Scanderia

E-BOOK

INSTALLEZ VOTRE IA EN LOCAL :

POUR LE TRAVAIL OU LES RÉVISIONS

(AVEC IDRISS ABERKANE ET PHILIPPE ANEL)

RÉSUMÉ DE LA FORMATION LIVE DU 19 MAI 2026

FORMATION

Table des matières

 Ce livret accompagne la Formation Live 7 «Installer votre IA en local pour le travail ou les révisions ».

Il reprend la matière du live et l'approfondit.

Validé avec LM Studio 0.4.17 sur macOS (Apple Silicon), modèles Gemma 4 et Qwen 3. Dernière vérification : juin 2026. L'écosystème évolue vite : la méthode reste valable, mais certains écrans, versions ou chemins peuvent avoir changé.

Pour toute remarque ou suggestion : support@scanderia.com

Première partie

Installer et faire tourner

Chapitre 1

Quelle machine pour quel modèle

La question que les gens me posent en premier, c'est :
« est-ce que ça va tourner chez moi ? »

La réponse honnête, ce n'est pas un chiffre absolu de RAM, c'est une méthode. Ce chapitre vous donne cette méthode, plus les seuils que j'ai validés en pratique sur les machines courantes.

Ce qu'on cherche à mesurer

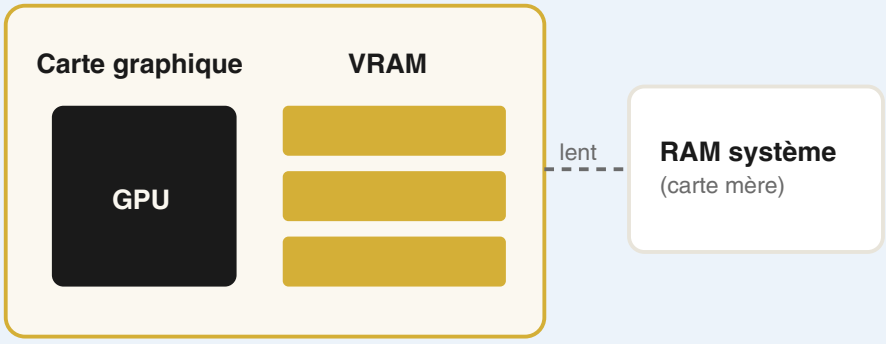
Quand on installe un modèle local, deux ressources comptent vraiment : la mémoire et la vitesse.

La **mémoire** détermine si le modèle rentre dans votre machine. Si le modèle ne rentre pas, il ne se lance même pas. C'est binaire. Sur un Mac avec mémoire unifiée, c'est la RAM système qui est aussi la mémoire GPU. Sur un PC avec carte graphique dédiée, c'est la VRAM de la carte qui compte; la RAM système sert de fallback lent.

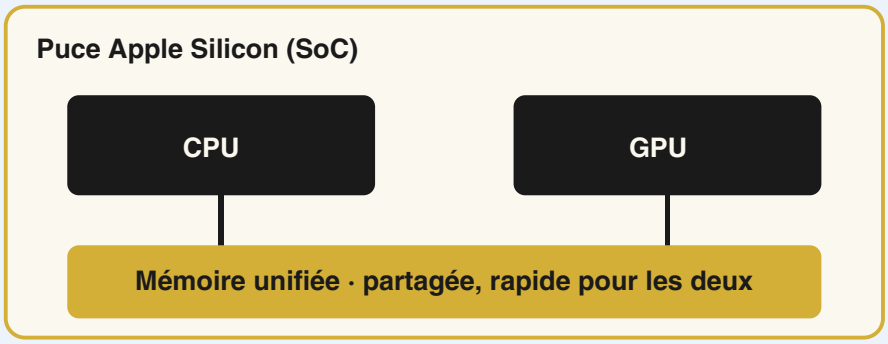
C'est quoi la VRAM?

VRAM veut dire ***Video RAM*** : c'est la mémoire vive embarquée directement sur la carte graphique, juste à côté du processeur graphique (le GPU). Elle est séparée de la RAM système, celle de la carte mère, et bien plus rapide d'accès pour le GPU.

Sur un PC à carte graphique dédiée, un modèle doit tenir dans cette VRAM pour tourner vite; s'il déborde, il bascule dans la RAM système, beaucoup plus lente. Sur un Mac à mémoire unifiée, la distinction disparaît : le processeur et le GPU partagent un seul pool de mémoire.



PC à carte graphique dédiée. ***La VRAM est embarquée sur la carte, à côté du GPU. La RAM système, sur la carte mère, ne sert que de réserve lente quand la VRAM déborde.***



Mac à mémoire unifiée. ***Un seul pool de mémoire, rapide, partagé par le CPU et le GPU. Pas de VRAM séparée ni de réserve lente : toute la mémoire est de la mémoire « GPU ».***

La **vitesse**, mesurée en tokens par seconde, détermine si l'expérience est utilisable. Un modèle qui tourne mais qui crache un mot toutes les trois secondes ne sert à rien pour du chat interactif. Pour de l'agent de fond qui résume vos documents la nuit, deux tokens par seconde suffisent. Le bon seuil dépend du cas d'usage.

Ces deux ressources se tiennent. Plus le modèle est gros, plus il prend de mémoire, et plus il décode lentement. Le compromis classique : prendre le plus gros modèle que votre machine fait tourner ***confortablement***, pas le plus gros qu'elle fait tourner ***du tout***.

Combien de mémoire pour un modèle

Pour estimer si un modèle rentre, multipliez son nombre de paramètres par le poids de chaque paramètre selon la quantization.

En FP16, chaque paramètre prend deux octets. Un modèle 8B en FP16 fait donc 16 Go. En Q4 (la quantization standard de LM Studio), chaque paramètre prend environ un demi-octet : le même modèle 8B en Q4 tombe à environ 5 Go. La Q4 perd quelques pourcents de qualité par rapport à la FP16, mais le gain mémoire est tel que c'est presque toujours le bon choix pour le local.

Voilà l'ordre de grandeur que j'utilise au quotidien :

Modèle	FP16	Q4
2B	~4 Go	~1,7 Go
4B	~8 Go	~2,5 Go
8B	~16 Go	~5 Go
14B	~28 Go	~9 Go
30B (MoE)	~60 Go	~18,5 Go

La colonne Q4 reprend la taille réelle du fichier Q4_K_M de modèles représentatifs : Gemma 2 2B (1,71 Go), Qwen3-4B (2,50 Go), Qwen3-8B (5,03 Go), Qwen2.5-14B (8,99 Go), Qwen3-30B-A3B (18,63 Go). Le «demi-octet par paramètre» est une approximation : en pratique le Q4_K_M tourne autour de 0,6 octet par paramètre, d'où des fichiers un peu plus lourds que la règle ne le laisse croire.

Attention au piège des modèles «E» : Gemma 4 E2B annonce 2 milliards de paramètres ***effectifs*** (le E veut dire ***effective parameters***). Ce n'est pas une architecture ***mixture-of-experts*** mais un modèle dense doté de ***per-layer embeddings*** : une partie des paramètres, les embeddings par couche, gonfle la taille stockée sur le disque sans alourdir le calcul, qui reste celui d'un 2B. Résultat, un E2B pèse plus lourd en Q4 qu'un 2B dense, autour de 3 Go au lieu de 1,7 Go, alors que sa vitesse reste celle d'un 2B. Devant un nom en E2B ou E4B, fiez-vous à la taille du fichier, pas au chiffre du nom. (source : carte du modèle Gemma 4, Google)

Combien de mémoire pour le contexte

Ce que la plupart des tutoriels oublient de dire, c'est que la taille de contexte consomme de la mémoire elle aussi. À côté du poids du modèle, LM Studio alloue un cache pour les tokens en cours (le KV-cache).

Pour Llama 8B en 128k tokens de contexte, le KV-cache en FP16 pèse environ 16 Go, autant que le modèle en pleine précision et bien plus qu'une version quantifiée Q4 (autour de 5 Go). Voilà pourquoi LM Studio, par défaut, plafonne le contexte bien en dessous du maximum du modèle : sans cette limite, charger un modèle avec son contexte plein saturerait la mémoire. (calculateur de VRAM et KV-cache)

En pratique, pour un usage chat normal, 8k à 30k tokens de contexte sont confortables et coûtent quelques gigas supplémentaires. Si vous voulez passer un PDF entier ou un long historique de conversation, montez à 64k ou 128k, mais surveillez la mémoire qui reste sur votre machine.

Ce qui détermine la vitesse : la bande passante mémoire

On a vu que la mémoire décide si le modèle ***rentre***. Reste la question de la vitesse, et là le facteur dominant surprend : ce n'est presque jamais la puissance de calcul, c'est le débit de la mémoire. Pour produire chaque token, le modèle doit relire l'intégralité de ses poids. Un modèle Q4 de 5 Go, c'est 5 Go à parcourir en mémoire pour ***chaque*** mot généré. La vitesse de décodage est donc plafonnée par la vitesse à laquelle la mémoire débite ces octets, ce qu'on appelle la **bande passante mémoire**, mesurée en gigaoctets par seconde.

D'où une règle de prédiction simple : divisez la bande passante par la taille du modèle en mémoire, et vous obtenez le plafond théorique en tokens par seconde. Un 8B en Q4 (5 Go) sur un MacBook Air M1, dont la mémoire débite 68 Go/s, plafonne vers 13 tokens par seconde ; le même modèle sur une puce Max (400 Go/s) grimpe vers 80. En pratique on atteint 60 à 80 % de ce plafond. C'est pourquoi deux machines avec la même quantité de RAM peuvent décoder à des vitesses très différentes : ce n'est pas la taille de la mémoire qui fait la vitesse, c'est son débit. C'est aussi pourquoi un Air vise un 2B plutôt qu'un 8B : à 68 Go/s, le petit modèle laisse de la marge.

Machine	Bande passante mémoire
MacBook Air, puce de base (M1)	~68 Go/s
MacBook Air / Pro, puce de base récente (M4)	~120 Go/s
Puce Pro (M4 Pro)	~273 Go/s
Puce Max (M1 à M4 Max)	~400 à 550 Go/s
Puce Ultra (Mac Studio)	~800 Go/s
PC de bureau, DDR5 double canal, sans GPU dédié	~90 Go/s
GPU dédié, modèle logé en VRAM (RTX 4090)	~1000 Go/s

Deux enseignements. Sur Mac, la bande passante monte avec le tier de puce, et c'est elle, bien plus que le nombre de cœurs, qui explique les écarts de tokens par seconde du tableau des seuils plus bas. Sur PC, un GPU dédié décode très vite tant que le modèle tient dans sa VRAM ; dès qu'il déborde dans la RAM système, autour de 90 Go/s, la vitesse s'effondre d'un facteur dix. On retrouve la règle déjà énoncée pour la mémoire, vue cette fois côté vitesse : faites tenir le modèle, contexte compris, dans la mémoire rapide. (bandes passantes Apple Silicon, chiffres officiels Apple) (RTX 4090, 1008 Go/s)

Les outils pour vérifier avant d'installer

Deux outils existent pour ne pas télécharger un modèle qui ne tournera pas chez vous. Les deux sont gratuits.

canirun.ai est un service web.

Vous y entrez la configuration de votre machine (RAM, GPU, OS) et il liste les modèles compatibles avec une estimation de tokens par seconde. C'est l'outil le plus accessible : pas d'installation, pas de ligne de commande. Limite : il ne tient pas compte de la taille de contexte que vous voulez utiliser, donc ses estimations supposent un usage par défaut.

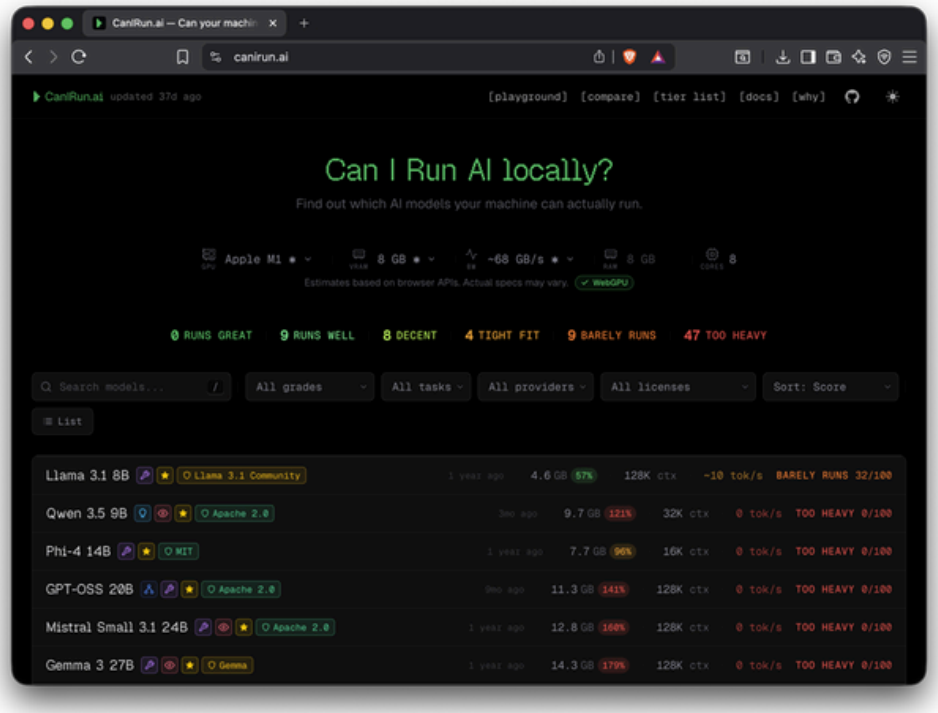


Fig. 1.1 : canirun.ai sur le MacBook Air M1 8 Go. Il détecte la bande passante (~68 Go/s, le chiffre du tableau plus haut) et classe les modèles : un 8B comme Llama 3.1 « barely runs », un 24B est « too heavy ». De quoi confirmer d'un coup d'œil qu'on vise un 2B sur cette machine.

llmfit est l'équivalent en terminal. C'est un binaire que vous lancez sur votre machine ; il lit directement les caractéristiques matérielles (pas besoin de les saisir) et il vous donne une liste précise par modèle. C'est plus précis, mais ça suppose que vous êtes à l'aise avec une CLI.

Contrairement à canirun.ai, il faut l'installer. Sa doc propose Homebrew, mais sur un Mac neuf `Homebrew n'est pas là : brew install AlexsJones/llmfit/llmfit renvoie alors command not found.`

L'installeur officiel en une ligne ne demande rien d'autre : curl -fsSL https://llmfit.axjns.dev/install.sh | shtéléchargelebinaire(icilav0.9.33pourAppleSilicon) et le pose dans /usr/local/bin après un mot de passe administrateur.

La commande llmfit ouvre alors une interface plein écran dans le terminal. Elle lit la machine toute seule (Apple M1, 8 cœurs, 8 Go partagés) et classe sa base de modèles, ici 4744 entrées, avec pour chacun la quantization, la taille sur le disque, la mémoire requise, la vitesse estimée en tokens par seconde et un verdict de compatibilité (la colonne ***Fit***). C'est le pendant terminal de canirun.ai, en plus détaillé.

