. In Australia, I have seen the 24GB RTX PRO 4000 Blackwell listed under AUD 3,000. That is not pocket change, but it is very different from buying a datacentre GPU or building a multi-card inference box.

The key constraint is simple:

> *The model needs to fit inside 24GB of VRAM and still leave enough headroom to be usable.*

That one sentence drives almost every practical decision.

The Hardware Reality

The card I am talking about is the NVIDIA RTX PRO 4000 Blackwell, with 24GB of GDDR7 ECC memory. My test machine is running Ubuntu 26.04, which is the same base I would use for a managed developer workstation rollout. It is a workstation GPU, not a gaming flagship, and that matters. The attraction is not only raw speed. It is the combination of 24GB VRAM, a professional driver stack, sane power and cooling, and the ability to put useful local inference into a developer workstation without turning the desk into a small server room.

On 24GB, you are not running giant 70B dense models comfortably. You are not replacing frontier cloud models for every task. But you can run genuinely capable 12B, 26B, 27B, 31B, and sparse 35B-class models if you choose the right quantized files and keep context length under control.

That last bit matters. VRAM is not just model weights. You also need memory for the KV cache, runtime overhead, and whatever your inference engine is doing around the edges. A model file that is 23.8GB is technically under 24GB, but that does not mean it is a good daily-driver choice on a 24GB card.

I learned to aim for breathing room.

Picking Models That Actually Fit

There are two model families I keep coming back to for this size of card.

The first is **Gemma 4**. I have written previously about why Gemma 4's Apache 2.0 license matters, and the licensing still matters here. It is much easier to have a serious enterprise conversation about local models when the legal posture is boring and understandable.

The second is **Qwen 3.6**, especially for coding and agentic workflows. The Qwen3.6-35B-A3B model card describes a 35B total parameter model with only 3B activated per token. That is the important trick. It is a Mixture-of-Experts style model, so you get a larger pool of parameters without paying the full dense-model cost on every token.

That does not make it free. It still needs memory, and the quantization still matters. But it is exactly the kind of architecture that makes edge inference interesting.

Parameter Sizes In Plain English

Parameter count is not a perfect measure of intelligence, but it is a good first approximation of memory pressure.

- **E2B / E4B / small models**: fast, easy to run, good for classification, small summaries, simple drafting, and autocomplete-style jobs.
- **12B class**: the first size where local chat starts to feel genuinely useful. A well-quantized 12B model can be responsive and surprisingly capable.
- **26B / 27B class**: the sweet spot for a 24GB workstation card if you want better reasoning and code review without falling off a performance cliff.
- **31B dense models**: possible with aggressive quantization, but you need to be realistic about context size and speed.
- **35B sparse MoE models**: interesting because only part of the model is active per token. Qwen3.6-35B-A3B is the obvious example for coding.

The raw model is only the starting point. The version you actually run locally is usually a quantized release.

Quantization Is The Whole Game

A model published in BF16 or FP16 precision is often too large for workstation VRAM. Very roughly, 16-bit weights need about two bytes per parameter before you even think about runtime overhead. A 26B model can therefore be around 50GB in full precision. That is not going to live inside a 24GB card.

Quantization reduces the precision of the weights so the model uses less memory. The trade-off is quality. A good quant loses surprisingly little. A bad or too-aggressive quant can make the model feel vague, brittle, or weirdly forgetful.

In the GGUF world, you will see names like:

- `Q8_0`: large, high quality, often too big for this card once overhead is included.
- `Q6_K`: very good quality, but can be tight on 24GB with larger models.
- `Q5_K_M`: a nice middle ground when Q6 is too large.
- `Q4_K_M`: often the practical default for 24GB cards.
- `IQ4_NL`, `IQ4_XS`, `IQ3_*`: newer or more aggressive options that can make awkward models fit.

For example, Bartowski's Gemma 4 26B A4B GGUF page lists a full BF16 file at about 50.5GB, a `Q6_K_L` quant at about 23GB, and a `Q4_K_M` quant at about 17GB. On paper the Q6 file fits. In practice, the Q4 or Q5 variants are more comfortable because they leave room for context.

For Qwen3.6, the Unsloth Qwen3.6-35B-A3B GGUF releases include 4-bit files around the 20GB mark. That is exactly why this model is interesting on a 24GB card: it is large enough to be useful, but the right quant still fits.

