Ton copilote de code, 100 % local

Chapitre 13 — Quand ça casse : diagnostiquer et réparer

La semaine 1 d'un setup local, le problème n'est presque jamais « le modèle est nul ». Ce sont cinq pannes, toujours les mêmes. Elles font croire que le local ne marche pas. En réalité, il manque juste deux lignes de config. Ce chapitre les diagnostique **par symptôme** : tu pars de ce que tu vois (un message d'erreur, un débit qui s'effondre, une autocomplétion qui rame), et tu remontes à la cause, puis au fix. C'est le chapitre de référence du diagnostic. Les chapitres 4, 6 et 12 te renvoient ici dès qu'une panne se manifeste.

Repère de travail : tu tournes sur ta `monprojet` (la petite API web Flask/Python qui sert de fil rouge depuis le chapitre 7), backend Ollama, ancre matérielle RTX 5070 Ti 16 Go. À chaque fois je te donne la version « pour TA VRAM ». Un fix valable en 24 Go peut être exactement le mauvais réflexe en 8 Go.

> ✅ **À RETENIR — La règle d'or du débogage local.** 90 % des pannes semaine 1 viennent d'**un seul nombre** : `num_ctx`. Le `num_ctx`, c'est la taille de la mémoire de travail du modèle. Tu l'as monté pour faire rentrer ton repo, et il a poussé le modèle hors VRAM. Avant de soupçonner autre chose, regarde où ton modèle s'exécute (`ollama ps`) et de combien tu as gonflé le contexte.
> 📄 **Dans le guide complet :** ce chapitre s'ouvre sur un **arbre de décision** (symptôme → section → fix) qui couvre les 5 pannes de la semaine 1. Cet extrait gratuit contient la panne n°1 (OOM) et le début de la n°2 (GPU inutilisé).

Section A — OOM GPU : « out of memory » ou débit qui s'effondre

Le symptôme

Il prend deux formes. La franche : un message `CUDA error: out of memory` (ou l'équivalent ROCm/Metal), et le modèle refuse de répondre. La sournoise, plus fréquente et plus traître : **ça répond, mais à quelques tokens/seconde**. Le modèle a débordé de la VRAM. Ollama a basculé une partie des couches sur le CPU (c'est l'offload), et ton GPU à 16 Go regarde le CPU ramer.

La cause

Le budget VRAM est simple, et tu peux le poser toi-même :

```
VRAM nécessaire ≈ (poids du modèle quantifié)
                + (cache KV, qui grandit avec num_ctx)
                + overhead
```

Le cache KV, c'est la mémoire où le modèle stocke le contexte déjà traité. Sur tes 16 Go, le modèle agentic par défaut — Devstral Small 24B en `q4_K_M` (~14 Go) — tient **tant que tu ne gonfles pas `num_ctx`**. Le

piège classique : tu passes `num_ctx` de 8192 à 32768 pour faire entrer plus de ta `monprojet`. Le cache KV double ou triple, et tu franchis la barre des 16 Go sans message d'erreur — juste l'offload silencieux.

Le diagnostic en une commande

```
ollama ps
```

Lis la colonne `PROCESSOR` (son interprétation détaillée est au **chapitre 4**). C'est l'instrument de diagnostic central de tout le chapitre. Voici comment t'en servir pour l'OOM :

Affichage `ollama ps`	Interprétation	Action
`100% GPU`	Tout est en VRAM, c'est sain	Rien à faire côté OOM
`58%/42% CPU/GPU` (ou tout %CPU non nul)	Offload partiel → débit effondré	Réduire `num_ctx` ou la taille du modèle
`100% CPU`	Le GPU n'est pas utilisé du tout	Va en **Section B** (ce n'est pas un OOM)

Tableau 13.1 — Lecture de `ollama ps` pour diagnostiquer l'OOM.

Le fix (par ordre, du moins destructeur au plus)

1. Baisser `num_ctx` d'abord. C'est le levier n°1 et le moins coûteux. Repasse à une valeur qui rentre (8192 est le défaut chat/agent sur 16 Go). Vérifie avec `ollama ps`, puis remonte par paliers jusqu'à juste avant l'offload.

```
# Test direct : impose un contexte modeste et regarde si on repasse 100% GPU
OLLAMA_CONTEXT_LENGTH=8192 ollama run devstral:24b
```

Côté Modelfile (cf. **chapitre 11** pour la structure complète), la même chose en dur :

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

2. Descendre d'un cran de quantification ou de taille. Si ça déborde même à `num_ctx` raisonnable, ton modèle est trop gros pour ta carte. Reste en `q4_K_M` (le compromis qualité/place par défaut), pas plus gourmand. Le réglage fin temp/top-p/`num_ctx` est traité au **chapitre 12** ; le choix du bon modèle pour ta VRAM, au **chapitre 5**.