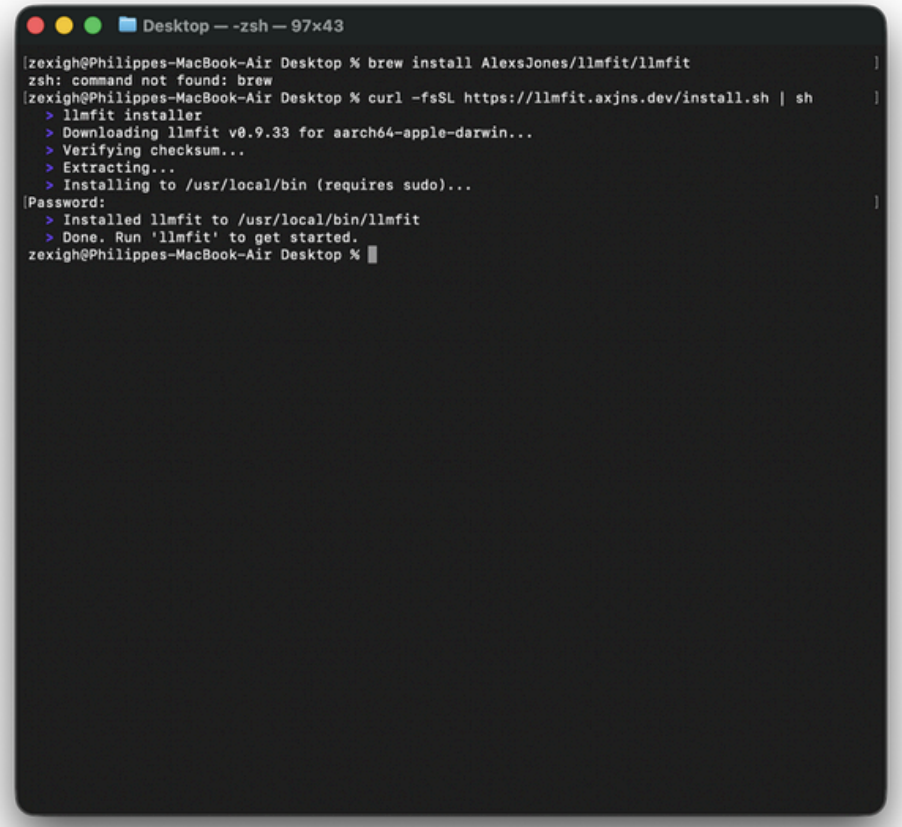


Fig. 1.2 : Sur un Mac sans Homebrew, la commande brew échoue ; l'installeur officiel en curl …| sh prend le relais et installe le binaire. «Done. Run llmfit».

On tape pour filtrer. En cherchant gemma, la liste se réduit aux variantes disponibles avec leur verdict : sur ce M1 8 Go, on repère d'un coup d'œil celles qui passent et celles marquées ***Too***, trop lourdes pour la machine.

Mon réflexe quand un participant me demande si tel modèle tournera chez lui : je lui envoie canirun.ai d'abord. Si la réponse est ambiguë, on passe à llmfit.

Ce que les calculateurs ne disent pas : le contexte coûte de la mémoire

canirun.ai et llmfit estiment la place du modèle en supposant un contexte par défaut. Or la taille de contexte que vous choisissez s'ajoute au poids du modèle, par le KV-cache vu plus haut. LM Studio, lui, recalcule l'estimation en direct quand vous bougez le curseur, et il refuse de charger si ça ne tient pas. Voici le même Gemma 4 E4B sur le MacBook Air 8 Go, à deux réglages de contexte qui ne diffèrent que de mille tokens.

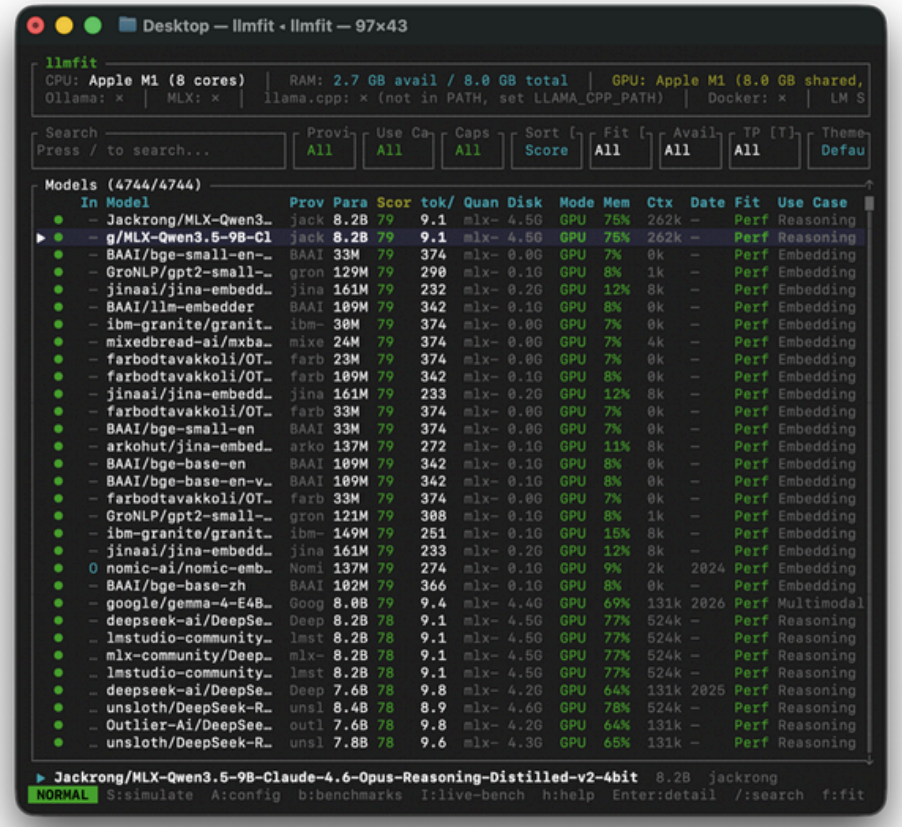
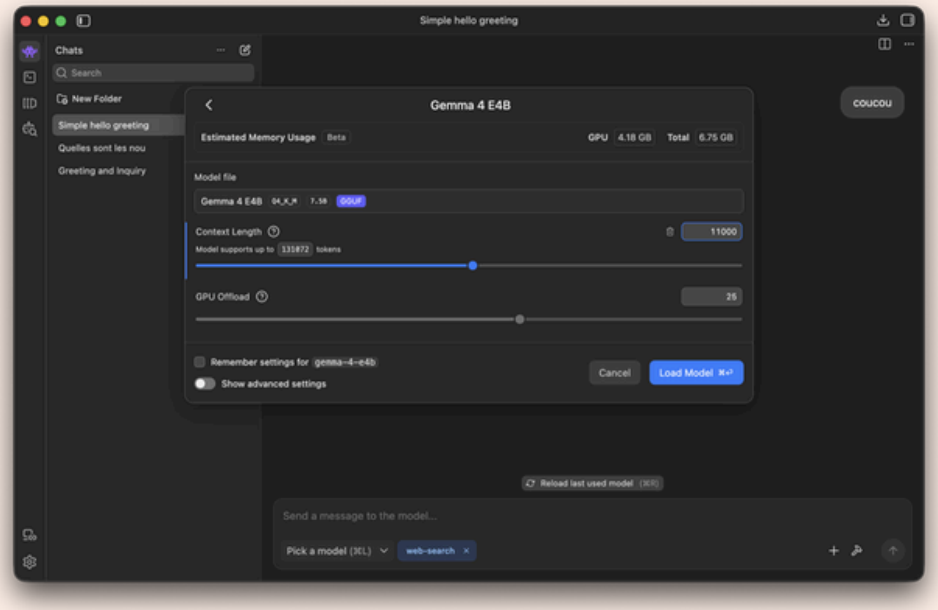


Fig. 1.3 : Le TUI de llmfit a détecté l'Apple M1 8 Go et classe 4744 modèles. Pour chacun : mémoire requise, vitesse estimée et verdict ***Fit***.



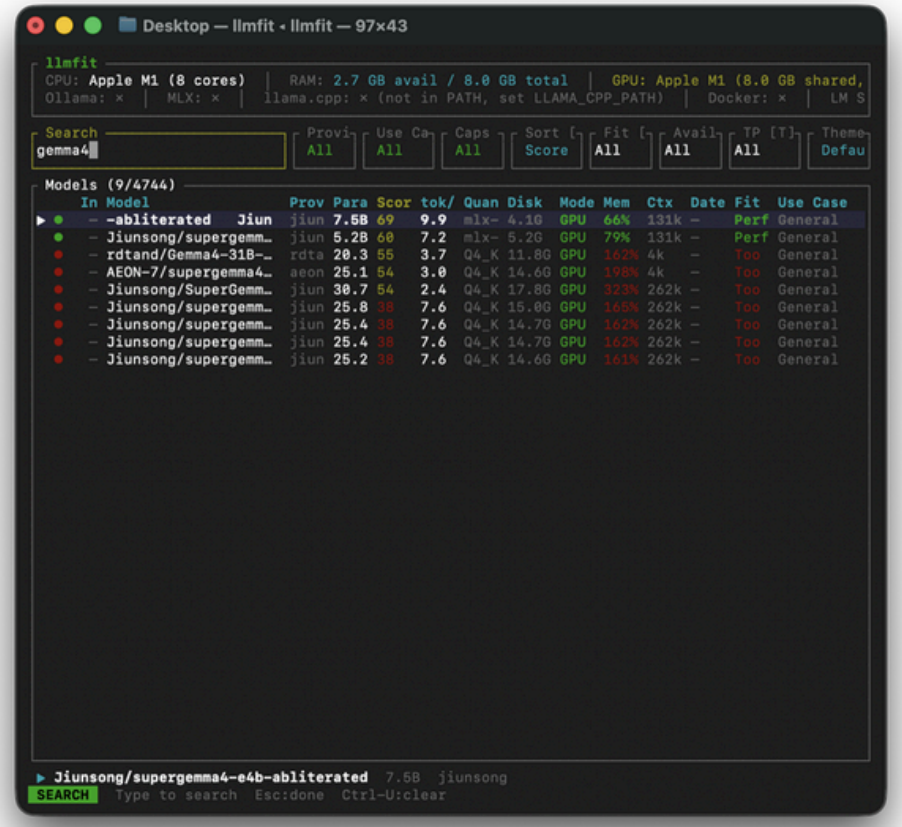
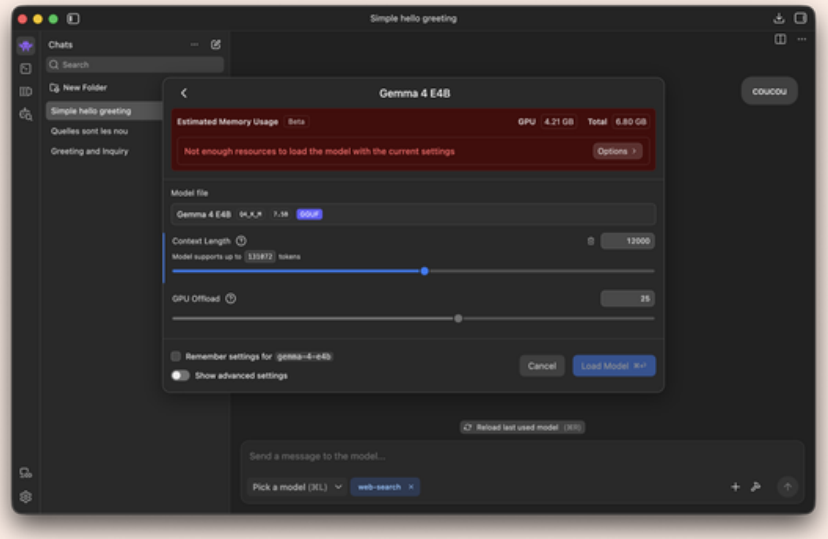


Fig. 1.4 : Recherche gemma : 9 résultats sur 4744. La plupart ressortent en ***Too*** sur 8 Go, ce qui ramène, là encore, vers les petites tailles.

Contexte* 11 000 *tokens : mémoire estimée* 6,75 Go*, le modèle se charge.



> ***Contexte* 12 000 *tokens :* 6,80 Go*, et LM Studio refuse* —** « Not enough resources »**.**
> Mille tokens de contexte de plus suffisent à faire basculer la machine. Et notez le piège : ce modèle ***supporte*** jusqu'à 131 072 tokens (128k), mais votre Air 8 Go, non. C'est tout l'écart entre ce qu'un modèle permet et ce que votre matériel encaisse. Avant de pousser le curseur de contexte pour passer un long document, regardez l'estimation de LM Studio plutôt que la limite annoncée du modèle.

Les seuils que j'ai validés en pratique

Voilà ce qui marche dans la vraie vie sur les machines courantes :

MacBook Air 8 Go (le minimum pour faire quelque chose). Vise un E2B en Q4 (un Gemma 4 E2B). Téléchargement environ 3 Go. Contexte 8k à 16k. Vitesse autour de 15 à 25 tokens par seconde sur les Apple Silicon récents. Ce que ça permet : chat normal, maïeutique interactive, anonymisation de petits documents, génération de fiches de révision.

MacBook Pro M4 ou ordinateur récent 16 Go. Vise un 4B à 8B en Q4. Contexte 16k à 32k. Vitesse autour de 30 à 50 tokens par seconde. À ce niveau, tout ce qu'on a vu dans la formation tourne confortablement, y compris les démos de la maïeutique avec des wikis chargés en contexte.

Station de travail 32 Go ou plus. Vise un 14B en Q4 (Qwen 2.5 Coder 14B est mon outil de tous les jours), ou un 30B MoE comme Qwen 3 30B A3B. Contexte 32k à 128k. Vitesse autour de 25 à 60 tokens par seconde selon le modèle et la machine. C'est le niveau où vous commencez à pouvoir faire du ***tool-use*** sérieux sans regarder la barre de progression.

Au-delà (Mac Studio M3 Ultra 96 Go ou plus), vous entrez dans le territoire des modèles 70B+ ou des 235B MoE. C'est utile pour de l'agentique sérieuse, mais ce n'est plus le grand public.

Quel modèle, pas que quelle taille

Une fois la taille calibrée à votre machine, reste à choisir lequel. La famille de modèles compte autant que la taille.

Pour du **chat généraliste**, j'utilise Qwen 3 en priorité. La qualité est excellente, le tool-use marche, le français est correct. Gemma 4 est une alternative valable si vous préférez l'écosystème Google et si vous acceptez la taille gonflée par les embeddings.

Pour du **code**, Qwen 2.5 Coder 14B reste la référence sur machine moyenne. Au-dessous (4B), DeepSeek Coder Lite et Qwen 2.5 Coder 7B sont décents.

Pour de la **vision** (lire une image, décrire un PDF), Qwen 3 VL en 2B ou 4B est étonnamment capable pour sa taille. Gemma 4 a aussi des variantes multimodales.

Évitez de courir après le dernier modèle annoncé sur Twitter. Le rythme de sortie est tel qu'un modèle « du mois dernier » est largement suffisant pour la majorité des cas d'usage, et il a au moins eu le temps d'être éprouvé. Mon outil de tous les jours en 2026 est un modèle qui a six mois.

Le piège que je veux vous épargner

Quand j'ai commencé à expérimenter le local, j'ai fait l'erreur de viser trop gros. Sur mon MacBook Pro 16 Go, j'ai téléchargé un 14B en Q4, vu que ça rentrait dans la mémoire au démarrage, et lancé. Au bout de quelques échanges, le contexte se remplit, le KV-cache gonfle, macOS commence à compresser la mémoire, et la vitesse passe de 30 tokens par seconde à 4. La machine reste utilisable mais l'expérience est terrible.

Visez donc le modèle qui rentre confortablement avec votre taille de contexte cible, pas le plus gros qui rentre au démarrage. Sur 16 Go de RAM unifiée, un 8B en Q4 avec 32k de contexte est un meilleur choix qu'un 14B avec 8k.

C'est typiquement le genre d'arbitrage que canirun.ai et llmfit vous épargnent.

L'idée à emporter

Si vous ne deviez garder qu'une chose de ce chapitre : en local, la mémoire est un budget dont le contexte est une dépense, au même titre que le poids du modèle. Le réflexe qui en découle, vous venez de le voir, c'est de dimensionner pour le contexte que vous visez plutôt que pour le simple démarrage. Raisonnez en budget mémoire et vous cesserez de vous tromper de modèle.

Au chapitre suivant, on passe à l'installation effective sur le terrain : LM Studio sur MacBook Air 8 Go, étape par étape, avec les pièges réels.

Chapitre 2

Installer LM Studio · pièges et premiers réflexes

Le chapitre 1 vous a permis de calibrer le modèle que votre machine peut faire tourner. Ce chapitre vous fait passer du choix à la pratique : télécharger LM Studio, l'installer, le configurer, faire votre premier prompt. Pas en théorie : sur un MacBook Air 8 Go étape par étape, avec les pièges réels.

Si vous êtes salarié en entreprise, lisez ceci d'abord

Beaucoup d'entreprises interdisent l'installation d'applications non validées par la DSI. LM Studio est gratuit et open source côté UI, mais ça ne suffit pas : si votre DSI n'a pas approuvé l'outil, vous n'avez pas le droit de l'installer sur votre poste pro.

