

Agents IA en local

*Faire tourner un modèle, puis en faire un agent —
GPU, CPU, ARM, ESP32*



*Raspberry Pi 5 — le
poste, le CPU*



*ESP32-P4-WIFI6
— le
microcontrôleur*

#1 · Un agent, c'est un modèle plus un harnais

Imed MAGROUNE

Un agent = un modèle + un harnais



#1 · Un agent, c'est un modèle plus un harnais

LA CHAÎNE LOCALE

des poids au dépôt jusqu'à l'agent : quatre étapes, toujours les mêmes



#1 · Un agent, c'est un modèle plus un harnais

Ce que vous saurez faire ce soir

Les familles que nous croiserons : Qwen, Gemma, DeepSeek — et une vague de petits modèles entraînés pour les harnais.

- Lancer un serveur de modèle compatible OpenAI sur votre machine
- Mesurer vos tokens par seconde : préremplissage et génération
- Brancher un agent et lui confier une vraie tâche d'exploitation
- Refermer les accès : utilisateur dédié, conteneur, validation humaine

LES FAMILLES DE MODÈLES (2026)

celles que nous croiserons dans le cours

 Qwen Alibaba — la famille ouverte la plus large	 DeepSeek raisonnement ouvert, poids publiés
 Gemma Google DeepMind — du téléphone au poste	 Granite IBM — dense 3, 8 et 30 Md, contexte 512K

Les familles de modèles (2026) — fermés et poids ouverts

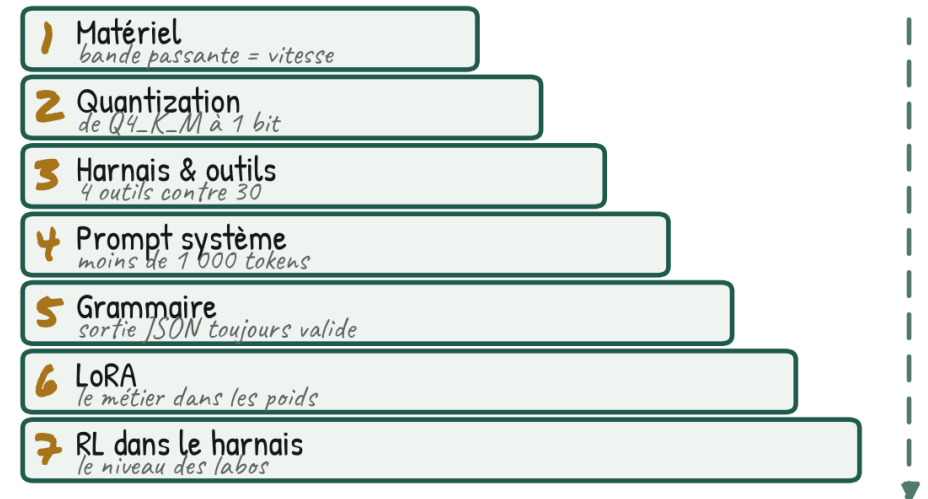
L'échelle d'intervention — le fil du cours

- Sept marches, du matériel à l'entraînement par renforcement
- Chaque marche coûte plus cher et rapporte plus de fiabilité
- Aujourd'hui : les marches 1 à 5 — les 6 et 7 sont le chemin des labos

#1 · Un agent, c'est un modèle plus un harnais

L'ÉCHELLE D'INTERVENTION

sept façons d'améliorer un agent local — de la moins chère à la plus puissante
on règle le système on modifie le modèle



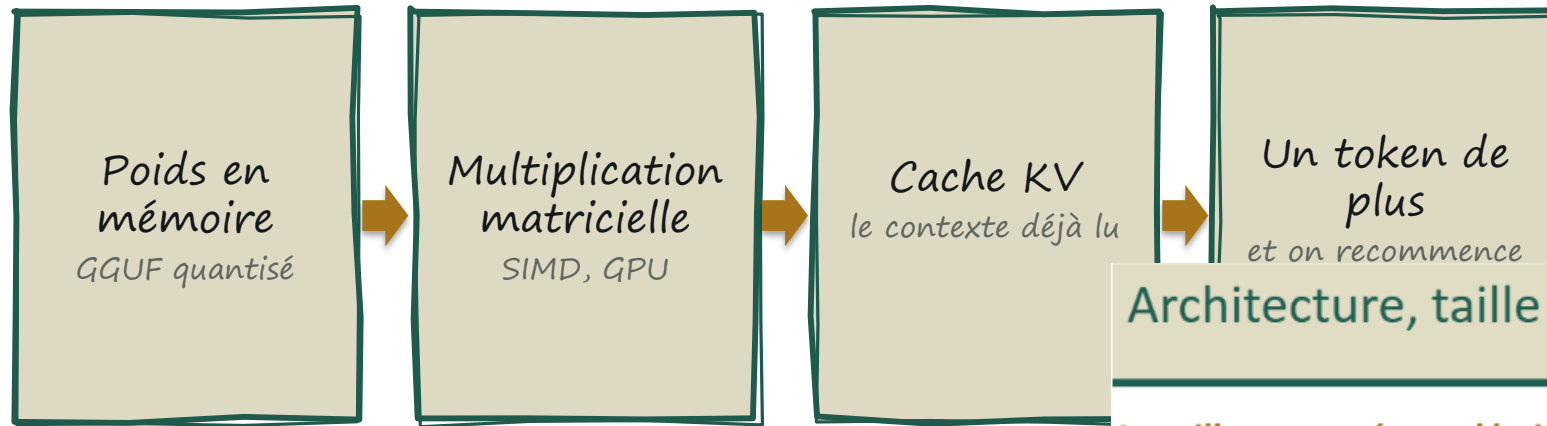
profondeur d'intervention

plus on descend, plus ça coûte (temps, données, matériel) et plus ça rapporte (fiabilité)

Anatomie de l'inférence locale

GGUF, quantization, budget mémoire : ce qui décide si ça tient et si c'est rapide

Le chemin d'un token



Architecture, taille et modalités

La taille annoncée ne décrit pas tout le modèle.