The rule I have settled on is simple:

> *Pick a quant at least 1-2GB smaller than your VRAM, and preferably more if you want larger context windows.*

Step 1: Download The GGUF File

GGUF is a local model file format used heavily by `llama.cpp` and tools built around that ecosystem. Hugging Face describes it as a binary format optimised for quick loading and inference, with both tensors and model metadata stored in the file.

That metadata part matters. A GGUF file is not just a bag of numbers. It normally includes enough information for local inference tools to understand the model architecture, tokeniser, quantization type, and prompt format.

In a connected environment you can download directly:

```
python -m pip install -U "huggingface_hub[cli]"

mkdir -p ~/models/qwen3.6
cd ~/models/qwen3.6

huggingface-cli download unsloth/Qwen3.6-35B-A3B-GGUF \
 --include "Qwen3.6-35B-A3B-UD-Q4_K_M.gguf" \
 --local-dir .
```

For Gemma 4:

```
mkdir -p ~/models/gemma4
cd ~/models/gemma4

huggingface-cli download bartowski/google_gemma-4-26B-A4B-it-GGUF \
 --include "gemma-4-26B-A4B-it-Q4_K_M.gguf" \
 --local-dir .
```

In a restricted environment, I would not have each workstation randomly pulling files from Hugging Face. I would download once from a controlled machine, record the hash, put the GGUF into an internal artefact store, and use an orchestration tool such as Ansible to push the approved model files and configuration out to the endpoints.

That gives you:

- repeatability
- provenance
- hash verification
- no surprise model swaps
- no direct developer workstation Internet dependency
- a controlled way to replace or retire models across the fleet

Step 2: Install Ollama

Ollama is still the easiest way to get local LLMs running on a workstation. It wraps the model runtime, exposes a local API, and gives you a clean CLI for testing.

On a normal Linux workstation:

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

In the sort of environment I care about, that installer should be mirrored or packaged internally. The workstation should install Ollama from an approved repository, not from a random curl pipe to the Internet.

Once installed, check that it is alive:

```
ollama --version
curl http://127.0.0.1:11434
```

You should see:

```
Ollama is running
```

I also keep Ollama bound to localhost by default. A local model endpoint should not casually become a LAN service.

Step 3: Create A Modelfile And Load The Model

Ollama can import a local GGUF with a `Modelfile`. The minimal version is almost comically small:

```
FROM /opt/models/qwen3.6/Qwen3.6-35B-A3B-UD-Q4_K_M.gguf

PARAMETER num_ctx 8192
PARAMETER temperature 0.6
PARAMETER top_p 0.95
```

Save that as:

```
/opt/models/qwen3.6/Modelfile
```

Then create the Ollama model:

```
ollama create qwen36-local -f /opt/models/qwen3.6/Modelfile
```

For Gemma:

```
FROM /opt/models/gemma4/gemma-4-26B-A4B-it-Q4_K_M.gguf

PARAMETER num_ctx 8192
PARAMETER temperature 0.7
PARAMETER top_p 0.9
```

Then:

```
ollama create gemma4-local -f /opt/models/gemma4/Modelfile
```

Ollama's Modelfile reference is worth reading because the file is where you can set defaults like context length, stop tokens, temperature, and the system prompt.

Step 4: Test With Ollama Directly

Before wiring this into anything else, test the model directly.

```
ollama list
ollama run qwen36-local
```

Then give it something real:

```
You are reviewing a Python function for production readiness.
Find bugs, security issues, missing tests, and readability problems.
Be concise.
```

For a one-shot CLI test:

```
ollama run qwen36-local "Write a PHP function that safely renders a tag link for a blog post."
```

And check memory usage:

```
ollama ps
nvidia-smi
```

This is where the theory becomes visible. If the model spills out of VRAM, gets partially offloaded, or has an over-large context window, you will feel it immediately. Token generation slows down, prompts take longer to process, and the whole thing stops feeling like a useful assistant.

For coding, Qwen3.6 has felt more natural to me than Gemma. It is more comfortable with repository-shaped tasks, code review, and tool-like instructions. Gemma 4 is still a strong general local assistant, especially when licensing and broader local deployment matter.

Step 5: Get Hermes Agent Running

Once Ollama is working, the next question is: what do you actually do with it?

Chatting in a terminal is useful for testing, but it is not an agent. That is where Hermes Agent becomes interesting.