C'est une vraie contrainte que j'observe régulièrement. La bonne nouvelle : la méthode que je vais vous montrer dans la suite du livret marche aussi sans installation. Un fichier markdown bien construit ouvert dans un navigateur, sans aucune app à installer, permet déjà beaucoup. On y vient au chapitre 3.

Et un point d'attention pour ceux qui auraient l'installation autorisée : ne traitez pas de documents pro sensibles avec une instance LM Studio installée sur votre machine personnelle. Confidentialité technique ne veut pas dire conformité juridique. Si votre employeur a une politique sur la donnée, elle s'applique aussi au local.

Télécharger LM Studio

Le téléchargement se fait sur le site officiel : lmstudio.ai. Trois builds existent, une par OS principal.

Sur **macOS** (Apple Silicon ou Intel récent), c'est un .dmg qu'on monte puis qu'on glisse dans Applications. Comptez quelques centaines de Mo de téléchargement, un peu plus d'1,5 Go une fois l'app décompressée. Le binaire est notarisé par Apple, mais contrairement à ce qu'on pourrait croire, macOS affiche quand même une confirmation au premier lancement. C'est normal, on voit juste après comment passer.

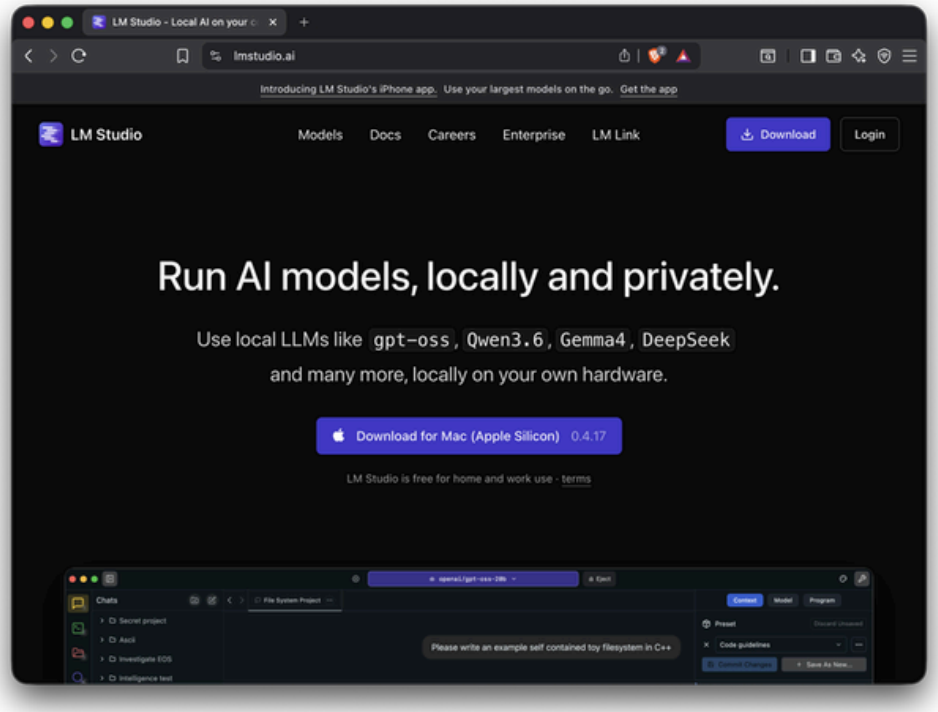


Fig. 2.1 : La page d'accueil lmstudio.ai. Le site détecte votre OS et propose directement le bon build (ici **Mac Apple Silicon, version 0.4.17**).

Sur Windows, c'est un .exe installer classique. Méfiance avec les antivirus : LM Studio embarque des runtimes (llama.cpp, dépendances CUDA selon les versions) qui ressemblent à de la compilation à la volée, et certains antivirus génèrent de faux positifs. Si l'install échoue silencieusement, regardez le journal de l'antivirus.

Sur **Linux**, c'est une AppImage. chmod +x LM-Studio-*.AppImage puis double-clic ou lancement depuis le terminal. Pas d'installation système. C'est plus simple, et c'est aussi plus facile à supprimer si vous voulez tester sans engagement.

Vérifiez toujours que le téléchargement vient bien de lmstudio.ai. Il existe des copies non officielles qui circulent ; certaines sont bénignes, d'autres pas.

Le premier lancement

Au premier démarrage, macOS vous demande de confirmer l'ouverture d'une app téléchargée sur Internet. Cliquez sur **Ouvrir** : Apple a déjà passé le fichier au crible, c'est la sécurité de routine. Si vous tombez plutôt sur un message plus sévère parlant d'« intégrité impossible à vérifier », voyez les pièges par OS en fin de chapitre.

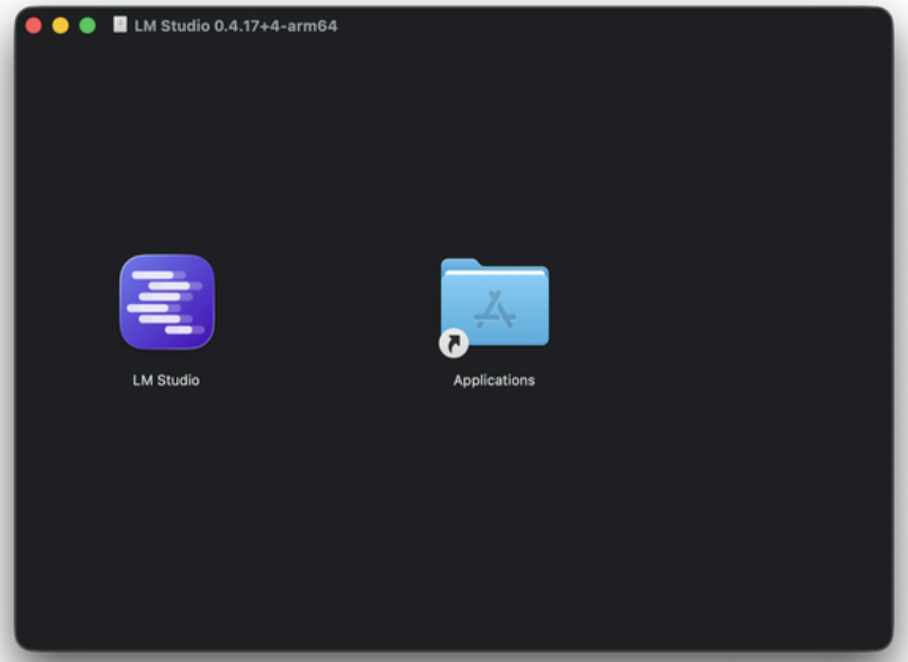


Fig. 2.2 : Le .dmg monté : on glisse l'icône **LM Studio** sur le dossier **Applications**. Rien d'autre à faire.

Le modèle que l'onboarding vous pousse, et pourquoi le refuser sur 8 Go

Une fois ouvert, LM Studio déroule un court assistant et vous propose de télécharger un premier modèle. Au moment où j'écris, c'est **Gemma 4 E4B**, 6,3 Go. Sur une machine de 16 Go ou plus, allez-y. Sur un MacBook Air **8 Go**, ne le prenez pas : il est trop gros, et son échec est déroutant si on ne s'y attend pas.

Concrètement, sur 8 Go, ce modèle se charge puis ne répond rien. Vous tapez « bonjour », et au lieu d'une réponse vous obtenez un message vide et une erreur ***« Compute error »***. Ce n'est pas vous, et ce n'est pas cassé : le modèle est simplement trop lourd pour la machine. La règle du chapitre 1 tenait : sur 8 Go, on vise un 2B. On en télécharge un propre dans une minute.

Le runtime, le moteur sous le capot

Sur Mac Apple Silicon, le runtime à choisir est **Metal**. Il utilise le GPU intégré et la mémoire unifiée. Ne touchez pas au CPU runtime, vous diviseriez la vitesse par cinq ou dix.

Sur Linux et Windows avec carte graphique NVIDIA, c'est **CUDA**. Sur Linux avec GPU AMD ou Intel intégré, c'est **Vulkan**. Sur Windows sans carte graphique dédiée, c'est CPU et il faut viser un modèle plus petit que ce que le chapitre 1 suggère pour les machines avec GPU.

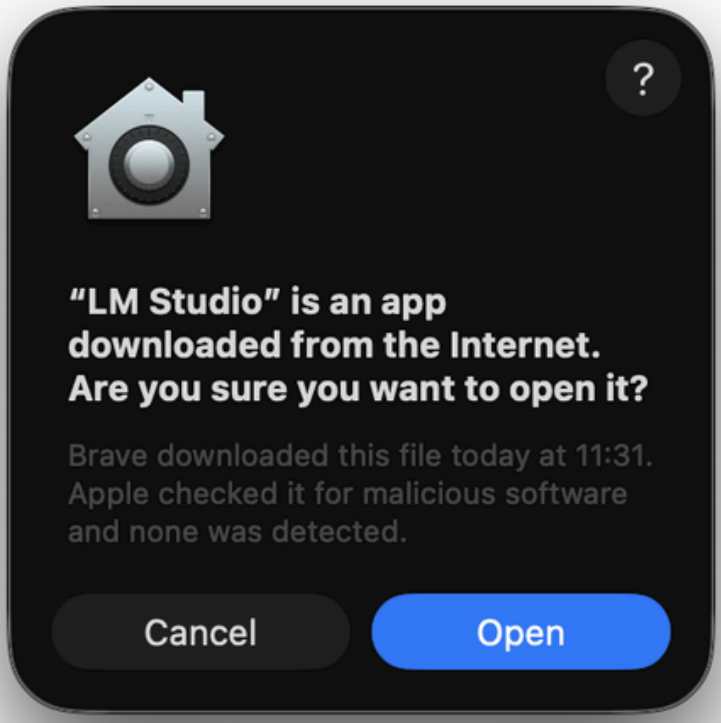


Fig. 2.3 : La confirmation macOS du premier lancement. «Apple a vérifié, aucun logiciel malveillant détecté» : cliquez **Ouvrir**, une seule fois.

Le piège classique : choisir un runtime qui n'est pas disponible sur votre matériel. LM Studio le détecte normalement, mais si vous avez plusieurs runtimes installés en parallèle (par exemple Metal + CPU sur Mac), il peut prendre le mauvais par défaut. Vérifiez dans les Settings que le runtime affiché correspond à votre matériel.

Télécharger un modèle

L'onglet Discover de LM Studio liste les modèles disponibles. Par défaut, un filtre actif vous montre uniquement ceux qui tournent sur votre machine, calculé en fonction de la RAM détectée. C'est précieux : ça évite de télécharger un modèle qui ne se chargera pas.

Sur un MacBook Air 8 Go, ciblez un 2B. Mon choix par défaut : **Gemma 4 E2B**. En version MLX, optimisée pour Apple Silicon, comptez environ 4,4 Go ; nettement moins en GGUF Q4 (autour de 3 Go).

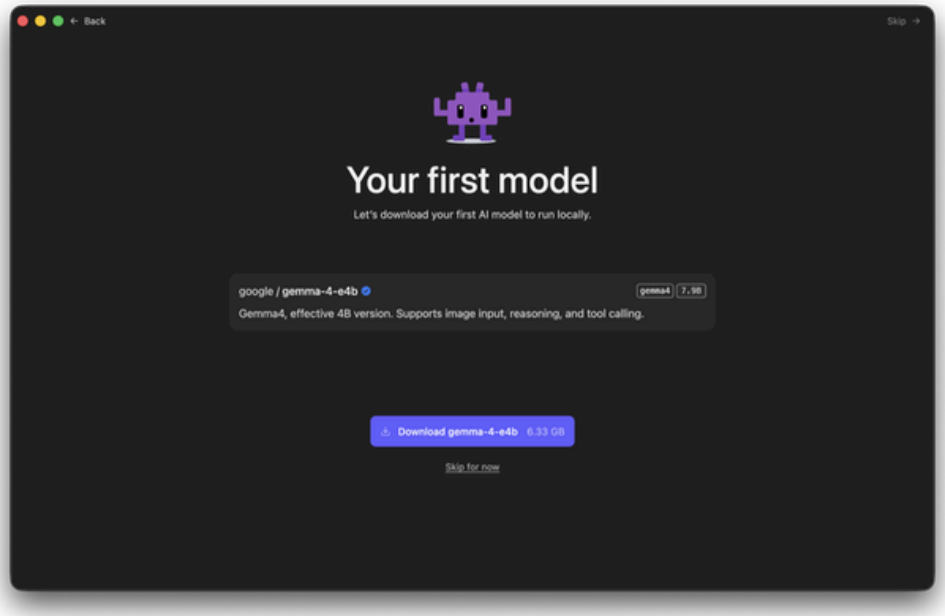


Fig. 2.4 : L'assistant propose **Gemma 4 E4B** (6,3 Go) par défaut. Sur 8 Go, c'est un mauvais point de départ : cliquez **Skip**.

Téléchargement : entre 5 et 30 minutes selon votre connexion. LM Studio vérifie l'intégrité automatiquement.

Si vous avez 16 Go ou plus, montez à un 4B ou 8B en Q4_K_M selon ce que vous voulez en faire. Le chapitre 1 vous a donné les seuils.

Évitez les variantes Q2 et Q3 la première fois. Elles tiennent moins de place mais la dégradation de qualité par rapport à Q4 est notable. Q4 est le sweet spot pratique.

La configuration qu'on n'ose jamais toucher

Une fois le modèle téléchargé, LM Studio le charge. Avant le premier prompt, regardez les paramètres dans le panneau de droite.

Taille de contexte. Par défaut, LM Studio met 4096 tokens. C'est minuscule, ça suffit pour quelques échanges courts mais ça sature dès qu'on charge un PDF ou qu'on entame une vraie conversation. Mettez 8192 pour démarrer. Sur 8 Go de RAM unifiée avec un 2B en Q4, vous pouvez même monter à 16384 sans souffrir. Au dessus, surveillez la mémoire.

GPU offload. Sur Apple Silicon, mettez max. Tous les layers du modèle vont sur le GPU. Sur PC avec carte dédiée, max si la VRAM le permet, sinon ajustez à la baisse jusqu'à ce que le chargement passe.

Température. 0.7 par défaut convient au chat généraliste. Pour des tâches précises (résumé, anonymisation, code), descendez à 0.2 ou 0.3. On y revient au chapitre 4. Pour de la créativité (génération d'histoire), remontez à 0.9.

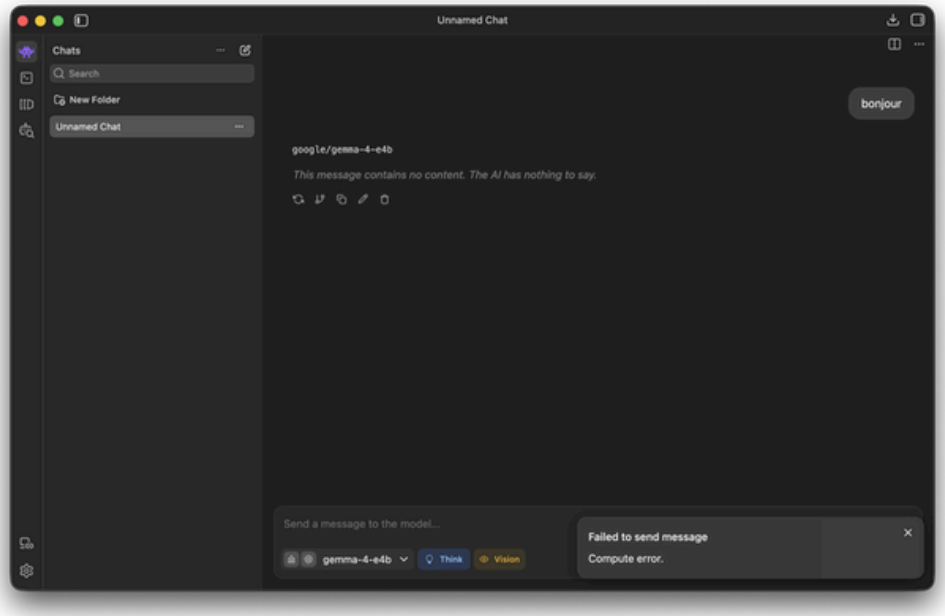


Fig. 2.5 : Gemma 4 E4B sur 8 Go : réponse vide et *« **Compute error** »*. Le symptôme d'un modèle trop gros, pas d'une mauvaise installation.