	Dense	MoE : mélange d'experts
Calcul par token	Réseau principal mobilisé	Quelques experts sélectionnés
Taille à lire	Paramètres totaux	Paramètres totaux ET actifs
Mémoire des poids	Dépend du total et des bits	Dépend aussi du total stocké

LES OUTILS DE LLAMA.CPP

un moteur, un serveur, un banc d'essai — installés en une ligne



Petits models et besoins matériel

Petit dès le départ

Peu de paramètres, beaucoup de données.

Distillation

Apprendre à partir d'un modèle professeur.

Pruning / élagage

Retirer des structures, puis réentraîner.

Quantification

Moins de bits par poids, même nombre.

**Entraîner demande plus de mémoire
que faire de l'inférence.**

Mémoire minimale des poids

Taille	BF16	Q4
2B	4 Go	1 Go
4B	8 Go	2 Go
8B	16 Go	4 Go
32B	64 Go	16 Go

Mémoire $\approx$ paramètres $\times$ bits / 8

Repères Q4, contexte court et 1 session :

8 Go RAM : 2–4B

16 Go VRAM : 8–14B

Poids seuls en Go décimaux. Q4 idéal, hors métadonnées, contexte et moteur. Sources en notes.

2

Quantifier : faire tenir le modèle

- GGUF, c'est le format standard du local : des poids compressés
- $Q4_K_M \approx 0,6 \times$ les paramètres : 7 Md $\rightarrow$ 4,4 Go, le bon compromis
- $Q8 \approx 1,0 \times$, quasi sans perte ; $Q4$ 3-bit, $Q3$, des tailles intermédiaires
- Extrême : Bonsai 8B, 8,19 Md de paramètres en 1,15 Go
- Ses poids sont à 1 bit dès l'entraînement, pas quantifiés après coup
- Le format Q1_0 est intégré à la version officielle de llama.cpp

BONSAI 8B — LE CAS EXTRÊME

PrismML : les poids à 1 bit dès l'entraînement



8,19 Md de paramètres en 1,15 Go

poids à 1 bit dès l'entraînement, pas quantifiés après coup
format Q1_0 intégré à llama.cpp · licence Apache 2.0

Bonsai 8B (PrismML) — 8,19 Md de paramètres en 1,15 Go, poids à 1 bit natifs, format Q1_0 intégré à llama.cpp

Choisir la bonne distribution

Préentraînement

Langues, domaines, code, mathématiques, images, qualité et fraîcheur des données.

Post-entraînement

Base, Instruct, Reasoning, appels d'outils...
Des comportements entraînés différemment.

Format et précision

Format compatible avec le moteur, puis quantification
Exemple : GGUF en Q4 pour llama.cpp.

Le choix se mesure sur vos tâches : qualité, m

Sources : model cards MiniCPM5 et Qwen3.5. Vérifier aussi la licence et la compatibilité du mot

Exemple MiniCPM5

Web + code + math

Septembre 2026 : le SOTA selon la tâche

MiniCPM5-2B face à Qwen3.5-4B

Un 2B peut dépasser un 4B sur certains tests.

Benchmark	MiniCPM5 2B*	Qwen3.5 4B
LiveCodeBench v6	69,1	56,4
SWE-bench Verified	46,4	33,6
MMLU-Pro	70,8	78,0

*2,52B paramètres au total, dont 1,98B hors embeddings.

élection de modèles ouverts au 17/09/2026. Scores OpenBMB, non un classement universel.

DeepSeek-V4.1-Flash

MoE multimodal

552B de backbone
+ 196B de mémoire Engram

8B actifs en préfill
16B actifs en génération

Contexte jusqu'à 1M tokens

Serveur ou API

Les deux vitesses d'un modèle

Lecture du prompt (préremplissage)

- Tous les tokens d'un coup, en parallèle
- Fixe l'attente avant le premier token
- C'est là que le Raspberry Pi souffre

Génération (décodage)

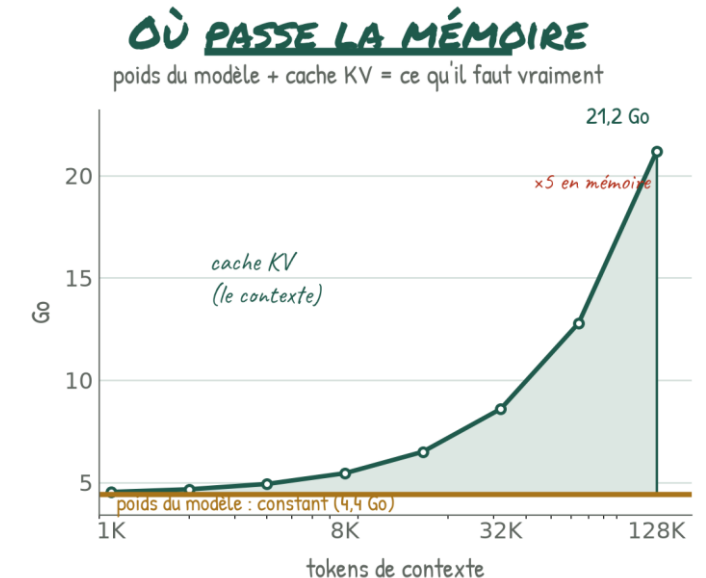
- Un token à la fois, séquentiel
- Mesurée en tokens par seconde
- Bornée par la bande passante mémoire

llama-bench mesure les deux : pp512 pour le préremplissage, tg128 pour la génération.

