Your code copilot, 100% local

Chapter 13 (excerpt) — « When It Breaks: Diagnose and Fix » · v2026.09 · bestllmfor.com/local-copilot-kit

Chapter 13 — When It Breaks: Diagnose and Fix

In week one of a local setup, the problem is almost never "the model is bad." It's five failures, always the same ones. They make local feel broken. In reality, you're just missing two lines of config. This chapter diagnoses them **by symptom**: you start from what you see (an error message, throughput that collapses, autocomplete that crawls), and you trace back to the cause, then to the fix. This is the reference chapter for diagnostics. Chapters 4, 6, and 12 send you back here the moment something breaks.

Working setup: you're running on `myproject` (the small Flask/Python web API that's been the running example since chapter 7), Ollama backend, hardware anchor RTX 5070 Ti 16 GB. Every time, I'll give you the "for YOUR VRAM" version. A fix that's right on 24 GB can be exactly the wrong move on 8 GB.

> ✅ **KEY TAKEAWAY — The golden rule of local debugging.** 90% of week-one failures come from **a single number**: `num_ctx`. `num_ctx` is the size of the model's working memory. You raised it to fit your repo in, and it pushed the model out of VRAM. Before suspecting anything else, check where your model is actually running (`ollama ps`) and by how much you inflated the context.

> 📄 **IN THE FULL KIT** — This chapter opens with a decision tree (symptom → section → fix) covering all 5 week-one failures. This free sample contains failure #1 (OOM) and the start of failure #2 (GPU not used).

Section A — GPU OOM: "out of memory" or collapsing throughput

The symptom

It shows up in two forms. The blunt one: a `CUDA error: out of memory` message (or the ROCm/Metal equivalent), and the model refuses to respond. The sneaky one, more common and more treacherous: **it responds, but at a few tokens per second**. The model has overflowed VRAM. Ollama has pushed part of the layers onto the CPU (that's offload), and your 16 GB GPU sits there watching the CPU crawl.

The cause

The VRAM budget is simple, and you can work it out yourself:

```
VRAM needed ≈ (quantized model weights)
            + (KV cache, which grows with num_ctx)
            + overhead
```

The KV cache is the memory where the model stores context it has already processed. On your 16 GB, the default agentic model — Devstral Small 24B in `q4_K_M` (~14 GB) — fits **as long as you don't inflate**

`num_ctx`. The classic trap: you bump `num_ctx` from 8192 to 32768 to fit more of `myproject` in. The KV cache doubles or triples, and you cross the 16 GB line with no error message — just silent offload.

The one-command diagnosis

```
ollama ps
```

Read the `PROCESSOR` column (a detailed walkthrough is in **chapter 4**). It's the central diagnostic tool for this entire chapter. Here's how to use it for OOM:

`ollama ps` shows	Interpretation	Action
`100% GPU`	Everything's in VRAM, healthy	Nothing to do on the OOM front
`58%/42% CPU/GPU` (or any non-zero %CPU)	Partial offload → throughput collapsed	Reduce `num_ctx` or model size
`100% CPU`	The GPU isn't being used at all	Go to **Section B** (this isn't an OOM)

Table 13.1 — Reading `ollama ps` to diagnose OOM.

The fix (in order, least to most destructive)

1. Lower `num_ctx` first. This is lever #1 and the cheapest one. Go back to a value that fits (8192 is the chat/agent default on 16 GB). Check with `ollama ps`, then raise it in steps until just before offload kicks in.

```
# Direct test: force a modest context and see if you're back to 100% GPU
OLLAMA_CONTEXT_LENGTH=8192 ollama run devstral:24b
```

On the Modelfile side (see **chapter 11** for the full structure), the same thing hardcoded:

```
FROM devstral:24b
PARAMETER num_ctx 8192
```

2. Drop a notch in quantization or size. If it still overflows at a reasonable `num_ctx`, your model is too big for your card. Stay at `q4_K_M` (the default quality/size trade-off), not something hungrier. Fine-tuning temp/top-p/`num_ctx` is covered in **chapter 12**; picking the right model for your VRAM, in **chapter 5**.

3. Explicitly cap GPU layers (edge case). Want to force a controlled split instead of letting automatic offload happen to you? `num_gpu` sets the number of layers sent to the GPU:

```
FROM devstral:24b
PARAMETER num_ctx 8192
PARAMETER num_gpu 99   # "as many as possible"; lower this number if you want to
deliberately offload to the CPU
```

> ⚠️ **WARNING — `num_gpu` isn't a magic formula.** Lowering it doesn't "free up" performance. Quite the opposite: it sends *more* layers to the CPU, so it slows things down. It's a tool for when you deliberately want to run a model that's too big, accepting the slowdown that comes with it. It's not an everyday setting. Under normal use, let Ollama manage it. Tune `num_ctx` and model size instead.

For YOUR VRAM:

VRAM	Main OOM reflex
8 GB	The dense 24B agentic tier is out of reach; aim for a light model (Qwen2.5-Coder 7B) and `num_ctx` ≤ 8192. Any context bump costs you dearly.
12 GB (RTX 5070, RTX 4070)	Devstral Small 24B does NOT fit here: pick Qwen2.5-Coder 14B (~9 GB) or DeepSeek-Coder-V2-Lite 16B (~10 GB) in `q4_K_M`, `num_ctx` 4096. Beyond that, watch `ollama ps`.
16 GB (ANCHOR — RTX 5070 Ti / 4080 / 5080 / RX 9070 XT, Mac M4 24 GB)	Devstral Small 24B `q4_K_M` (~14 GB) + `num_ctx` 8192 = the agentic sweet spot. Also room for Qwen2.5-Coder 14B in Q8.
24 GB (RTX 3090/4090, RX 7900 XTX, Mac M4 Pro 48 GB)	You can afford Qwen2.5-Coder 32B `q4_K_M` (~19 GB), Devstral in Q5/Q6, or long context without offload; OOM becomes rare.
Apple Silicon (Metal)	VRAM is shared with system RAM (unified memory). No hard OOM crash, but memory pressure that swaps: watch system memory pressure, close heavy processes before a big session.

Table 13.2 — OOM reflex by VRAM tier.

Section B — The GPU isn't being used at all

The symptom

`ollama ps` shows `100% CPU`, the GPU fan is asleep, and generation is slow everywhere — not collapsed like an offload, just slow *everywhere*, from the very first token. This is the most infuriating case: the hardware is right there, unused.

The cause

Ollama can't see your accelerator. Three common causes: - The GPU runtime isn't detected (CUDA / ROCm drivers missing or too old). - On Linux, Ollama is running in a context (container, systemd service) that doesn't have access to the GPU device. - On Windows/WSL, GPU passthrough isn't configured.

The diagnosis

```
# NVIDIA (your 5070 Ti) — the GPU should show up and list an ollama process during
generation
nvidia-smi

# AMD ROCm
rocm-smi

# And most importantly: read Ollama's startup logs, where it announces what it detected
```

On Linux, the service logs:

```
journalctl -u ollama --no-pager | grep -i -E "gpu|cuda|rocm|library"
```

You're looking for a line where Ollama says it found a GPU library (CUDA/ROCm). If it announces it's falling back to CPU, you've found your cause. (A clean runtime install for your OS is covered in **chapter 4** — if this section applies to you, it's often because the GPU step in chapter 4 got skipped.)

(The excerpt stops here. The rest of Section B — plus Section C “Context overflow”, Section D “The model won’t load / wrong tag”, Section E “Autocomplete (FIM) too slow”, and the symptom → cause → fix reference table — are in the kit.)