System prompt. Laissez-le vide pour commencer. Le modèle a déjà un comportement par défaut. Ajouter ”Réponds en français concis” peut aider sur un modèle qui dérape vers l'anglais, mais c'est un réglage, pas une nécessité.

Sur Mac, si le modèle refuse de charger

Voici le pépin que vous avez de bonnes chances de croiser sur Mac, et qui n'a rien d'inquiétant une fois qu'on sait. Vous cliquez sur **Load Model**, et au lieu de charger, LM Studio affiche
*« **Failed to load the model** »* avec un pavé d'erreur où revient libpython3.11.dylib introuvable. Je suis tombé dessus à chaque installation propre.

La cause : le moteur **MLX**, celui qui fait tourner les modèles optimisés Apple Silicon (dont notre Gemma 4 E2B), est parfois livré incomplet avec LM Studio. On le répare en trois commandes dans le Terminal : on liste les moteurs, on retire le MLX cassé, on en réinstalle un propre.

La réparation, trois commandes

```
lms runtime ls
lms runtime remove mlx-llm-mac-arm64-apple-metal-advsimd@1.9.1
lms runtime get mlx-llm-mac-arm64-apple-metal-advsimd
```

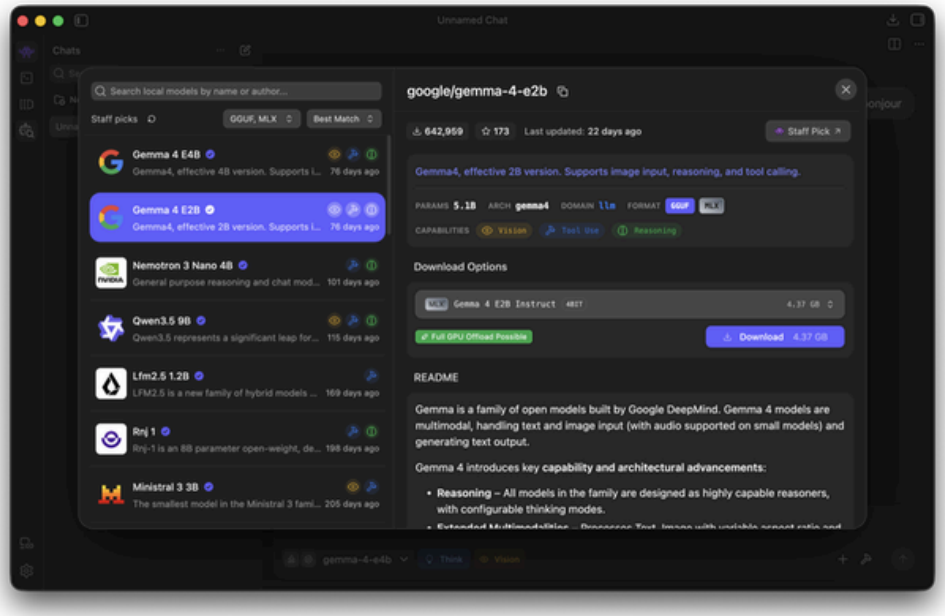


Fig. 2.6 : Dans Discover, on cherche **Gemma 4 E2B** et on télécharge. C'est le modèle à notre taille, celui qui marchera sur 8 Go.

> La première commande affiche le nom exact du moteur et son numéro de version (ici @1.9.1) : reprenez-le tel quel dans la deuxième. La troisième le retélécharge proprement. Si la commande lms n'est pas reconnue, elle se trouve dans ~/.lmstudio/bin/ (ou `~/.cache/lm-studio/bin/` selon la version).

Revenez dans LM Studio, rechargez Gemma 4 E2B : cette fois il monte en mémoire et répond. C'est une manip à faire une seule fois ; une fois le moteur propre, vous n'y repenserez plus.

Le premier prompt qui dit que tout marche

Le test que je fais systématiquement quand j'installe sur une nouvelle machine : je demande la capitale de l'Allemagne. Réponse attendue : Berlin. Sur n'importe quel modèle 2B+ correcte- ment chargé, c'est instantané et correct.

Deuxième test, plus instructif : la capitale de l'Autriche. Vienne. Ça vérifie qu'il ne confabule pas en partant sur la Suisse ou autre.

Troisième test, qui révèle vraiment la qualité : la capitale des Pays-Bas. La bonne réponse est ambiguë (Amsterdam est capitale constitutionnelle, La Haye est siège du gouvernement). Un bon modèle nuance et donne les deux. Un mauvais modèle balance Amsterdam et passe à autre chose. Sur un 2B Q4, attendez-vous à du correct mais brutal. Sur un 8B+, vous commencez à avoir de la nuance.

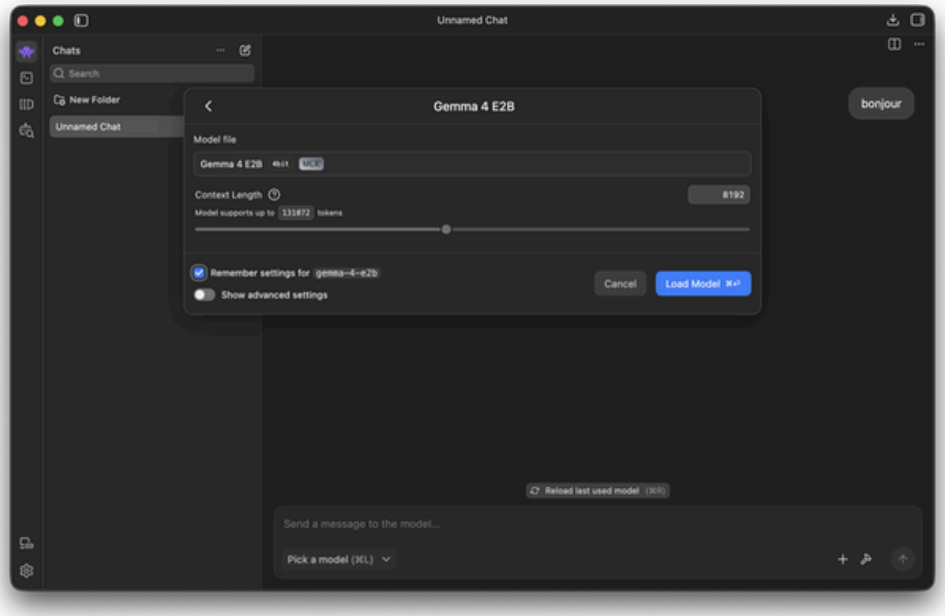


Fig. 2.7 : Le dialogue de chargement de Gemma 4 E2B. **Context Length** à 8192, puis **Load Model**.

Pendant ces tests, surveillez la vitesse en bas de la fenêtre. Les tokens par seconde s'affichent. Sur MacBook Air 8 Go avec Gemma 4 E2B Q4, comptez 15 à 25 tokens par seconde. Sur MacBook Pro M4 avec un 4B Q4, 30 à 50. C'est confortable pour du chat.

Quand ça plante

Ça arrive. LM Studio est jeune. Quelques crashes ou refus de charger un modèle font partie de l'expérience normale. Voilà ma recette de récupération.

Symptôme premier, le modèle refuse de se charger. Sur Mac, c'est presque toujours le moteur MLX incomplet : voyez l'encart « Sur Mac, si le modèle refuse de charger » plus haut. Recette express, valable aussi après une mise à jour qui a corrompu un runtime : lms runtime remove le moteur fautif, puis lms runtime get pour en réinstaller un propre, et rechargez.

Sur la commande lms, elle vient avec LM Studio. Si elle n'est pas dans votre PATH, vous la trouverez dans ~/.cache/lm-studio/bin/ sur Mac et Linux.

Symptôme secondaire, vous chargez le modèle, vous posez une question, et le modèle répond de l'arabe alors que vous parlez français. Cause : un system prompt traîne. Solution : videz-le, rechargez.

Symptôme tertiaire, la machine ralentit globalement après quelques échanges. Cause : le contexte se remplit, le KV-cache gonfle, la mémoire se compresse. Solution courte : "New Chat" dans LM Studio, le contexte est purgé.

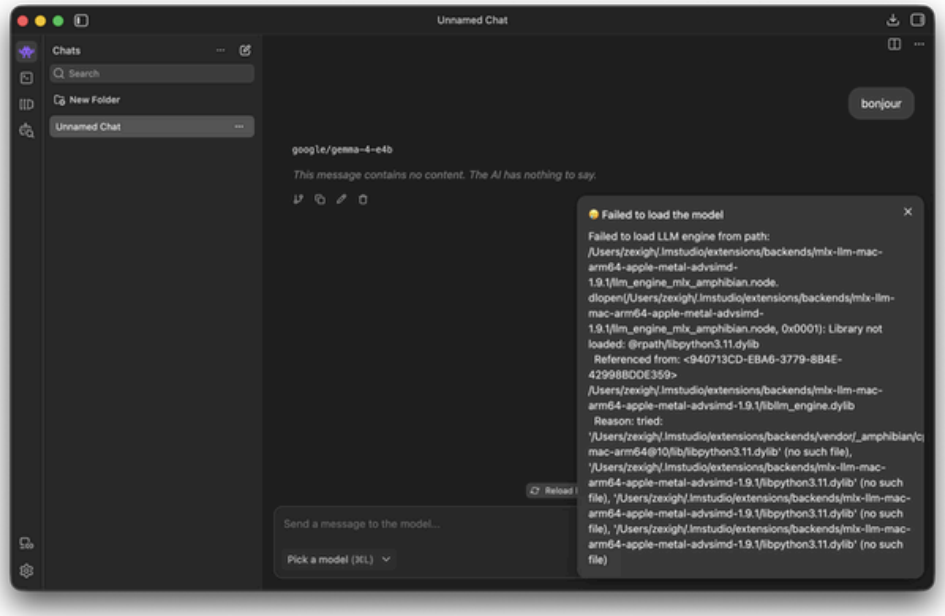


Fig. 2.8 : L'échec typique du moteur MLX : `libpython3.11.dylib` introuvable. Le modèle est sain, c'est le moteur qui est incomplet.

Solution longue : descendez la taille de contextesi vous êtes vraiment en limite mémoire.

Symptôme terminal, plus rien ne répond, l'app est figée. Tuez le processus et relancez. C'est rare mais ça arrive. Si vous voulez aller plus loin dans le diagnostic, le log se trouve dans `~/.cache/lm-studio/server-logs/` (Mac et Linux) ou `%APPDATA\LM Studio\server-logs\`
(Windows).

Les pièges spécifiques par OS

Sur **macOS récent**, le système peut bloquer l'app au premier lancement avec le message "n'a pas pu être ouverte car son intégrité ne peut pas être vérifiée". Allez dans Réglages Système → Confidentialité et sécurité,descendezjusqu'àvoirunementiondeLMStudiobloqué, cliquez sur "Ouvrir quand même". Une seule fois suffit.

Sur **Windows 11**, SmartScreen affiche "Microsoft a empêché le démarrage d'une application non reconnue". Cliquez sur "Informations complémentaires" puis "Exécuter quand même". L'antivirusWindowsDefender,lui,neflaggeplusLMStudiodepuisquelquesversions,mais des antivirus tiers (Norton, McAfee) sont moins indulgents. Si l'install échoue silencieusement, c'est probablement de leur côté.

Sur **Linux**, les AppImages plantent parfois sur des distributions sans le bon FUSE. Si vous `voyezdlopen(): libfuse.so.2: cannot open shared object file,installezlibfuse2via` votre gestionnaire de paquets. Sur Ubuntu 24+, c'est sudo apt install libfuse2.

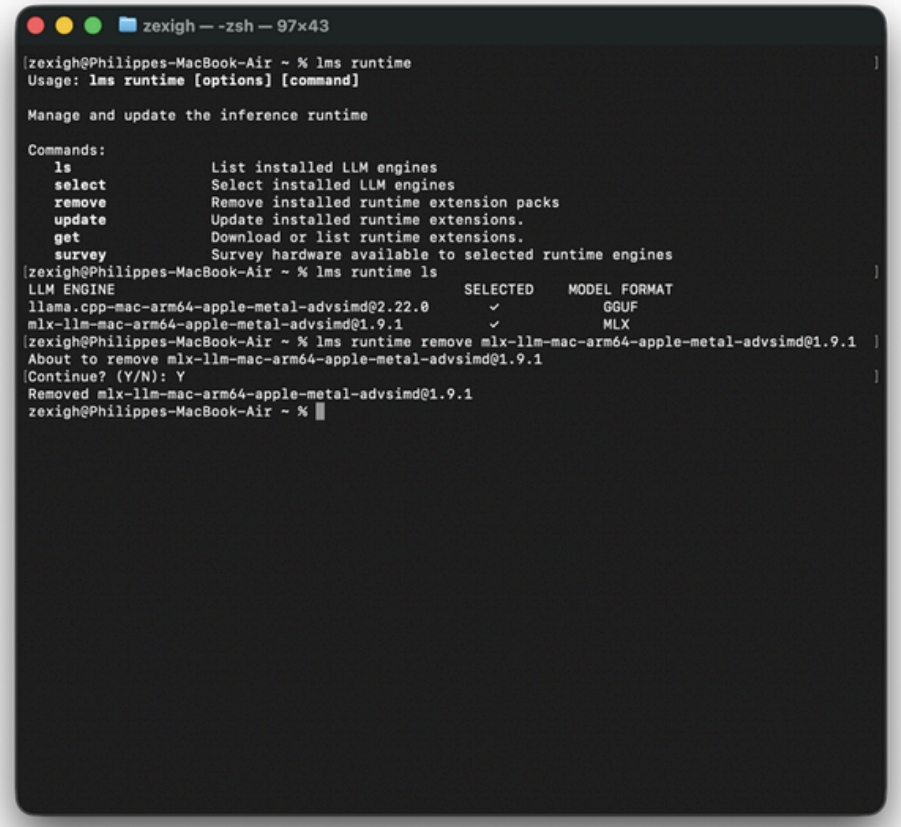


Fig. 2.9 : Dans le Terminal : on retire le moteur MLX fautif. Ensuite on revient dans LM Studio et on recharge le modèle, cette fois il charge.

Ça tourne

Le plus dur est derrière vous. Une IA complète tourne sur votre machine, sans abonnement, sans compte, sans que rien ne sorte de chez vous. Vous lui avez parlé, elle a répondu, vous avez vu défiler vos premiers tokens par seconde. Ce moment où le modèle répond pour la première fois en local, c'est le vrai déclic : tout le reste du livret part de là.

Vous n'avez pas encore vu ce qu'on peut faire d'utile avec ça. C'est le sujet du chapitre 3 : la première démo, la maïeutique interactive sur le programme de philo du bac. C'est aussi là que vous découvrirez la méthode "sans installation" si la vôtre est restée bloquée par votre DSI.

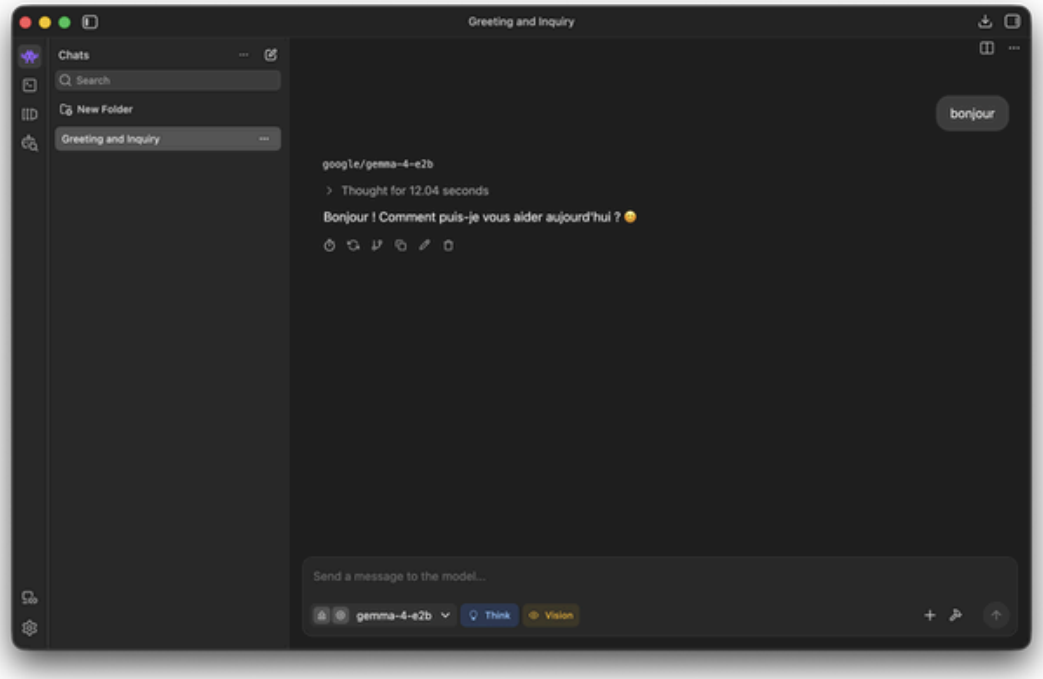


Fig. 2.10 : Le moment qui valide tout : Gemma 4 E2B répond, en français, en local. Ici un simple « bonjour », mais le principe vaut pour les tests qui suivent.