3. Plafonner explicitement les couches GPU (cas limite). Tu veux forcer un partage maîtrisé plutôt que subir l'offload automatique ? `num_gpu` fixe le nombre de couches envoyées au GPU :

```
FROM devstral:24b
PARAMETER num_ctx 8192
PARAMETER num_gpu 99   # "le plus possible" ; baisse ce nombre si tu veux délester
volontairement vers le CPU
```

> ⚠️ **AVERTISSEMENT — `num_gpu` n'est pas une formule magique.** Le baisser ne « libère » pas de la performance. Au contraire : ça envoie *plus* de couches sur le CPU, donc ça ralentit. C'est un outil pour les cas où tu veux faire tourner un modèle volontairement trop gros, en acceptant la lenteur. Ce n'est pas un réglage de tous les jours. En usage normal, laisse Ollama gérer. Joue plutôt sur `num_ctx` et la taille du modèle.

Pour TA VRAM :

VRAM	Réflexe OOM principal
8 Go	Le tier agentic dense 24B est hors de portée ; vise un modèle léger (Qwen2.5-Coder 7B) et `num_ctx` ≤ 8192. Toute montée de contexte se paie cash.
12 Go (RTX 5070, RTX 4070)	Devstral Small 24B NE rentre PAS ici : choisis Qwen2.5-Coder 14B (~9 Go) ou DeepSeek-Coder-V2-Lite 16B (~10 Go) en `q4_K_M`, `num_ctx` 4096. Au-delà, surveille `ollama ps`.
16 Go (ANCRE — RTX 5070 Ti / 4080 / 5080 / RX 9070 XT, Mac M4 24 Go)	Devstral Small 24B `q4_K_M` (~14 Go) + `num_ctx` 8192 = sweet spot agentic. Marge aussi pour Qwen2.5-Coder 14B en Q8.
24 Go (RTX 3090/4090, RX 7900 XTX, Mac M4 Pro 48 Go)	Tu peux te permettre Qwen2.5-Coder 32B `q4_K_M` (~19 Go), du Devstral en Q5/Q6, ou du contexte long sans offload ; l'OOM devient rare.
Apple Silicon (Metal)	La VRAM est partagée avec la RAM système (mémoire unifiée). Pas d'OOM brutal mais une pression mémoire qui swappe : surveille la pression mémoire système, ferme les gros process avant une grosse session.

Tableau 13.2 — Réflexe OOM par palier de VRAM.

Section B — Le GPU n'est pas utilisé du tout

Le symptôme

`ollama ps` affiche `100% CPU`, le ventilo du GPU dort, et la génération est lente partout — pas effondrée comme un offload, juste lente *partout*, dès le premier token. C'est le cas le plus rageant : le matériel est là, mais inexploité.

La cause

Ollama ne voit pas ton accélérateur. Trois causes courantes : - Le runtime GPU n'est pas détecté (drivers CUDA / ROCm absents ou trop vieux). - Sous Linux, Ollama tourne dans un contexte (conteneur, service systemd) qui n'a pas accès au device GPU. - Sous Windows/WSL, le passthrough GPU n'est pas configuré.

Le diagnostic

```
# NVIDIA (ta 5070 Ti) — le GPU doit apparaître et lister un process ollama pendant une
génération
nvidia-smi

# AMD ROCm
rocm-smi

# Et surtout : lire les logs Ollama au démarrage, où il annonce ce qu'il détecte
```

Sous Linux, les logs du service :

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

Tu cherches une ligne où Ollama dit avoir trouvé une librairie GPU (CUDA/ROCm). S'il annonce qu'il bascule sur CPU, tu tiens ta cause. (L'installation propre du runtime selon ton OS est couverte au **chapitre 4** — si cette section te concerne, c'est souvent que l'étape GPU du ch.4 a été sautée.)

Le fix

(L'extrait s'arrête ici. La suite de la Section B — et les sections C « dépassement de contexte », D « mauvais tag », E « autocomplétion FIM lente », plus le tableau de référence symptôme → cause → fix — sont dans le guide.)