Le budget mémoire

- Les poids sont constants : 4,4 Go pour un 8 Md en Q4_K_M
- Le cache KV croît avec le contexte : il double tous les 32K
- À 128K de contexte, le cache dépasse les poids : 21 Go contre 4,4
- Règle de travail : poids + cache + 1 à 2 Go de marge

#2 · Anatomie de l'inférence locale : mémoire et vitesse



exemple : 8 B en Q4_K_M, 32 couches, 8 têtes KV (GQA), cache fp16 – calcul, pas mesure
Poids (constant) + cache KV (croît avec le contexte) — exemple d'un 8 Md en Q4_K_M à 8 têtes KV

TP1 — llama-server en 15 minutes

*Une installation en une ligne, une commande, une API
compatible OpenAI*

LES OUTILS DE LLAMA.CPP

un moteur, un serveur, un banc d'essai — installés en une ligne



llama

llama.cpp

le moteur : CPU, GPU, Raspberry Pi



llama-bench

mesure préremplissage et génération



llama serve

l'API OpenAI + l'interface web



Hugging Face

les poids et les commandes à copier

HUGGING FACE

le dépôt public des poids — le point de départ du cours



Chaque carte de modèle donne :

l'installation en une seule ligne,
la commande llama serve -hf <modèle>,
les extraits de configuration à copier.

Installer — trois chemins

- Linux et macOS : l'installateur en une ligne de llama.app
- Windows : winget install llama.cpp (paquet ggml.llamacpp)
- Les pages Hugging Face affichent désormais l'installation et les commandes
- Elles donnent aussi des extraits de configuration prêts à copier

HUGGING FACE

Le dépôt public des poids — le point de départ du cours



Chaque carte de modèle donne :
l'installation en une seule ligne,
la commande llama serve -hf <modèle>,
les extraits de configuration à copier.

Hugging Face — le dépôt public des poids ; les cartes de modèles donnent l'installation en une ligne et les commandes à copier

Sur Linux ou macOS

Une ligne d'installation, une commande de lancement — c'est tout.

```
● ● ● bash – ~/cours
```

```
# une seule ligne pour installer
```

```
$ curl -LsSf https://llama.app/install.sh | sh
```

```
llama installé dans ~/.local/bin
```

```
# lancer un serveur : llama serve -hf <modèle>
```

```
$ llama serve -hf Qwen/Qwen3.5-4B-GGUF --port 8080
```

```
listening on http://127.0.0.1:8080 – API OpenAI + interface web
```

Sur Windows

Même chose par le gestionnaire de paquets de Windows, puis le serveur.

Windows PowerShell

installation par le gestionnaire de paquets de Windows

```
PS> winget install llama.cpp
```

Le paquet ggml.llamacpp a été installé.

même commande ensuite, avec un contexte un peu plus large

```
PS> llama serve -hf Qwen/Qwen3.5-4B-GGUF --port 8080 -c 8192
```

```
Listening on 127.0.0.1:8080
```

Le test en trente secondes

Deux appels suffisent à prouver que tout est branché.

```
● ● ● bash – le serveur répond-il ?
```

```
# 1. Le serveur est-il vivant ?
```

```
$ curl -s http://localhost:8080/v1/models
```

```
{"object": "list", "data": [{"id": "Qwen/Qwen3.5-4B-GGUF"}]}
```

```
# 2. Le client Python openai, sans rien changer d'autre
```

```
$ python3 client.py # base_url = http://localhost:8080/v1
```

```
réponse complète, en flux, sans quitter la machine
```

Ce qui peut casser (et la réponse)

- Contexte par défaut trop court : le harnais exige `-c 64000`
- Modèle qui refuse de charger : quantization trop grosse pour la mémoire
- Pas de GPU reconnu : le serveur bascule en CPU, plus lent mais correct
- Tags et options changent : vérifier les commandes la veille

L'échelle matérielle

Llama bench sur quatre marches : GPU 16 Go (voire 8 Go + RAM) · CPU + 64 Go · Raspberry Pi 5 · ESP32-P4

#4 · L'échelle matérielle : llama-bench sur quatre marches

Mesurer avant de promettre

- llama-bench mesure le préremplissage (pp512) et la génération (tg128)
- Quatre marches : GPU 16 Go (voire 8 Go + RAM), CPU puissant, Pi 5, ESP32-P4
- On note les tokens par seconde mesurés, jamais les chiffres annoncés
- Toute la démonstration alimente le TD : une ligne par marche



*ESP32-P4-WIFI6
(Waveshare) — bicoeur RISC-
V 400 MHz + ESP32-C6
(Wi-Fi 6), RAM externe : le
terrain des LLM de quelques
mégaoctets*

La commande du TD

Une seule commande, lancée sur les quatre machines du TD.

```
● ● ● bash – llama-bench
```

```
# Le banc d'essai officiel : préremplissage (pp) et génération (tg)
```

```
$ llama-bench -m modele-Q4_K_M.gguf -p 512 -n 128
```

```
| model | backend | pp512 t/s | tg128 t/s |
```

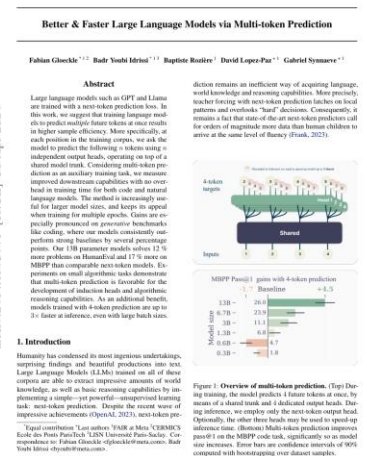
```
| à vous de remplir chaque ligne du tableau |
```

```
# Les quatre marches se mesurent avec cette seule commande
```

Marche 1 — GPU 16 Go (voire 8 Go + RAM)

- Qwen3.8-27B dense en Q4_K_M : 17,1 Go de poids — il ne rentre pas dans 16 Go
- À 16 Go on vise un dense 12-14 Md en Q4 (7 à 9 Go) ; à 8 Go + RAM, l'offload fait le reste
- Prédiction multi-tokens : llama.cpp l'active sur les modèles équipés
- MTP : 1,4 à 2,2× pour Gemma 4, jusqu'à +120 % sur Qwen3.8-27B, sans perte de précision