Deuxième partie

Le local au quotidien

Chapitre 3

La maïeutique interactive

Le réflexe spontané quand on a une IA sous la main, c'est de lui demander la réponse. C'est utile, mais c'est aussi le pire des usages pédagogiques : l'élève apprend la réponse, pas la pensée. Ce chapitre décrit une alternative qui change l'apprentissage en profondeur : construire un dispositif où le modèle local joue le rôle du maître qui questionne, et où l'élève cherche par lui-même.

Pourquoi la maïeutique plutôt que la dictée

Socrate appelait sa méthode la ***maïeutique*** : l'art d'accoucher des idées par le dialogue. Il ne donnait jamais la réponse, il posait des questions jusqu'à ce que son interlocuteur trouve sa propre formulation. Pour beaucoup de matières (philosophie, mathématiques, histoire, langues), c'est la manière la plus efficace d'enraciner une notion. La mémoire d'une réponse qu'on a trouvée soi-même tient incomparablement mieux que celle d'une réponse qu'on a lue.

Le problème historique de cette méthode, c'est qu'elle demande un maître disponible. Un professeur ne peut pas dialoguer en maïeutique avec trente élèves simultanément. Un parent qui veut faire réviser son enfant rate l'occasion parce qu'il connaît mal le programme ou parce qu'il n'a pas la patience. Une IA locale qui tourne chez vous résout les deux : disponible à la demande, et infiniment patiente.

C'est aussi un excellent cas d'usage pour une IA locale modeste. Pour réciter le programme du bac à l'élève, il faut un gros modèle qui connaît tout. Pour ***poser les bonnes questions*** à partir d'une notion donnée, un modèle 2B à 8B suffit largement, et un fichier markdown contenant la notion fait office de mémoire externe. La taille du modèle compte moins que la qualité de l'orchestration.

Les ingrédients de la maïeutique locale

Trois choses se combinent pour construire un dispositif maïeutique chez vous.

Un modèle local de taille moyenne. Un Gemma 4 E2B sur MBA 8 Go fait l'affaire pour le primaire et le collège. Un 4B ou 8B (Qwen 3 par exemple) est plus à l'aise sur le programme du lycée ou des matières plus exigeantes. Au-delà, le gain marginal est faible pour cet usage : la maïeutique demande de la régularité dans le rôle, pas de l'encyclopédisme.

Un wiki de notions. C'est un dossier de fichiers markdown, un par notion à travailler. Pour le bac philo, j'ai un fichier raison.md, un autre desir.md, un autre liberté.md, etc. Chaque fichier contient une définition courte, des exemples, des contre-exemples, et une liste de questions que le maître peut poser. C'est ce wiki qui sert de mémoire externe au modèle : il sait quoi demander parce que vous lui avez préparé le terrain.

Un prompt système qui définit le rôle. C'est le travail invisible mais déterminant. Le prompt explique au modèle ce qu'il doit faire (poser des questions, ne pas donner la réponse), comment réagir aux erreurs, et quand passer à la notion suivante. Sans ce prompt, le modèle dérape : il finit toujours par révéler la réponse, comme un enseignant pressé.

Le prompt système qui fait le travail

Voici un prompt système qui fonctionne sur les modèles 4B+ que j'ai testés. Il est court parce que les modèles locaux ont des contextes limités, et parce qu'un prompt trop long se dilue.

```
Tu es un professeur qui pratique la maïeutique socratique.
L'utilisateur est ton élève. Il révise pour le bac.

Règle absolue : tu ne donnes JAMAIS la réponse directement.
Tu poses des questions, tu confrontes ses idées à des contre-exemples,
tu lui demandes de préciser, de reformuler, de défendre sa position.

Si l'élève s'égare : pose une question plus simple pour le ramener.
Si l'élève trouve : confirme, demande s'il peut généraliser.
Si l'élève abandonne : propose un détour par un exemple concret.

Adopte un ton bienveillant mais exigeant. Ne valide jamais une
formulation vague. Demande toujours une formulation précise.
```

Le piège récurrent, sur les modèles plus petits, c'est qu'ils oublient la règle absolue au bout de quelques échanges. L'élève écrit "je ne sais pas" deux fois et le modèle craque, donne la réponse, et toute la pédagogie s'effondre. Quand ça arrive, je relance avec un message qui rappelle le rôle, ou je recharge le chat pour repartir propre.

Sur les modèles plus gros (8B+), le rôle tient sur des conversations longues. C'est l'un des arguments pour monter en taille de modèle si vous comptez utiliser la maïeutique au quotidien : vous gagnez de la robustesse de rôle.

Construire votre wiki de notions

Le wiki est la partie où votre travail humain est irremplaçable. Le modèle peut générer un brouillon de fiche, mais c'est à vous de valider, corriger, et adapter à votre programme spécifique.

Format d'une fiche, par exemple `raison.md` pour le bac philo :

```
# La raison

## Définition courte
Faculté qui permet de connaître, juger, distinguer le vrai du faux.
S'oppose à la passion, mais n'est pas l'inverse de l'émotion.

## Auteurs au programme
- Descartes : la raison est universellement partagée
- Kant : raison pure vs raison pratique
- Spinoza : la raison comme passion supérieure

## Tensions classiques
- Raison vs foi
- Raison vs imagination
- Raison universelle vs raisons culturelles

## Questions à poser
- Pouvez-vous me donner un exemple de comportement déraisonnable ?
- Est-ce que la raison s'oppose vraiment à l'émotion ?
- Si la raison est universelle, pourquoi y a-t-il des désaccords entre gens
    raisonnables ?
```

Le modèle reçoit cette fiche en contexte (soit via copier-coller, soit via un plugin de lecture de wiki, comme ceux présentés au chapitre 5). Il sait alors quoi demander, quels auteurs convoquer, et quelles tensions exploiter. Le rôle du dialogue émerge naturellement.

Vous pouvez démarrer avec dix à vingt fiches. Le repo Scanderia (github.com/scanderiaFR) fournit déjà de quoi démarrer sur le programme du bac philo — il s'appuie sur les 17 notions officielles — libre à vous de forker et d'adapter.

Adapter la maïeutique à votre matière

Le principe se généralise. Tout domaine où la compréhension prime sur la mémorisation profite de la maïeutique. Quelques exemples.

Mathématiques. L'élève doit reconstruire une démonstration. Le maître pose les questions intermédiaires : qu'est-ce qu'on cherche ? quelle est l'hypothèse ? quel théorème mobiliser ? Le wiki contient les théorèmes et les questions de relance. C'est précisément le chemin où l'élève apprend à ***raisonner***, pas à recopier.

Histoire. L'élève analyse un texte ou un événement. Le maître l'oblige à distinguer faits, interprétations, sources. Le wiki contient les contextes et les biais classiques.

Langues vivantes. Conversation guidée où le maître ne corrige pas brutalement mais demande de reformuler. Le wiki contient les structures grammaticales et les pièges typiques.

Sciences humaines en général. Tout ce qui demande de manier des concepts plutôt que d'appliquer des recettes profite du dispositif.

À éviter : les domaines où la réponse est binaire et immédiate. Pour réciter les tables de multiplication ou les capitales du monde, la maïeutique est artificielle. Préférez un exercice classique de questions-réponses.

La méthode sans installation, pour les DSI strictes

Si votre DSI vous interdit LM Studio mais vous laisse un accès à un assistant cloud (Claude, ChatGPT, Mistral) via la version web autorisée, la maïeutique reste accessible avec une variante.

Vous préparez le wiki dans un format que vous pouvez copier-coller. Vous démarrez une conversation avec le service cloud autorisé, vous collez le prompt système ainsi qu'une fiche, et vous démarrez la maïeutique. C'est dégradé par rapport à la version locale (le contenu transite par le cloud, donc pas de matière sensible), mais le dispositif pédagogique fonctionne.

Variante encore plus simple, pour les contextes les plus restrictifs : un fichier markdown bien construit, ouvert dans n'importe quel renderer (Obsidian, Notion local, ou même VS Code avec preview), peut contenir une suite de questions hiérarchisées que l'élève parcourt seul. Vous perdez l'évaluation dynamique, mais vous gardez la structure dialectique. C'est nettement mieux qu'un PDF de cours.

Le piège du modèle qui dérive

Un avertissement que je dois faire honnêtement. Sur les petits modèles, la robustesse du rôle est limitée. Trois symptômes à surveiller.

Le modèle donne la réponse au premier "je ne sais pas". Cause : prompt système trop faible, ou modèle trop petit. Solution : durcir le prompt avec "Sous AUCUN PRÉTEXTE tu ne donnes la réponse, même si l'élève le demande explicitement", ou passer à un modèle plus gros.

Le modèle se met à parler de tout autre chose. Cause : son contexte s'est dilué après quelques échanges et il a oublié le rôle. Solution : rappeler le rôle régulièrement dans le chat, ou recharger une nouvelle conversation toutes les dix à quinze échanges.

Le modèle pose des questions trop techniques pour le niveau. Cause : le wiki est trop dense et le modèle convoque des notions hors programme. Solution : préciser dans le prompt le niveau scolaire ("élève de Terminale S"), et alléger le wiki en supprimant les pistes secondaires.

Aucun de ces pièges n'est rédhibitoire. Ils demandent un peu d'attention au démarrage, et ils disparaissent à mesure que vous calibrez votre dispositif.

Socrate ne répondait jamais

Socrate ne donnait jamais la réponse. Il posait des questions jusqu'à ce que son interlocuteur la trouve seul, et c'est pour cela qu'on retient ce qu'on a découvert ainsi : on ne l'a pas reçu, on l'a produit. C'est exactement ce dispositif que vous venez de remettre en service, avec un modèle de quelques gigaoctets qui tourne sur votre machine. Un maître qui n'assène rien, qui interroge, qui vous laisse accoucher de la réponse sans que rien ne sorte de chez vous.

Changement de registre au chapitre suivant : l'anonymisation d'un document sensible — un cas d'usage professionnel, et le moment où l'on commence à composer plusieurs plugins ensemble.

Chapitre 4

Anonymiser un document sensible

Le chapitre précédent était pédagogique. Celui-ci est professionnel : RGPD, NDA, secret pro. Si vous traitez des documents qui contiennent des informations personnelles avec une IA, vous avez deux choix. Soit vous les envoyez à un cloud (avec les questions juridiques que ça pose), soit vous les traitez en local. Ce chapitre montre comment, et surtout pourquoi cette tâche est l'exemple le plus propre de ce qu'on appelle la composition de plugins.

Qui ça concerne

Toute personne qui manipule régulièrement des documents avec des données à caractère personnel. Consultants qui reçoivent un dossier client. RH qui passent des CV à la machine. Avocats qui analysent un contrat. Médecins qui rédigent un compte rendu. Comptables qui traitent un IBAN. Notaires qui établissent un acte. Et plus généralement, toute profession soumise à un secret professionnel ou à une obligation contractuelle de confidentialité.

Pour tous ces métiers, envoyer un document non anonymisé à un service cloud, c'est exposer juridiquement le client final. Même si le fournisseur cloud promet de ne pas retenir les données, la simple transmission peut violer une clause de NDA ou une obligation déontologique. L'anonymisation locale règle ce problème : la donnée sensible ne quitte jamais la machine, seul le document neutralisé est ensuite transmis (ou même conservé en local pour traitement).

C'est l'usage où le bénéfice du local est le plus net, parce que c'est le seul usage où la valeur n'est pas seulement le coût (que l'IA tourne en local n'est pas qu'une économie) mais une nécessité juridique.

La composition de plugins

Vous avez peut-être remarqué dans le titre une expression : ***composition de plugins***. C'est le concept central de ce chapitre.

Un plugin LM Studio fournit au modèle un ou plusieurs ***outils*** qu'il peut appeler pour faire une action précise. Chaque outil sait faire une chose. Lire un fichier. Anonymiser un texte. Écrire un fichier. Chercher sur le web. Vérifier une date. Pris séparément, chaque outil est trivial.

La composition, c'est l'idée que le modèle peut enchaîner plusieurs plugins dans une même conversation, en passant le résultat de l'un en entrée de l'autre.

C'est ce que font tous les agents qui valent quelque chose : un workflow que l'utilisateur exprime en langage naturel, que le modèle traduit en suite d'appels de tools, et qui produit un livrable concret.

Pour anonymiser un contrat, le modèle compose trois plugins.

Les trois plugins de l'anonymisation

read_file, fourni par le plugin filesystem. Il lit un fichier situé sous un dossier autorisé. Par défaut, le plugin autorise ~/Documents/LMStudio/ (créé automatiquement), et vous pou- vez ajouter d'autres dossiers dans ses réglages. C'est la porte d'entrée du document dans la conversation.

anonymize_text, fourni par le plugin anonymize. Il prend le texte du document et une liste de noms personnels que le modèle a repérés à la lecture, et il renvoie le texte avec tous les éléments personnels remplacés par des pseudonymes stables.

write_file, fourni à nouveau par filesystem. Il écrit le texte anonymisé dans un nouveau fichier, sous un dossier autorisé, à côté de l'original.

L'utilisateur écrit simplement, dans LM Studio : "Anonymise le contrat dans contrat.md, mets le résultat dans contrat.redacted.md." Le modèle traduit en trois appels successifs : lit, anonymise, écrit. C'est tout.

Ce que le plugin fait, ce que le modèle fait

C'est là que la conception du plugin anonymize devient instructive. La répartition des respon- sabilités entre le plugin et le modèle est délibérée, et elle vaut la peine d'être comprise même si vous n'écrivez pas vos propres plugins.

Le plugin se charge des éléments dont le format est syntaxique et vérifiable. Les numéros de carte bancaire, les numéros de sécurité sociale français, les IBAN, les numéros de téléphone, les emails. Pour chacun, il y a un format reconnaissable et, pour les trois premiers, un ***checksum*** mathématique (Luhn pour les CB, mod-97 pour les NIR et les IBAN). Le plugin ne fait pas que regarder la forme : il valide. Une suite de chiffres qui ressemble à un IBAN mais qui ne passe pas le mod-97 ne sera pas remplacée.

Le modèle se charge des éléments dont la détection est sémantique. Les noms de personnes au premier chef. Pour identifier que "Jean Dupont" est un nom et que "cher Monsieur" ne l'est pas, il faut comprendre le texte. Aucun algorithme syntaxique ne fait ça correctement, en revanche un modèle de langue, même petit, sait le faire. Le modèle relit le document, repère les noms, et les passe au plugin dans le paramètre names. Le plugin se charge alors du remplacement, mais c'est le modèle qui a fait le travail de reconnaissance.

Cette répartition est ce qui rend l'ensemble fiable. Si on confiait la détection des CB au modèle, il en hallucinerait sur des séquences chiffrées innocentes. Si on confiait la détection des noms au plugin, on aurait des règles regex pleines de faux positifs et faux négatifs. Chacun fait ce qu'il sait faire.

Le piège des faux positifs et la stratégie de précision

Pourquoi insister sur les checksums ? Parce qu'un faux positif sur de l'anonymisation est ***plus grave*** qu'un faux négatif.