The normal Hermes install path is a script, but in a locked-down workstation environment I prefer to be explicit:

```
sudo apt install python3 python3-venv nodejs npm

python3 -m venv ~/.venvs/hermes-agent
source ~/.venvs/hermes-agent/bin/activate

python -m pip install --upgrade pip
pip install hermes-agent
```

In our environment, package dependencies come from a local PyPI mirror, so the install looks more like this:

```
pip config set global.index-url https://pypi-mirror.example.internal/simple
pip config set global.trusted-host pypi-mirror.example.internal

pip install hermes-agent
```

Then point Hermes at Ollama's OpenAI-compatible endpoint:

```
hermes model
```

Choose a custom or self-hosted endpoint, then use:

```
Base URL: http://127.0.0.1:11434/v1
API key: leave blank, or use a local placeholder if the UI insists
Model: qwen36-local
```

Hermes stores its configuration under `~/.hermes/`, which makes it fairly easy to inspect and manage. The Hermes provider docs note that custom/self-hosted endpoints work when they expose an OpenAI-compatible `/v1/chat/completions` API, which Ollama does.

Breakout: Hermes Gives Me An Agent

This is the point where the setup stops feeling like "I installed a model" and starts feeling like "I have a local assistant."

I have called mine **Frank**.

Frank is an arrogant Frenchman who is constantly criticising my code and getting upset with me for wasting his time. This sounds ridiculous, but it is genuinely useful. A local model with a memorable persona is easier to work with than a bland text box. Frank has a job: read my code, complain about the parts that deserve complaint, and begrudgingly suggest better approaches.

The important part is not the accent. The important part is continuity and role.

Hermes gives the model:

- a persistent assistant shell
- memory across sessions
- tool access
- channel support
- configuration that survives the terminal window
- the ability to behave like an agent rather than a single prompt-response loop

That distinction matters. Ollama serves the model. Hermes gives it a place to live.

I do not want developers thinking about local AI as "open a terminal and ask a model a question." I want them thinking: "I have a workstation assistant that can inspect code, remember project conventions, and operate inside the security boundary of this machine."

Frank is rude about it, obviously. But that is his burden.

Step 6: Get VS Code Working With Continue

The next piece was VS Code.

Ollama now has a native VS Code integration path through GitHub Copilot Chat. The awkward bit is that VS Code requires a login for the model selector even when you are using custom local models. Ollama's own VS Code integration documentation says this does not require a paid Copilot account, but it still requires the login path.

In a locked-down or offline-first environment, that is not what I want.

So the workaround is Continue.

Install VS Code from the internal mirror of Microsoft's repository, then install the Continue extension from your approved extension mirror or a vetted `.vsix`:

```
sudo apt install code
code --install-extension Continue.continue
```

For an offline extension package:

```
code --install-extension continue-extension.vsix
```

Then configure Continue to use Ollama. A minimal `~/.continue/config.yaml` can look like this:

```
name: Local AI Workstation
version: 0.0.1
schema: v1
models:
 - name: Qwen 3.6 Local
 provider: ollama
 model: qwen36-local
 apiBase: http://127.0.0.1:11434
 roles:
 - chat
 - edit
 - apply
 capabilities:
 - tool_use

 - name: Gemma 4 Local
 provider: ollama
 model: gemma4-local
 apiBase: http://127.0.0.1:11434
 roles:
 - chat
 - edit
 - apply

 - name: Small Local Autocomplete
 provider: ollama
 model: qwen2.5-coder:1.5b
 apiBase: http://127.0.0.1:11434
 roles:
 - autocomplete
```

The model names must match `ollama list`. If Ollama knows the model as `qwen36-local:latest`, use that exact name.

Continue's Ollama guide also supports autodetection:

```
models:
 - name: Autodetect
 provider: ollama
 model: AUTODETECT
 roles:
 - chat
 - edit
 - apply
 - autocomplete
```

That is handy for experimentation. For a managed developer fleet, I would rather be explicit.

Breakout: What VS Code Is Doing With The Local Model

The exciting thing about this setup is that VS Code does not use the model in only one way.

There are at least three different coding patterns here.

Autocomplete is the fast path. The model sees the current file and nearby context, then predicts the next few lines. This wants a smaller, faster model. It should feel instant. A huge reasoning model is usually the wrong choice.

Question and answer is the chat path. This is where a developer asks, "Why is this failing?", "What does this function do?", or "Where should I add the validation?" This can use a larger model because latency matters less than understanding.