#4 · L'échelle matérielle : llama-bench sur quatre marches



Better & Faster Large Language Models via Multi-token Prediction — Gloeckle et al., Meta AI, 2024 (arXiv:2404.19737)

Marche 2 — CPU + 64 Go : les MoE

- Qwen3.6-35B-A3B : 35 Md au total, 3 Md actifs par token
- Gemma 4 26B-A4B : 25,2 Md au total, 3,8 Md actifs par token
- Seule une fraction des experts est lue pour chaque token
- D'où une vitesse qui surprend pour la taille : à mesurer
- Le goulot reste la lecture du prompt : surveillez le préremplissage

#4 · L'échelle matérielle : llama-bench sur quatre marches

LES OUTILS DE LLAMA.CPP

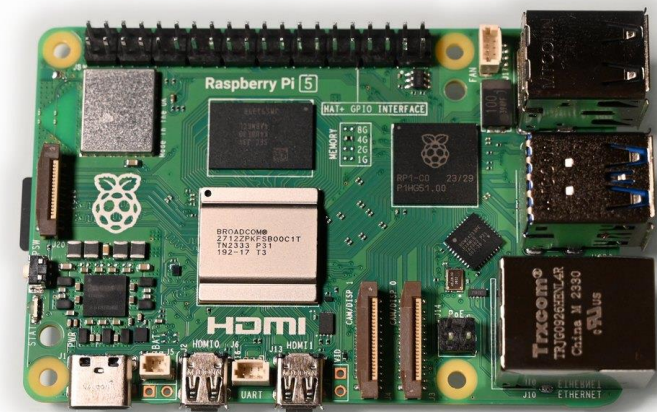
un moteur, un serveur, un banc d'essai — installés en une ligne

 llama llama.cpp Le moteur : CPU, GPU, Raspberry Pi	 llama-bench mesure préremplissage et génération
 llama serve l'API OpenAI + l'interface web	 Hugging Face les poids et les commandes à copier

*llama.cpp (le moteur) · llama serve
(l'API compatible OpenAI) · llama-bench (mesurer)*

Marche 3 — Raspberry Pi 5

- Chiffres officiels Google (LiteRT-LM, pas llama.cpp) : ordre de grandeur
- Pi 5 16 Go, CPU : E2B lit le prompt à 133 tokens/s et génère à 8
- E4B : 51 tokens/s en lecture, 3,2 en génération
- Dix mille quatre cents tokens de contexte : environ 4,6 minutes avant le premier token
- Moins de 1 000 tokens : une vingtaine de secondes
- En llama.cpp, le préremplissage est plus lent : 9 à 13 minutes



Raspberry Pi 5 (~80 €) — les modèles locaux tiennent sur la carte

Marche 4 — ESP32-P4 : un LLM de 14 Mo

- Needle 2 (Cactus Compute) : 45 M de paramètres, 2 bits par poids
- Un seul binaire de 14 Mo, moteur compris — rien à installer à côté
- Session complète dans ~28 Mo de RAM ; les appels d'outils sont contraints par grammaire
- 500 tokens/s sur Pi 5 ; sur ESP32-P4, la RAM externe suffit

#4 · L'échelle matérielle : llama-bench sur quatre marches

NEEDLE 2 — UN LLM DE 14 MO

Cactus Compute : 45 M de paramètres, pour les appareils qui n'ont ni GPU ni NPU



CACTUS COMPUTE

- 45 M de paramètres, 2 bits par poids (Cactus Quants)
- un seul binaire de 14 Mo, moteur compris
- session complète dans ~28 Mo de RAM (fenêtre 256 tokens)
- appels d'outils contraints par grammaire ; refus si hors sujet
- 500 tokens/s sur Raspberry Pi 5 ; tourne sur ESP32-P4 (RAM externe)

Needle 2 (Cactus Compute) — 45 M de paramètres, un seul binaire de 14 Mo, une session dans ~28 Mo de RAM : un LLM d'agent sur microcontrôleur

TD — ce que vous notez pendant la démo

<i>marche</i>	<i>modèle type</i>	<i>ce qu'on mesure</i>	<i>limite attendue</i>
GPU 16 Go (voire 8 Go + RAM)	dense 12-14 Md Q4, ou 27B en Q3	tokens/s avec et sans MTP, contexte max	poids et cache à surveiller
CPU + 64 Go	MoE, 3 à 4 Md actifs	tokens/s, effet de la quantisation	lecture du prompt lente
Pi 5	2 à 4 Md Q4, dont un modèle 1 bit	temps d'ingestion du prompt système	missions courtes et ciblées
ESP32-P4	Needle 2 : 45 M en 2 bits, 14 Mo	temps de réponse, RAM libre	une seule tâche, jamais de conversation

Chaque ligne du TD est un arbitrage : ce qu'on gagne, ce qu'on paie, et la limite qu'on accepte.