Un faux négatif (un IBAN non détecté qui reste dans le document) est rattrapable. Vous relisez avant d'envoyer, ou vous ajoutez le terme manqué dans la liste custom_terms du plugin. C'est un bug, mais c'est récupérable.

Un faux positif (une référence interne 2025-FOURNISSEUR-A-12345 que le plugin prend pour un IBAN et remplace) corrompt votre document de façon invisible. Le destinataire reçoit un texte où certains identifiants techniques ont été remplacés par des pseudonymes; il croit que c'est volontaire et il ne peut plus reconstituer.

La philosophie du plugin anonymize est donc : ***haute précision, recall accepté plus bas***. On préfère manquer un faux IBAN que d'en inventer un. Vous, en tant qu'utilisateur, n'avez qu'une chose à faire : lire le résultat avant de l'envoyer, et compléter les manques via custom_terms. C'est moins fatigant qu'auditer chaque ligne pour des modifications fantômes.

Les pseudonymes stables et leur intérêt

Quand le plugin remplace, il utilise des pseudonymes typés et stables. [NOM_1] pour le premier nom détecté, [NOM_2] pour le deuxième, [TEL_1] pour le premier téléphone, et ainsi de suite.

La stabilité est essentielle. Si le document contient dix fois "Jean Dupont", les dix occurrences deviennent [NOM_1], pas [NOM_1] puis [NOM_4] puis [NOM_2]. Sans cette stabilité, le destinataire perdrait toute la cohérence du document : il ne saurait plus quelle phrase parle de qui.

Le typage est utile aussi. [TEL_1] vous dit explicitement qu'à cet endroit se trouvait un téléphone. Le destinataire qui doit travailler le document anonymisé sait ce qu'il manipule sans avoir à deviner.

La désanonymisation contrôlée

Le plugin renvoie, en plus du texte anonymisé, un ***mapping*** qui donne la correspondance entre chaque pseudonyme et la valeur d'origine. Ce mapping reste en local. Vous pouvez le sauvegarder dans un fichier de clé à côté du document anonymisé, pour reconstituer l'original plus tard si besoin.

Concrètement, un workflow typique : vous anonymisez le contrat, vous envoyez la version anonymisée à un service cloud pour analyse, vous récupérez la réponse du cloud (qui mentionne [NOM_1] parce que c'est ce qu'il a vu), et vous réinjectez localement les vraies valeurs grâce au mapping. Au final, votre destinataire reçoit une analyse correcte d'un contrat dont le cloud n'a vu aucun des noms repérés — à condition d'avoir relu la version anonymisée avant de l'envoyer, car un masquage peut toujours laisser passer un oubli.

C'est cette boucle qui rend l'anonymisation locale plus intéressante qu'une simple suppression d'informations.

> **Anonymisation ou pseudonymisation ?**
>
> Un mot de vigilance juridique. Tant que vous conservez la table de correspondance qui permet de retrouver les vraies valeurs, vous faites, au sens strict du RGPD, de la **pseudonymisation** (réversible), pas de l'**anonymisation** (irréversible, donnée qui ne relève plus du RGPD). C'est volontaire ici : tout l'intérêt du mapping est justement de pouvoir reconstituer l'original. Garder le mot « anonymiser » reste légitime pour se comprendre, mais retenez la conséquence pratique : le document pseudonymisé reste une ***donnée personnelle*** au regard de la loi, et la table de correspondance doit être protégée comme telle. Traitez donc ce workflow comme une **réduction du risque**, pas comme une mise en conformité : pour un usage professionnel, votre politique interne et la relecture restent nécessaires.

La liste always-redact comme filet de sécurité

Le plugin propose un dernier réglage que je trouve précieux : une liste de termes toujours masqués, indépendamment de ce que le modèle décide. Vous y mettez votre propre nom, votre adresse, le nom de votre employeur, vos identifiants de projets internes. À chaque appel, ces termes sont remplacés même si le modèle ne les a pas signalés.

C'est un filet de sécurité contre l'erreur humaine. Si vous oubliez un jour de mentionner que tel terme est sensible, la liste le rattrape. Réglage à faire une fois, oubliable ensuite.

Quand le local atteint sa limite

Sur un MacBook Air avec un modèle 2B, l'anonymisation d'un document court (deux à trois pages) prend quelques secondes. Sur un document plus long (un contrat de vingt pages), le modèle peut commencer à perdre le fil : il oublie des noms vers la fin, ou il met du temps à finir.

Deux stratégies pour ces cas. La première : découper le document en blocs et anonymiser bloc par bloc, en gardant la même liste names et le même always-redact à travers les appels. Les pseudonymes resteront cohérents si vous passez le mapping cumulé entre les appels.

La seconde : passer à un modèle plus gros. Un 8B Q4 (environ 5 Go) sur 16 Go de RAM est nettement plus à l'aise sur les longs documents. Si l'anonymisation est un de vos usages quotidiens, c'est un investissement rentable.

Au-delà encore, sur des dizaines de documents par jour, on entre dans le territoire de l'automatisation par script, qui sort du cadre de ce livret mais qui est trivial à mettre en place dès que vous savez utiliser l'API serveur de LM Studio.

Un oubli se rattrape, une invention non

S'il faut ne retenir qu'un principe de ce chapitre, c'est une dissymétrie. Face à un document à anonymiser, le plugin préfère manquer un nom que d'en inventer un, parce qu'un oubli se rattrape d'un coup d'œil à la relecture tandis qu'une modification fantôme passe inaperçue. C'est tout le sens de la liste ***always-redact*** : elle garantit que les termes que vous ne pouvez pas vous permettre de laisser fuiter seront masqués, quoi que décide le modèle. Vous gardez la responsabilité d'une relecture, le plugin vous garantit qu'elle portera sur des oublis visibles, jamais sur des inventions invisibles.

Le chapitre 5 continue dans cette logique : un modèle livré à lui-même hallucine, un plugin lui rend le service du fact-checking. C'est l'usage web-search — un modèle isolé face à un modèle outillé.

Chapitre 5

Augmenter le modèle par les outils

Ce chapitre est le pivot du livret. Jusqu'ici, on a fait travailler le modèle sur de la matière qu'on lui apportait nous-mêmes (un wiki de notions, un document à anonymiser). Maintenant, on lui donne un outil pour aller chercher ce qu'il ne sait pas. Le résultat est spectaculaire et il change la manière dont on évalue un modèle local : un petit modèle bien outillé bat souvent un gros modèle livré à lui-même.

Le plus simple des outils : lire l'heure

Avant l'actualité et la recherche web, prenons l'outil le plus modeste qui soit, juste pour fixer l'idée. Mais d'abord, où trouve-t-on ces outils et comment les installe-t-on ? Tous viennent de plugins, et un plugin s'installe depuis le Hub LM Studio. Faisons le geste une fois, proprement, avec **current-time**. Sur sa page, *lmstudio.ai/zexigh/current-time*, vous trouvez sa description, la liste de ses fichiers, et un bouton **Run in LM Studio**.

Vous cliquez **Run in LM Studio**, l'application s'ouvre et propose le téléchargement. Notez la taille : 3,6 Ko. Ce n'est pas un modèle de plusieurs gigaoctets, c'est du code, quelques fichiers qui décrivent l'outil et savent l'exécuter. Vous confirmez, et LM Studio annonce que le plugin est installé. C'est tout l'effort d'installation.

Le plugin en place, demandez l'heure au modèle. Vous touchez aussitôt une limite que personne n'imagine au départ : il ne sait pas. Un modèle de langue n'a pas d'horloge. Ses poids ont été figés un jour donné et depuis, le temps ne passe plus pour lui. Même avec current-time chargé, un Gemma 4 à qui l'on demande « quelle heure est-il ? » répond d'abord honnêtement qu'il n'a pas accès à l'heure courante et vous renvoie à votre montre.

Le plugin expose pourtant un outil, get_current_time, que le modèle pourrait appeler. Sur un petit modèle, le réflexe ne vient pas toujours du premier coup et il faut parfois insister. On repose la question, « et là, quelle heure est-il ? », et cette fois le modèle comprend qu'il a un moyen de savoir. Il demande la permission d'appeler l'outil. LM Studio affiche un dialogue : le modèle veut appeler get_current_time, vous laissez faire ou vous refusez. Ce dialogue est le moment-clé, celui où le modèle sort de lui-même pour aller chercher une information qu'il n'a pas.

Vous autorisez, l'outil renvoie l'instant exact avec le fuseau, et le modèle compose sa réponse à partir de ce retour : vendredi 26 juin 2026, 12h07, fuseau Europe/Paris. Le résultat n'a rien de spectaculaire.

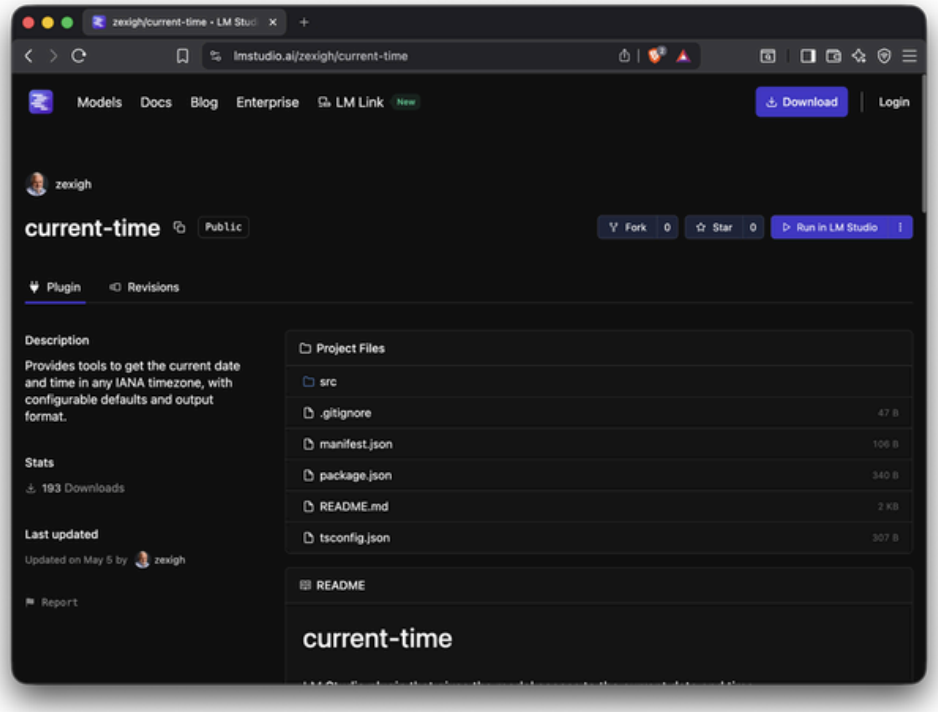


Fig. 5.1 : La fiche de `current-time` sur le Hub. Le bouton **Run in LM Studio** lance l'installation dans l'application.

Ce qui compte est le mécanisme, car il est exactement le même pour tout le reste : donner un outil au modèle, le laisser décider de l'appeler, lui faire composer sa réponse à partir de ce que l'outil rapporte. Lire l'heure ou interroger tout le web, c'est le même geste à deux échelles. Le reste du chapitre déroule la version puissante.

L'expérience qui clarifie tout

Prenez un modèle 4B en local, sans plugin. Posez-lui une question dont la réponse n'est pas dans ses poids parce qu'elle est postérieure à sa date de coupure. Par exemple, dans la démonstration ci-dessous : «Que sait-on du modèle Fable 5 d'Anthropic?», sorti après la coupure du modèle local.

Le modèle va répondre. Confiant. Détaillé. Et faux. Il va inventer des dates, des lieux, des déclarations qui n'ont jamais existé. Ce n'est pas qu'il ment : il complète le motif statistiquement le plus probable à partir de ce qu'il sait. Sans contact avec le monde réel, ce motif n'a aucune chance d'être juste.

C'est ce qu'on appelle une ***hallucination***. Le terme est trompeur parce qu'il évoque un dysfonctionnement, alors que c'est le mode normal d'un modèle de langue sans accès à la réalité. Halluciner, pour un LLM, c'est simplement faire son travail (prédire le mot suivant) dans une situation où il manque l'information.

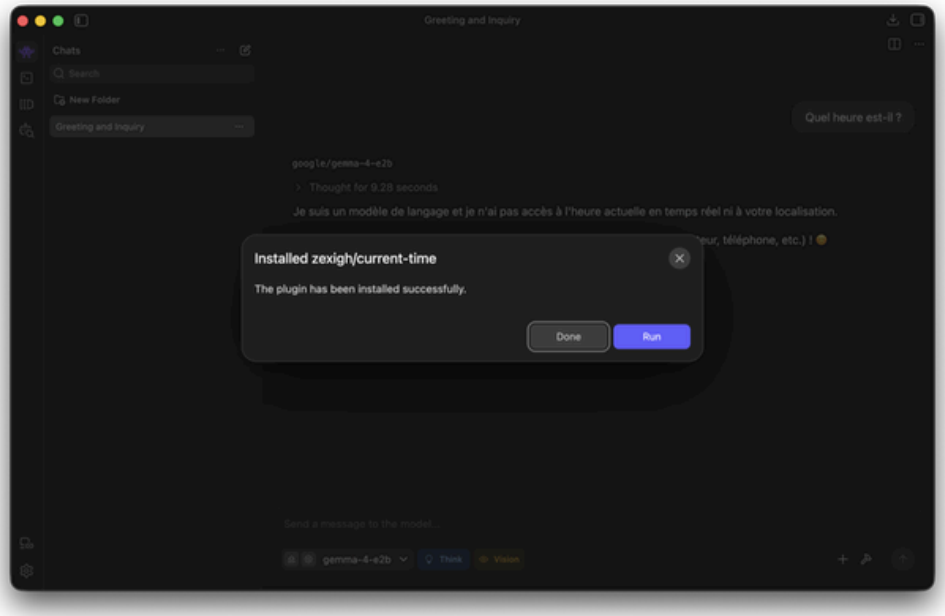


Fig. 5.2 : Le plugin est installé. **Run** l'active tout de suite dans la conversation en cours.

Maintenant, installez puis activez le plugin web-search (même geste que pour current-time : **Run in LM Studio** depuis sa fiche sur le Hub, lmstudio.ai/zexigh/web-search, capture plus bas). Reposez exactement la même question, au même modèle. Le modèle interroge un moteur de recherche (DuckDuckGo en l'occurrence), récupère quelques résultats, et compose sa réponse à partir de ces résultats réels. Cette fois, il cite, il date, il référence. Et c'est juste.

Même modèle. Même question. La seule différence est un outil. C'est le coup pédagogique le plus net que je connaisse pour faire comprendre la logique du local outillé.

Pourquoi c'est le concept central

Quand on découvre les LLM, on a tendance à juger un modèle uniquement sur sa taille et son score à des benchmarks. C'est la métrique de l'IA cloud, où l'on compare GPT-4, Claude Opus, Gemini Ultra sur leurs capacités intrinsèques. Cette métrique est inopérante en local.

En local, ce qui compte est le couple modèle + outils. Un 4B avec un wiki de notions bien fait fait mieux qu'un 70B nu sur la maïeutique d'une matière donnée. Un 8B avec web-search fait mieux qu'un 30B sans outil sur toute question d'actualité. Un 4B avec un plugin de calcul formel fait mieux qu'un 100B sans outil sur des problèmes de maths.

L'analogie qui revient souvent dans la communauté est celle du bureau. Vous ne jugez pas un comptable sur sa mémoire mais sur sa capacité à manipuler ses outils (calculatrice, tableur, codes fiscaux). Personne ne demande à un comptable de retenir par cœur le code général des impôts. C'est exactement le déplacement qu'il faut faire pour le local : le modèle est l'opérateur, les outils sont sa documentation.