Code synthesis and edit/apply is the workflow path. The assistant proposes changes, rewrites a function, generates tests, or applies an edit across a file. This is where Qwen3.6 starts to make sense because repository reasoning and coding fluency matter more than raw chat pleasantness.

That role separation is important. Local AI in an IDE should not be one model doing everything badly. It should be a small fast model for completion, a stronger model for chat, and a code-capable model for synthesis.

I am genuinely excited to push this out to the devs and see what they do with it. Not because I expect local models to replace every cloud tool on day one, but because the workflow changes when the assistant is local, fast enough, and available inside the editor without sending code to an external API.

Offline And Controlled Environments

The restricted-network angle is not an afterthought. It is one of the main reasons this experiment matters.

For a serious workstation rollout, I would mirror or internally host:

- Ollama packages
- approved GGUF model files
- model hashes and metadata
- Python packages through a local PyPI mirror
- Node packages through an internal npm mirror if needed
- VS Code packages from a Microsoft repository mirror
- approved VS Code extensions as `.vsix` artefacts
- Continue configuration templates
- Hermes configuration templates
- Ansible roles or playbooks that install the approved runtime, place the approved models, verify hashes, and keep endpoint configuration consistent

That gives developers a usable AI workstation without granting every machine direct Internet access.

It also gives the organisation a model governance story. You can say:

- these are the approved model files
- these are the hashes
- this is the license
- this is the intended use
- this is the runtime
- this is the endpoint binding
- this is how updates are tested
- this is the orchestration path that pushes the approved model set to endpoints

That is much better than "everyone install whatever model Reddit likes this week."

What Worked

The RTX PRO 4000 Blackwell is a very practical local inference card for the 24GB class.

The models that felt worth using were not tiny toys. Gemma 4 26B-class and Qwen3.6 35B-A3B-class quantized models are capable enough for real drafting, explanation, code review, and assistant workflows.

Ollama made the runtime easy. GGUF made the model artefacts portable. Hermes made the setup feel like an assistant. Continue made VS Code local-model integration workable without going through the Copilot login path.

The biggest lesson was that local AI is not one component. It is a stack:

```
approved GGUF file
 -> Ollama local runtime
 -> Hermes for agent workflows
 -> Continue for IDE workflows
 -> internal mirrors for repeatable offline deployment
```

When those pieces line up, the experience becomes surprisingly normal.

What Did Not Work Perfectly

The 24GB limit is real. You can make very large models load by using more aggressive quants or offloading to system RAM, but there is a point where the experience stops being pleasant.

Context length is the silent killer. A model that works beautifully at 8K context might become sluggish when a tool tries to push it to 32K or beyond. For local coding assistants, this matters because IDE tools love stuffing prompts with file context, diffs, terminal output, and instructions.

Tool support is also uneven. A model may claim tool support, but the integration may still behave strangely. This is why I like testing directly in Ollama first, then Hermes, then Continue. Each layer adds value, but each layer also adds another place to misconfigure the model.

The Bottom Line

The RTX PRO 4000 Blackwell has changed how I think about local AI workstations.

This is not about beating frontier models. It is about having a capable local baseline that works when cloud access is expensive, inappropriate, unavailable, or simply unnecessary.

For under AUD 3,000, a 24GB workstation GPU can run modern edge LLMs that are good enough to matter. Gemma 4 gives me a strong general local model with a clean licensing story. Qwen3.6 gives me a better coding-oriented model for repository work. Ollama makes them easy to serve. Hermes turns them into an agent. Continue brings them into VS Code.

That feels like the shape of the next developer workstation:

local by default, cloud when allowed, governed by design, and useful enough that developers will actually reach for it.

Frank still thinks my code is beneath him.

Unfortunately, he is often right.

References

- NVIDIA RTX PRO 4000 Blackwell
- Scorptec RTX PRO 4000 Blackwell listing
- Gemma 4 model overview
- My earlier post: Why Gemma 4's Apache 2.0 License Matters

- Qwen3.6-35B-A3B model card
- Bartowski Gemma 4 26B A4B GGUF
- Unsloth Qwen3.6-35B-A3B GGUF
- Hugging Face GGUF documentation
- Hugging Face: use Ollama with GGUF models
- Ollama importing models
- Ollama Modelfile reference
- Ollama VS Code integration
- Continue Ollama guide
- Hermes Agent documentation
- Hermes Agent AI providers