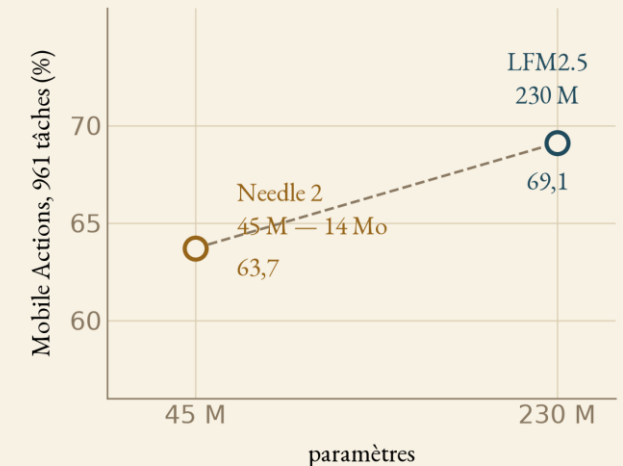
Ce que le TD nous apprend

- La vitesse se paie en mémoire et en bande passante
- Sur Pi, le coût n'est pas la génération mais la lecture du prompt
- Avant de changer de machine : réduire le contexte et les outils
- Un agent étroit sur un Pi bat un agent large qui n'arrive pas à démarrer

#4 · L'échelle matérielle : llama-bench sur quatre marches

Cinq fois plus petit, cinq points de moins

Needle 2 (Cactus) face à un modèle 5× plus gros



évaluation Mobile Actions (961 lignes) publiée par Cactus Compute

Needle 2 (45 M, 14 Mo) face à un modèle 5× plus gros — Mobile Actions, 961 tâches (Cactus Compute)

Panorama des modèles locaux

*Moins de 8 milliards de paramètres — état de
septembre 2026*

Moins de 4 Md (1/2)

- Nanbeige4.2-3B : 4 Md dont 3 hors embeddings, boucle $\times 2$, 262K
- Ses scores battent Qwen3.5-9B et Gemma 4 12B... selon son éditeur
- Dans llama.cpp depuis le 27 juillet 2026 ; sur Pi, la boucle coûte du calcul
- LFM2.5-2.6B : 2,7 Md, 128K, 113 tokens/s sur Ryzen en 2,5 Go
- Entraîné par renforcement directement dans les harnais agentiques

PETITS MODÈLES AGENTIQUES (2026)

moins de 8 milliards de paramètres — les six candidats



Nanbeige4.2-3B

4 Md dont 3 hors embeddings · boucle ×2



Qwen3.5-4B

0,8 / 2 / 4 / 9 Md · vision · mars 2026



LFM2.5-2.6B

2,7 Md · 128K · 113 t/s sur Ryzen



Ministral 3 3B

bons appels d'outils, hallucinations



Granite 4.1 3B

dense 3 / 8 / 30 Md · contexte 512K



Bonsai 8B

8 Md en 1 bit · 1,15 Go · Q1_0

Moins de 4 Md (2/2)

- Granite 4.1 3B (IBM) : dense 3, 8 et 30 Md, contexte 512K, Apache 2.0
- Qwen3.5-4B : sorti le 2 mars 2026 avec les 0,8 et 2 Md
- Contexte 262K, vision, Apache 2.0 : la famille la plus complète
- Ministral 3 3B : excellent en appel d'outils selon un testeur sur Jetson
- Mais beaucoup d'hallucinations d'outils : prévoir une validation

#5 · Panorama des modèles locaux : moins de 8 Md

LES FAMILLES DE MODÈLES (2026)

celles que nous croiserons dans le cours

 Qwen Alibaba — la famille ouverte la plus large	 DeepSeek raisonnement ouvert, poids publiés
 Gemma Google DeepMind — du téléphone au poste	 Granite IBM — dense 3, 8 et 30 Md, contexte 512K

*Les familles de modèles (2026) —
fermés et poids ouverts*

Entre 4 et 8 Md

- Qwen3.5-9B : le haut de la gamme compacte, 262K de contexte
- Granite 4.1 8B : appel d'outils nettement meilleur que Granite 4.0
- Bonsai 8B en 1 bit : un 8 Md dans 1,15 Go, taillé pour le Pi

#5 · Panorama des modèles locaux : moins de 8 Md

BONSAI 8B — LE CAS EXTRÊME

PrismML : les poids à 1 bit dès l'entraînement



8,19 Md de paramètres en 1,15 Go

poids à 1 bit dès l'entraînement, pas quantifiés après coup
format Q1_0 intégré à llama.cpp · licence Apache 2.0

Bonsai 8B (PrismML) — 8,19 Md de paramètres en 1,15 Go, poids à 1 bit natifs, format Q1_0 intégré à llama.cpp

« Les scores de Nanbeige et de LFM2.5 sont auto-déclarés. Faites votre banc d'essai avant l'école : la même tâche d'exploitation, sur le Pi, dans Pi, avec trois ou quatre modèles. Vos chiffres convaincront plus qu'un tableau. »

— le conseil du cours

Questions

Cinq minutes de tampon — puis une pause de 15 à 20 minutes

Ce qu'on retient de la session 1

- Un agent, c'est un modèle plus un harnais
- La mémoire se paie deux fois : les poids, puis le contexte
- Le serveur compatible OpenAI est la porte d'entrée de tout le reste
- Sur un Pi, on paie la lecture du prompt avant de payer la génération

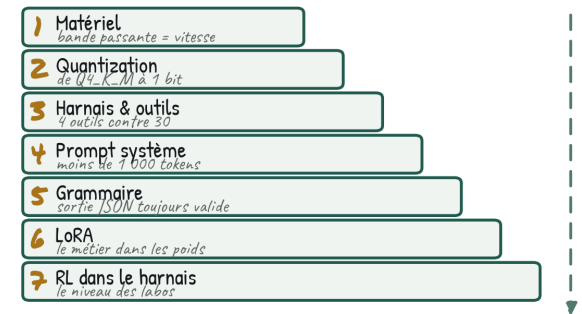
Session 2 : l'agent en action

- Anatomie d'un agent, et le coût caché des outils
- TP2 : un agent de dev branché sur votre serveur
- TP3 : un agent d'exploitation, en lecture seule puis encadré
- Sécurité, grammaires, et les marches 6 et 7 de l'échelle