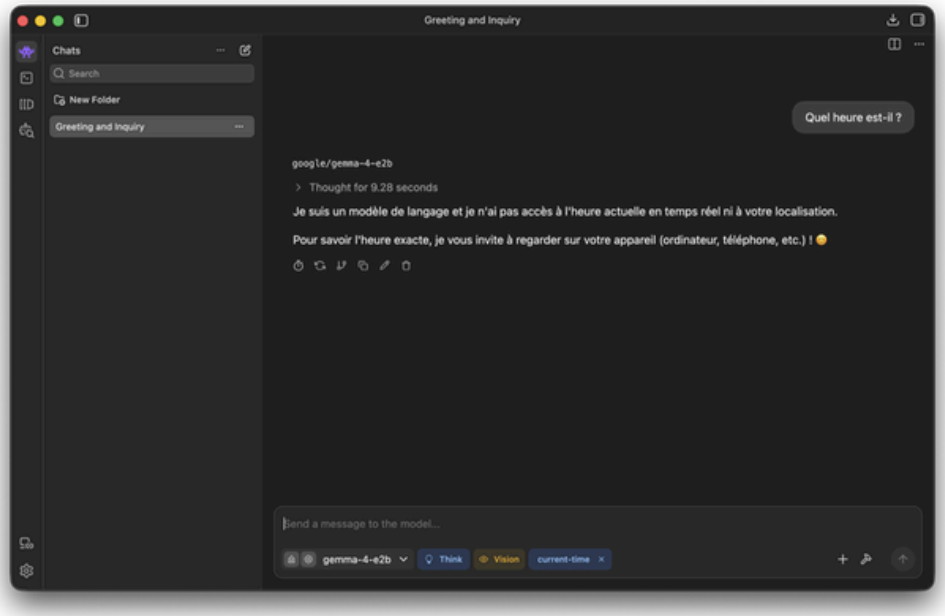


Fig. 5.3 : Le plugin current-time est actif (en bas), mais à la première formulation le modèle décline : il n'a pas d'horloge, et il le dit.

Ce déplacement n'a rien d'une bricole de bidouilleur. C'est le chemin qu'ont pris les grands modèles propriétaires eux-mêmes. Les premiers ChatGPT, il y a quelques années, ne savaient rien faire d'autre que répondre de mémoire : pas de recherche web, pas d'outils, une date de coupure et puis le silence sur tout ce qui venait après. Depuis 2023, OpenAI leur a ajouté la recherche web et l'appel de fonctions, et c'est aujourd'hui le modèle lui-même qui décide d'aller chercher en ligne ou de lancer un outil. Donner des outils à un modèle local ne fait que rapatrier une capacité devenue standard, sauf qu'ici elle tourne chez vous, sur vos données. (OpenAI, l'outil de recherche web de ses propres modèles)

Comment fonctionne le plugin web-search

Techniquement, web-search expose un seul outil au modèle : search_web. Le modèle formule une requête en langage naturel, l'outil renvoie une liste de résultats classés (titre, URL, extrait), puis le modèle compose sa réponse à partir de ces extraits. À chaque appel, il décide s'il a besoin d'une recherche, formule la requête, lit les extraits et continue son raisonnement.

La décision ***quand chercher*** est prise par le modèle, pas par vous. C'est sa responsabilité de juger qu'il ne sait pas et qu'il faut chercher. Sur les bons modèles, ce réflexe est correct : pour une question sur un événement récent, ils cherchent; pour une question conceptuelle stable, ils répondent de mémoire. Sur les modèles plus petits, il faut parfois forcer la main, en disant explicitement «cherche sur le web avant de répondre».

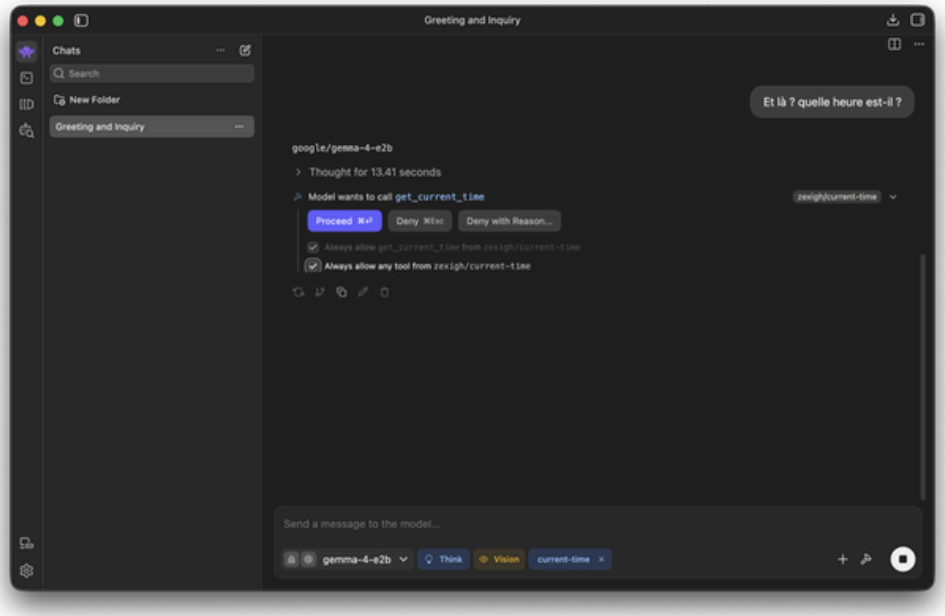


Fig. 5.4 : Le modèle demande l'autorisation d'appeler `get_current_time`. **Proceed** exécute l'outil, **Deny** le refuse. Vous restez maître de chaque appel.

Le moteur par défaut est DuckDuckGo, qui ne demande aucune configuration et respecte raisonnablement la vie privée. Le plugin accepte aussi **SearXNG**, un méta-moteur libre et auto-hébergé : si vous faites tourner votre propre instance, vous renseignez son URL et vos recherches ne sortent jamais de chez vous. Pour la plupart des usages quotidiens, DuckDuckGo suffit. Et comme le plugin est le vôtre, rien n'empêche d'y brancher d'autres fournisseurs : c'est quelques lignes à ajouter, dans l'esprit du chapitre 6.

Où trouver d'autres plugins

Les plugins de ce livret (anonymize au chapitre 4, le wiki du chapitre 3, web-search et current-time ici) sont rassemblés sur une même page, lmstudio.ai/zexigh : tout ce que la formation a produit, prêt à installer d'un clic.

Pour le reste, sachez qu'il n'existe pas, à ce jour, d'index officiel de tous les plugins LM Studio ; la découverte se fait de proche en proche. Une liste curatée par un utilisateur, tupik/top, en recense un bon paquet et donne une idée de ce qui se fait. Bon point de départ pour explorer, en gardant le réflexe du chapitre : un plugin, c'est du code qui tournera chez vous, jetez un œil à ses fichiers avant de l'installer.

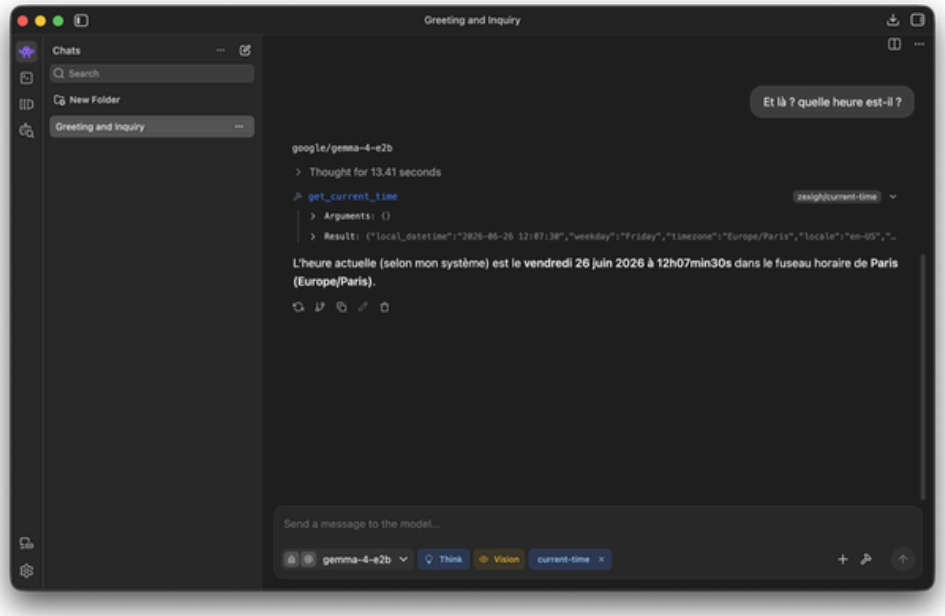


Fig. 5.5 : L'outil renvoie l'heure réelle, le modèle la met en phrase. Le même mécanisme, appel d'outil puis composition, vaut pour tous les outils qui suivent.

Ce que ça change pour l'utilisateur

Le bénéfice immédiat est évident : le modèle local devient utile sur les sujets d'actualité, qui étaient jusqu'ici hors de sa portée. Vous pouvez poser une question sur les annonces de la semaine, sur un article publié hier, sur un événement récent. Le modèle ne fera plus semblant : il ira voir.

Le bénéfice plus subtil est dans la ***posture***. Vous savez que le modèle peut chercher si besoin, donc vous lui posez les questions différemment. Au lieu d'écrire un prompt long avec tout le contexte que vous fournissez, vous écrivez court et vous le laissez compléter ses lacunes. C'est moins fatigant et c'est plus juste, parce que le modèle va chercher la version réelle plutôt que travailler sur celle que vous croyez connaître.

C'est aussi la première fois où le local devient un ***fact-checker***. Vous lui collez une affirmation entendue à la radio ou lue sur un réseau, vous demandez « vérifie », et il revient avec les sources. Sur un cloud commercial, ce service existe mais il transmet ce que vous voulez vérifier, rattaché
à votre compte. En local, vous gardez la main sur ce qui sort : seule la requête de recherche part vers le moteur choisi (DuckDuckGo par défaut, qui ne vous profile pas), pas le reste de votre conversation ni vos documents; et avec un SearXNG auto-hébergé, même la requête ne quitte pas votre réseau.

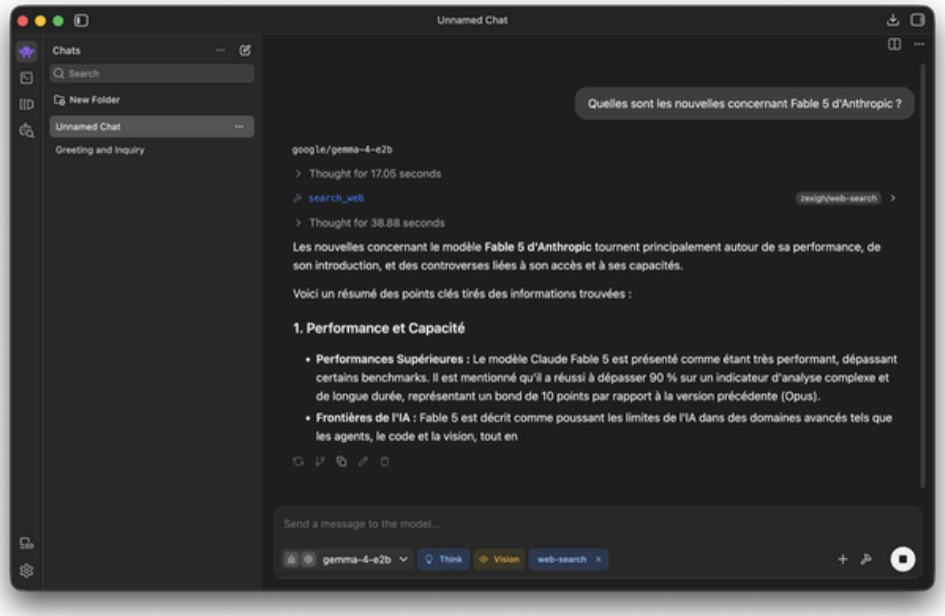


Fig. 5.6 : Même petit modèle, même question, mais web-search activé : la réponse est cette fois structurée, datée et tirée de résultats réels, là où sans outil il aurait inventé.

La limite : le modèle reste seul juge de ce qu'il lit

Un outil ne fait pas un discernement. Le modèle qui fait une recherche reçoit en retour des extraits de pages web, qu'il considère par défaut comme fiables. Si la première page qu'il rapporte est un blog complotiste, il composera sa réponse à partir de ce blog avec la même assurance que s'il citait Le Monde.

Vous gardez donc le travail d'évaluation. Quand un sujet est sensible (santé, finance, politique), regardez les sources que le modèle cite. Demandez-lui de chercher d'autres points de vue. Recoupez. Le plugin n'est pas une garantie de vérité, c'est une connexion au réel, et le réel inclut beaucoup de bruit.

L'expression que j'utilise volontiers : ***un modèle outillé est responsable de ce qu'il rapporte, vous êtes responsable de ce que vous croyez***.

Web-search et le wiki maïeutique : deux logiques différentes

Au chapitre 3, vous avez vu une autre forme d'augmentation : un wiki local de fiches markdown qui sert de mémoire externe au modèle pour la maïeutique. Web-search est cousin, mais pas identique. Il vaut le coup de comprendre la différence.

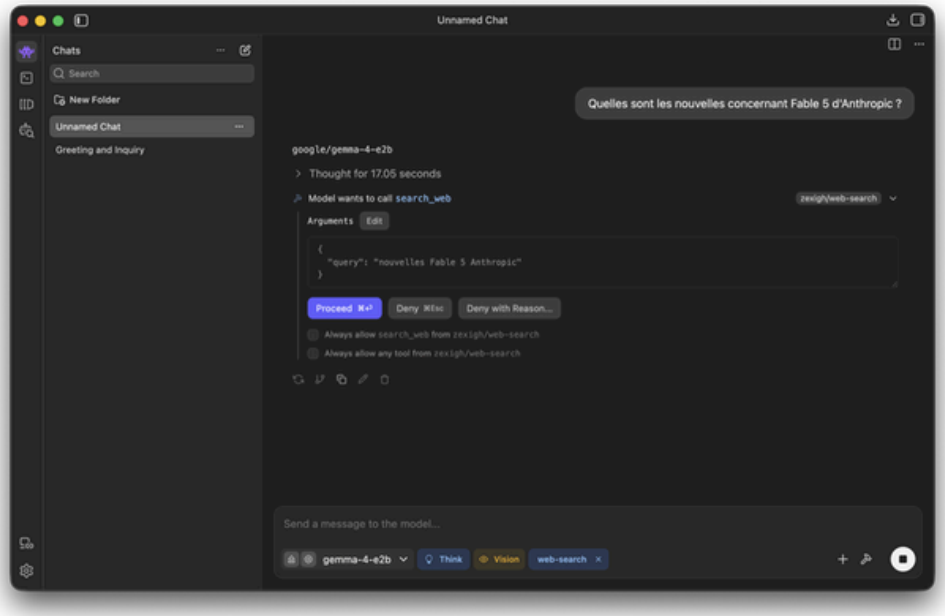


Fig. 5.7 : Le modèle a jugé seul qu'il devait chercher. Il propose son appel à `search_web` avec la requête qu'il a formulée : nouvelles Fable 5 Anthropic.

Le wiki est votre matière, choisie, calibrée, structurée pour l'usage pédagogique. Vous savez ce qu'il y a dedans, donc le modèle qui s'en sert ne peut pas vous surprendre. C'est précieux pour la pédagogie (vous voulez que le maître reste dans le programme) et pour le contrôle (vous savez sur quelle base le modèle compose ses questions).

Web-search est l'inverse : c'est ***le monde***, vaste, non calibré, mêlant excellent et médiocre. C'est précieux pour la couverture (le modèle peut parler de tout ce qui s'est publié) mais c'est coûteux en vigilance (il faut regarder les sources).

Dans un usage avancé, on combine les deux : le modèle a un wiki local pour les notions stables et web-search pour les questions d'actualité. C'est exactement la configuration qu'utilisent les agents les plus efficaces. Le chapitre 8 reviendra sur cette idée d'alimenter le local en savoir-faire structuré.

Le coût en latence et en bande passante

Activer web-search ralentit chaque réponse. Là où un modèle 4B répond en deux à trois secondes sur une question simple, une réponse avec recherche prend dix à trente secondes : il faut le temps de la requête de recherche, de la lecture des extraits et de la composition. Sur un MacBook Air, c'est tout à fait supportable. Sur batterie, c'est un usage à doser.

Côté bande passante, chaque requête tire quelques centaines de Ko à quelques Mo selon le nombre de pages que le modèle consulte. Rien de critique sur une connexion fixe, mais à garder en tête en partage de connexion mobile.