#6 · Questions et tampon

L'ÉCHELLE D'INTERVENTION

sept façons d'améliorer un agent local — de la moins chère à la plus puissante
on règle le système on modifie le modèle



profondeur d'intervention

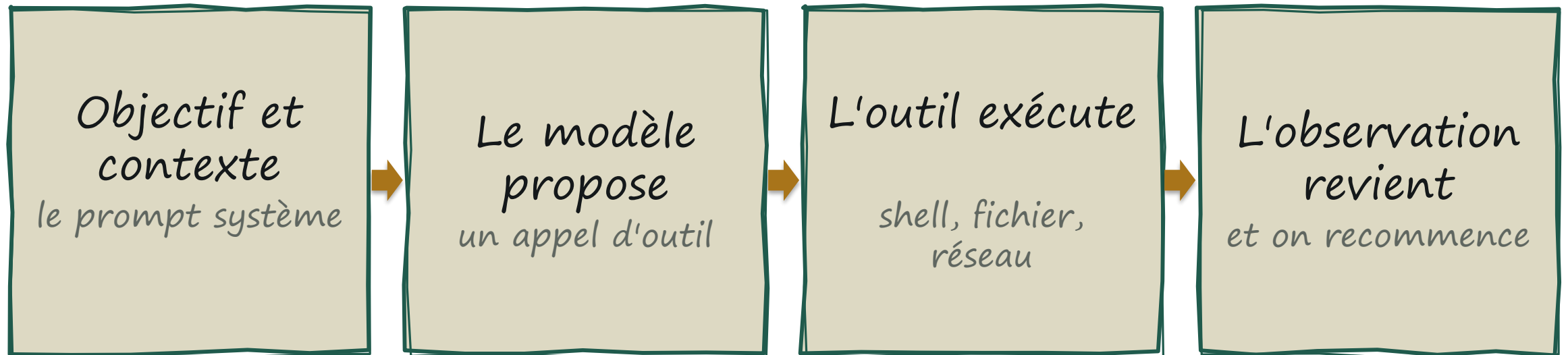
plus on descend, plus ça coûte (temps, données, matériel) et plus ça rapporte (fiabilité)

Sept marches, du matériel à l'entraînement RL dans le harnais — chacune coûte plus cher et rapporte plus de fiabilité

Anatomie d'un agent

*La boucle, les outils, MCP — et pourquoi le harnais
décide de tout*

La boucle agentique



Deux philosophies d'outillage

- Pi : quatre outils — read, write, edit, bash
- Et un prompt système de moins de 1 000 tokens
- À comparer aux 14 000 tokens du prompt système de Claude Code
- L'autre école : brancher des serveurs MCP, dizaines d'outils décrits en détail
- Sur une machine locale, la différence n'est pas une opinion : elle se mesure



*There are many agent harnesses
but this one is yours*

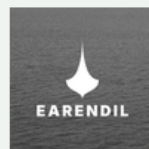
LES HARNAIS AGENTIKES

la couche qui transforme un modèle en agent — et qui l'édite



Hermes Agent

outils, mémoire, sous-agents



Pi

4 outils, < 1 000 tokens de prompt



OpenCode

l'agent de code en terminal



OpenClaw

assistant personnel, ex-Moltbot

Le coût caché de MCP

- Les descriptions d'outils occupent le contexte avant le premier mot
- Serveur MCP Playwright : 13 700 tokens pour 21 outils
- Serveur MCP Chrome DevTools : environ 18 000 tokens
- Un README décrivant des scripts CLI : 200 à 400 tokens
- Sur un Pi, ces tokens se paient en minutes avant le premier token

#7 · Anatomie d'un agent : boucle, outils, MCP

LE COÛT CACHÉ DE MCP

Les tokens réservés aux outils, avant le premier mot



*Tokens réservés aux descriptions d'outils
— Chrome DevTools $\approx$ 18 000,
Playwright 13 700, README de
scripts CLI $\approx$ 300*

Pi, OpenClaw et Hermes

- Pi est un agent de code minimal, open source, à quatre outils
- OpenClaw s'appuie sur Pi comme moteur central
- Hermes Agent : le harnais complet — outils, mémoire, sous-agents, tâches planifiées
- Le même modèle change de comportement selon le harnais qui le pilote

OPENCLAW

assistant personnel open-source (ex-Moltbot) — un agent qui agit

*OpenClaw — assistant personnel open source,
ex-Moltbot*

Brancher un harnais sur le serveur local

Côté harnais, tout tient dans une URL et un nom de modèle.

● ● ● extrait de configuration – harnais local

```
# Pi ou OpenCode : une URL et un nom de modèle, rien d'autre (enfin si :  
conetxe, thinking, ...)  
{ "provider": "local",  
  "baseUrl": "http://localhost:8080/v1",  
  "model": "Qwen/Qwen3.5-4B-GGUF" }  
# Le harnais ne voit aucune différence avec une API distante  
# Sandboxing
```

TP2 — un agent de dev sur votre modèle

Pi ou OpenCode branchés sur le serveur de la session 1

Brancher un harnais sur le serveur local

- Le serveur expose /v1 : le harnais ne voit aucune différence
- Pi : fournisseur compatible OpenAI, baseURL sur localhost:8080
- OpenCode : la même configuration, dans son fichier de providers
- Les pages Hugging Face donnent des extraits de configuration prêts à copier

LES HARNAIS AGENTIKES

la couche qui transforme un modèle en agent — et qui l'édite

 Hermes Agent outils, mémoire, sous-agents	 Pi 4 outils, < 1000 tokens de prompt
 OpenCode l'agent de code en terminal	 OpenClaw assistant personnel, ex-Moltbot

*Hermes Agent · OpenCode · Pi · OpenClaw —
la couche qui transforme un modèle en agent*

La tâche, dans cet ordre

- 1. Lire le dépôt et le résumer en dix lignes
- 2. Écrire un test qui échoue, et le montrer
- 3. Modifier le code jusqu'à ce que le test passe
- 4. Afficher le diff et le journal de la boucle

Ce qu'on observe en direct

- Le modèle perd le fil au-delà de quelques tours
- Chaque tour rejoue le contexte : la lecture du prompt s'additionne
- Un fichier, un objectif, une vérification : la règle qui tient
- Quand un agent s'arrête, la cause est presque toujours le contexte

TP3 — un agent d'exploitation

Lire d'abord, agir ensuite, et toujours laisser une trace

Trois temps



Temps 1 — le rapport de santé

- L'agent lit : `df`, `free`, `journalctl`, `systemctl status`, `nvidia-smi`
- Il rend un rapport structuré — pas des commandes à copier
- Le compte SSH est en lecture seule : rien ne peut être cassé
- Sur le Pi, la première mesure est le poids du rapport en contexte

Temps 1 — lire, sans rien écrire

Le rapport de santé : ce que l'agent observe, sans jamais modifier.

● ● ● bash — le rapport de santé par SSH

L'agent lit ; le compte est en lecture seule

\$ ssh agent@pi5.local 'df -h; free -m; uptime'

/dev/root 62% Mem: 3 812 / 16 384 Mo load: 0,42

\$ ssh agent@pi5.local 'journalctl -p err -n 20 --no-pager'

2 erreurs sur 24 h, les deux dans le service d'impression

Temps 2 et 3 — agir sans se tromper

- Le plan est proposé avant toute écriture, en dry-run
- L'humain valide ; l'agent exécute une action à la fois
- Chaque action laisse une preuve : avant, après, journal
- En démo, l'agent de supervision tourne sur le Pi, déclenché par cron

La règle du TP3

0 sudo

les droits d'administrateur que l'agent
d'exploitation n'a pas
*c'est précisément ce qui le rend utilisable sur une machine en
service*

Sécurité : le conteneur n'est pas une option

Utilisateur dédié, droits restreints, journalisation — et un mur autour de l'agent

Le point dur

- Pi n'a pas de sandbox intégré : il tourne avec les droits de l'utilisateur
- Aucune fenêtre d'approbation avant bash : le modèle décide seul
- Sa documentation recommande explicitement l'isolation par conteneur
- Conséquence pour le TP3 : le conteneur n'est pas une option, c'est la condition

Le conteneur, en une commande

L'isolation se décide ici, avant même que le modèle parle.

● ● ● bash – l'agent dans un conteneur

pas de réseau, journaux en lecture seule, rapports à part

```
$ docker run --rm -it --network none -v /srv/logs:/logs:ro -v  
/srv/rapports:/out agent-pi:1.0
```

```
agent@b7f1c2:/work$ # 4 outils, aucun droit d'administration
```

à l'intérieur : utilisateur dédié, sudoers restreint, dry-run

Les garde-fous qui marchent

- Un utilisateur système dédié, sans sudo
- Un sudoers restreint : des commandes nommées, jamais ALL
- Le dry-run par défaut : l'agent propose, l'humain dispose
- La journalisation de chaque commande et de sa sortie
- Un réseau limité aux hôtes nécessaires

L'injection par les données

- Un fichier de contexte, un commentaire ou une page lue peut porter des ordres
- Tout ce qui entre dans le contexte est une entrée non fiable
- Contre-mesures : conteneur, liste blanche d'outils, validation humaine
- Un README d'apparence inoffensive peut détourner la boucle de l'agent

« Un agent qui propose est utile. Un agent qui exécute sans témoin est un incident en attente. »

— la règle du TP3

Au-delà du harnais

Diagnostiquer, contraindre, puis entraîner — les marches 5 à 7

Diagnostic : pourquoi mon agent s'arrête

- Contexte trop court : la conversation déborde, le modèle perd le fil
- Mauvais gabarit de chat : le modèle ne reconnaît pas les outils
- Prompt système trop lourd : il mange le contexte utile
- Trois causes, trois correctifs — avant de changer de modèle

Levier 1 — la même tâche, deux harnais

Harnais lourd

- 14 000 tokens de prompt système et d'outils
- Quatre à cinq minutes avant le premier token sur Pi
- L'agent se noie dans ses propres outils

Pi et outils étroits

- Moins de 1 000 tokens de prompt système
- Une vingtaine de secondes avant le premier token
- L'agent agit, on le regarde faire

Même tâche, même modèle, même Pi. Seul le harnais change : c'est la démonstration avant/après de la séance.

#11 · Au-delà du harnais : diagnostiquer, contraindre, entraîner

Levier 2 — GBNF : contraindre la sortie

- Les grammaires forcent la forme : JSON valide, clés connues d'avance
- Bonsai 8B en JSON libre : sorties invalides ; avec grammaire : 92 %
- La grammaire répare précisément ce défaut, sans toucher aux poids
- Marche 5 de l'échelle : peu coûteuse, très rentable sur les petits modèles

#11 · Au-delà du harnais : diagnostiquer, contraindre, entraîner

BONSAI 8B — LE CAS EXTRÊME

PrismML : les poids à 1 bit dès l'entraînement



8,19 Md de paramètres en 1,15 Go

poids à 1 bit dès l'entraînement, pas quantifiés après coup
format Q1_0 intégré à llama.cpp · licence Apache 2.0

Bonsai 8B (PrismML) — 8,19 Md de paramètres en 1,15 Go, poids à 1 bit natifs, format Q1_0 intégré à llama.cpp

Levier 2 — la commande

La grammaire s'ajoute au lancement du serveur, sans toucher aux poids.

```
● ● ● bash — grammaire GBNF
```

```
# sans grammaire : sur un modèle 1 bit, le JSON sort invalide
```

```
$ llama-server -m bonsai-8b-Q1_0.gguf --port 8080
```

```
# avec la grammaire : la sortie ne PEUT plus être invalide
```

```
$ llama-server -m bonsai-8b-Q1_0.gguf --grammar-file grammars/json.gbnf
```

```
# 0 % de sorties valides sans contrainte, 92 % avec
```

Le chemin — deux exemples réels

- Un LoRA communautaire de Qwen3.6-27B adapté au Pi
- Son auteur en tire une leçon : l'aisance avec le harnais compte autant que les scores
- LFM2.5-2.6B a été entraîné par renforcement DANS les harnais (marche 7)
- Le séminaire de neurochirurgie montre le même geste sur un autre métier

#11 · Au-delà du harnais : diagnostiquer, contraindre, entraîner

LES HARNAIS AGENTIKUES

La couche qui transforme un modèle en agent — et qui l'édite

	Hermes Agent outils, mémoire, sous-agents		Pi 4 outils, < 1000 tokens de prompt
	OpenCode l'agent de code en terminal		OpenClaw assistant personnel, ex-Moltbot

*Hermes Agent · OpenCode · Pi · OpenClaw —
la couche qui transforme un modèle en agent*

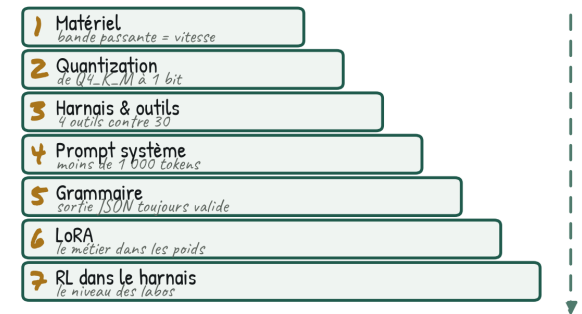
Où se situe ce que nous avons fait

- Marches 1 et 2 : le matériel et la quantization, réglés en une commande
- Marches 3 et 4 : le harnais et le prompt système, c'est là que se gagne la fiabilité
- Marche 5 : la grammaire, le meilleur rapport effort/résultat du cours
- Marches 6 et 7 : le chemin, déjà ouvert par la communauté

#11 · Au-delà du harnais : diagnostiquer, contraindre, entraîner

L'ÉCHELLE D'INTERVENTION

sept façons d'améliorer un agent local — de la moins chère à la plus puissante
on règle le système on modifie le modèle



profondeur d'intervention

plus on descend, plus ça coûte (temps, données, matériel) et plus ça rapporte (fiabilité)

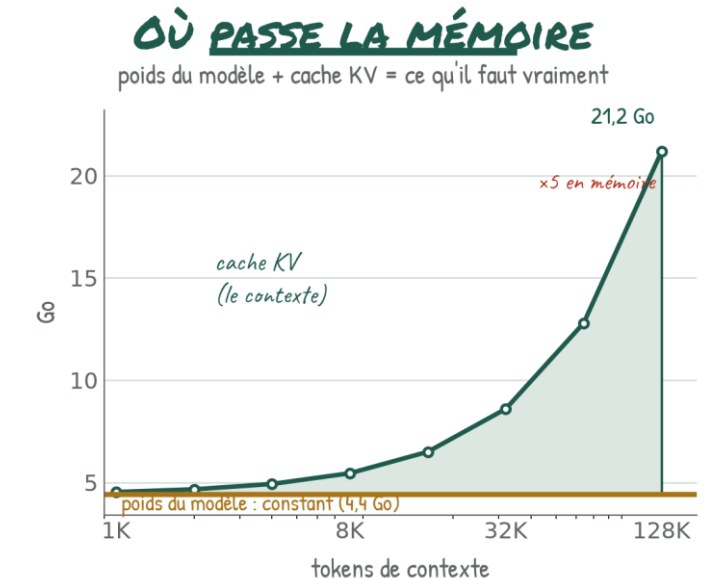
Sept marches, du matériel à l'entraînement RL dans le harnais — chacune coûte plus cher et rapporte plus de fiabilité

Synthèse

Trois idées

- Un agent, c'est un modèle plus un harnais
- La mémoire borne la vitesse : les poids, puis le contexte
- Le harnais coûte du contexte : chaque outil se paie en attente

#12 · Synthèse : votre prochaine marche



exemple : 8 B en Q4_K_M, 32 couches, 8 têtes KV (GQA), cache fp16 — calcul, pas mesure
Poids (constant) + cache KV (croît avec le contexte) — exemple d'un 8 Md en Q4_K_M à 8 têtes KV

Trois gestes à refaire chez vous

- Lancer un serveur compatible OpenAI : une ligne d'installation, une commande
- Mesurer votre machine avec llama-bench : préremplissage et génération
- Brancher un harnais étroit, dans un conteneur, avec un utilisateur dédié

#12 · Synthèse : votre prochaine marche

LES OUTILS DE LLAMA.CPP

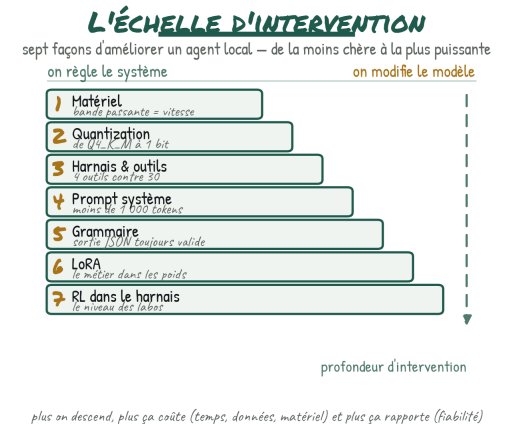
un moteur, un serveur, un banc d'essai — installés en une ligne

 llama llama.cpp Le moteur : CPU, GPU, Raspberry Pi	 llama-bench mesure préremplissage et génération
 llama serve l'API OpenAI + l'interface web	 Hugging Face les poids et les commandes à copier

llama.cpp (le moteur) · llama serve (l'API compatible OpenAI) · llama-bench (mesurer)

L'échelle d'intervention, une dernière fois

- Sous 4B (Md), on vise des missions courtes et ciblées
- L'écart se réduit : ces modèles sont entraînés dans les harnais
- Votre prochaine marche dépend de votre tâche — commencez par la mesurer



Sept marches, du matériel à l'entraînement RL dans le harnais — chacune coûte plus cher et rapporte plus de fiabilité