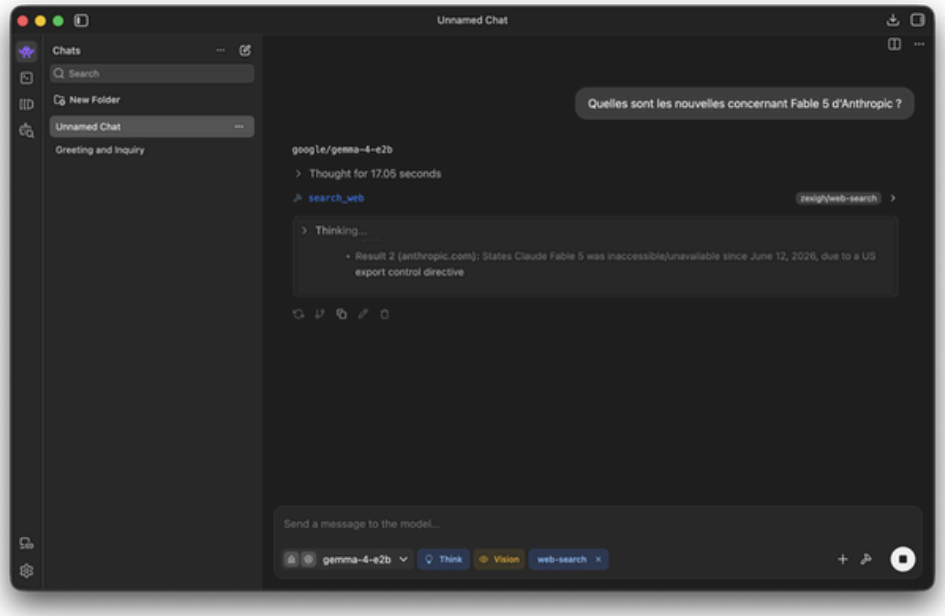


Fig. 5.8 : L'outil a renvoyé des résultats, le modèle les lit pendant sa réflexion. Ici un extrait provenant de anthropic.com entre dans son raisonnement.

Quand désactiver web-search

Trois cas où je désactive le plugin.

Confidentialité absolue. Quand je travaille sur un document sensible et que je ne veux pas qu'une requête liée à ce document apparaisse dans les logs DuckDuckGo, je coupe le plugin. Le modèle reste sur ses propres connaissances.

Pédagogie pure. Quand je suis en mode maïeutique et que je veux que le modèle reste dans le wiki, j'évite que web-search détourne la conversation vers une source externe. Le rôle du maître est de rester dans le cadre.

Mode hors-ligne. En déplacement sans connexion, le plugin produirait des erreurs. Mieux vaut le couper pour éviter les boucles d'échec.

La friction qui vient de disparaître

Le basculement de ce chapitre tient en une posture nouvelle : votre modèle local peut désormais vérifier. Vous lui collez une affirmation entendue à la radio, lue dans un mail, partagée sur un réseau, et il revient avec les sources. À vous de choisir par où passe la requête : un moteur qui ne vous profile pas comme DuckDuckGo, ou votre propre instance SearXNG d'où rien ne sort. La question que je vous laisse est simple : combien de fois, cette semaine, avez-vous laissé passer une affirmation douteuse parce que la vérifier en ligne revenait à la confier à un service qui vous suit à la trace? Cette friction-là vient de disparaître.

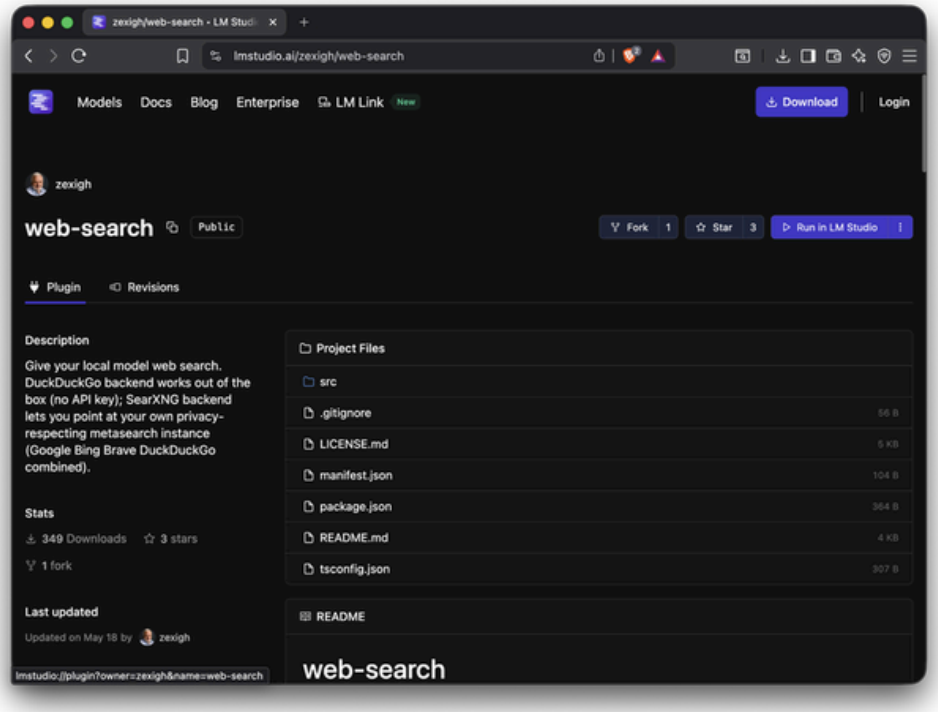


Fig.5.9 : La fiche du plugin web-search sur le Hub LM Studio, à l'adresse lmstudio.ai/zexigh/web-search. Le bouton **Run in LM Studio** l'installe, comme pour current-time. On y lit les deux moteurs gérés, DuckDuckGo par défaut et SearXNG pour l'auto-hébergé.

Vous avez aussi terminé la Partie II du livret : ***Le local au quotidien***. À partir d'ici, on entre dans la Partie III, celle du constructeur. Le chapitre suivant montre comment fabriquer vous-même un plugin LM Studio en une à deux heures, avec un assistant de code. Ensuite vient le chapitre du harness, qui est le cœur méthodologique du livret. Et on termine par la manière d'alimenter le local en savoir-faire structuré.

Troisième partie

Devenir constructeur

Chapitre 6

Vibe-coder un plugin LM Studio

Je vais être honnête avec vous. Les plugins que je présente dans cette formation, je ne les ai pas écrits à la main. Je les ai cadrés, spécifiés, testés. Mais le code lui- même a été produit par un assistant (Claude Code, Codex), à partir de prompts soigneusement construits. C'est ce qu'on appelle aujourd'hui vibe-coder : décrire ce qu'on veut, laisser le modèle composer, itérer jusqu'à ce que ça tienne. Ce chapitre vous donne la recette pour en faire autant.

Pourquoi c'est accessible

La barrière historique d'écriture d'un plugin était la maîtrise du SDK : connaître la structure d'un projet, les API à appeler, les conventions du framework, les pièges typiques. Cette barrière s'est effondrée avec les assistants de code, parce qu'un bon assistant lit le SDK pour vous, suit ses conventions, et produit un squelette fonctionnel en quelques minutes.

Le travail qui reste, et qui ne se délègue pas, c'est de savoir ***ce que vous voulez***. Quel problème le plugin résout. Quels paramètres il prend. Quel résultat il renvoie. Comment il échoue. Ces décisions sont métier, pas techniques. Personne d'autre que vous ne peut les prendre. L'assistant transforme cette spec en code, à la condition que la spec soit claire.

C'est pour ça que je parle d'une heure ou deux par plugin une fois la recette stabilisée. La première fois, ce sera plus long : vous tâtonnez sur le format de la spec, sur la manière de cadrer l'assistant, sur les tests à demander. À partir du deuxième ou troisième plugin, la recette se met en place et chaque nouveau plugin devient rapide.

L'anatomie minimale d'un plugin LM Studio

Avant de coder quoi que ce soit, il faut savoir à quoi ressemble un plugin. Trois fichiers suffisent dans le cas le plus simple.

manifest.json décrit le plugin à LM Studio : son nom, sa version, les outils qu'il expose au modèle, leur signature (paramètres acceptés, types de retour). C'est le contrat avec l'environnement.

src/ contient le code TypeScript du plugin. Une fonction par outil exposé, plus les utilitaires. C'est là que tout se passe.

package.json déclare les dépendances Node et les scripts de build. Le SDK LM Studio est installé via npm, comme n'importe quelle dépendance.

Un plugin complet (le mien web-search, par exemple) a six fichiers source et environ 450 lignes. Un plugin trivial comme current-time (rendre l'heure courante) tient en quatre fichiers et moins de 200 lignes. La barre est basse.

La recette · cinq étapes

Étape 1 · Écrire `PROMPT.md`

C'est l'étape qui détermine tout. Vous écrivez, en français ou en anglais, ce que vous voulez. Pas du code, du cahier des charges. Un exemple, pour un plugin fictif currency-convert qui convertit des montants entre devises.

```
# Plugin currency-convert

## But
Permettre au modèle de convertir un montant d'une devise vers une
autre en temps réel, en utilisant un taux de change officiel.

## Outil exposé
`convert_currency(amount: number, from: string, to: string): {
  amount: number,
  rate: number,
  asof: string
}`

## Spec
- `from` et `to` sont des codes ISO 4217 (EUR, USD, JPY##.).
- Source du taux : api.exchangerate.host (gratuit, sans clé).
- Si le code est inconnu, renvoyer une erreur typée
  `InvalidCurrencyError` avec le code fautif.
- Si l'API est injoignable, renvoyer `NetworkError` avec un message
  clair.
- Le champ `asof` est la date du taux retourné par l'API.

## Contraintes
- TypeScript, SDK LM Studio @lmstudio/sdk.
- Propager `ctx.signal` pour permettre l'annulation utilisateur.
- Émettre `ctx.status` pendant la requête HTTP ("Récupération du
  taux##.").
```

C'est tout. Trois paragraphes. La spec est complète : but, signature, comportement attendu, contraintes techniques explicites. C'est exactement ce dont l'assistant a besoin.

Étape 2 · Fournir une référence SDK

L'assistant connaît grossièrement les bibliothèques connues, mais il n'a pas forcément vu la version récente du SDK LM Studio. Donnez-lui une référence. Le plus simple : copier-coller un plugin existant qui marche.

Je garde un plugin de référence minimal, current-time, dont la structure est canonique. Quand je vibe-code un nouveau plugin, je commence par dire à l'assistant : « Voici la structure d'un plugin LM Studio qui fonctionne (et je colle le code de current-time). Reproduis cette structure pour le plugin spécifié dans PROMPT.md. »

Cette étape vous évite l'essentiel des erreurs de débutant : sans elle, l'assistant inférerait des conventions inventées, et le plugin ne se chargerait pas dans LM Studio.

Étape 3 · Laisser l'assistant générer

Vous lancez l'assistant avec les deux entrées (PROMPT.md + plugin de référence) et vous regardez. Sur les bons assistants (Claude Code, Codex), le squelette sort en quelques minutes : manifest, code TypeScript, package.json. Il n'y a généralement pas grand-chose à corriger sur cette première passe si la spec est nette.

Là où vous gardez la main, c'est sur la lecture du code. Pas pour le réécrire, pour le comprendre. Lisez-le. Demandez à l'assistant d'expliquer ce qu'il a fait. Vous n'êtes pas en train de devenir développeur, mais vous êtes responsable du plugin qui va tourner sur votre machine. Comprendre les grandes lignes est non négociable.

Étape 4 · Tester manuellement dans LM Studio

C'est l'étape où la réalité reprend ses droits. Vous installez le plugin dans LM Studio (un dossier dans ~/.lmstudio/plugins/), vous démarrez une conversation, et vous demandez explicitement au modèle d'utiliser le nouvel outil. «Convertis 100 euros en yens.»

Trois cas possibles. Le plugin marche du premier coup (assez fréquent sur les plugins simples). Le plugin charge mais l'outil échoue (le plus courant). Le plugin ne charge même pas (typique- ment, problème de manifest).

Pour chacun, le bon réflexe : copier-coller l'erreur dans l'assistant, dire « voici ce que j'obtiens, corrige ». Itérer trois ou quatre fois suffit la plupart du temps. Si vous tournez en rond, c'est en général que la spec initiale était floue : retournez à PROMPT.md, précisez, et reprenez.

Étape 5 · Tester les cas limites

Quand le cas nominal marche, vous testez les cas que vous avez spécifiés dans PROMPT.md.

Devise inconnue : est-ce que l'erreur typée sort correctement ? API injoignable (vous coupez le 47 Chapitre 6 Vibe-coder un plugin LM Studio wifi) : est-ce que le message est clair ? Annulation utilisateur en cours de requête : est-ce que ça s'arrête vraiment ?

Cette étape est ce qui sépare un plugin qui marche en démo d'un plugin que vous pouvez réutiliser sans surprise. Sautez-la et vous découvrirez les bugs au pire moment.

La frontière prototype vs produit

Il faut être lucide sur ce que produit cette méthode. Ce que vous obtenez, c'est un ***prototype*** fonctionnel. Pour passer au ***produit***, il manque tout un travail que l'assistant ne fait pas sponta- nément : robustesse face aux pannes, propagation propre des erreurs, internationalisation des messages, tests automatisés, performance sous charge.

Concrètement, j'ai vibe-codé une première version de web-search qui marchait en démo, et qui a planté la première fois qu'un utilisateur en production a déclenché une détection anti-bot de DuckDuckGo. Le plugin ne gérait pas ce cas. Il a fallu reprendre, ajouter le code de détection, typer l'erreur correctement, propager un message utilisable. Ce travail-là, je l'ai fait à la main une fois compris ce qui manquait.

Le vibe-coding est excellent pour explorer, prototyper, valider une idée. Pour livrer un outil dont d'autres dépendent en production, il faut généralement un deuxième passage avec plus de soin. Mais ce deuxième passage est facultatif pour la plupart de vos usages personnels.

Combien de plugins vous pouvez raisonnablement avoir

Une question revient souvent en formation : faut-il vibe-coder plein de plugins ? Ma réponse honnête : non. Vibe-codez les plugins dont ***vous*** avez besoin, qui n'existent pas déjà. Un plugin par usage récurrent que vous identifiez, pas plus.

Au moment où j'écris ce livret, j'ai une vingtaine de plugins, et la plupart sont des variantes spécialisées (revise-bac, citations-philo) plutôt que des outils génériques. Les vrais outils génériques (filesystem, anonymize, web-search) sont rares parce qu'ils sont déjà couverts par la communauté.

Un cas idéal de vibe-coding : un workflow précis dans votre métier (consultant, juriste, en- seignant) qui n'a aucune chance d'intéresser quelqu'un d'autre. Un plugin qui formate vos rapports clients selon votre template à vous. Un plugin qui interroge le SI de votre entreprise. Un plugin qui calcule un indicateur métier propre à votre activité. C'est là que la valeur est maximale.

Le coût de maintenance

Un plugin vibe-codé n'est pas gratuit dans le temps. Quand le SDK LM Studio évolue, votre plugin peut casser. Quand l'API externe qu'il appelle change, idem. Vous devrez le réviser, soit manuellement, soit en relançant l'assistant avec un nouveau cycle.

Mon expérience : sur une vingtaine de plugins maintenus depuis quelques mois, environ deux ou trois ont demandé une révision pour s'adapter à un changement d'API externe. Le SDK LM Studio a été stable jusqu'ici, mais ce ne sera probablement pas éternel. Comptez une demi-journée par an de maintenance pour cinq à dix plugins actifs.

La compétence qui se déplace

Si vous suivez cette méthode plusieurs fois, vous ne devenez pas développeur. Vous devenez ***cadreur***. Vous apprenez à écrire des specs précises. À reconnaître ce que vaut une référence donnée à l'assistant. À identifier les cas limites avant qu'ils ne vous tombent dessus. À lire du code sans le comprendre dans le détail mais en repérant les zones à risque.

C'est une compétence transverse, qui s'applique à toutes vos interactions avec un assistant de code, pas seulement aux plugins LM Studio. C'est ce que tend à devenir le métier de chef de produit ou de lead technique en 2026 : moins de code écrit, plus de spec, plus de relecture, plus de jugement. La méthode du vibe-coding vous fait travailler ces muscles.

Maintenant, allez en construire un

Le meilleur usage que vous puissiez faire de ce chapitre, c'est de le refermer et d'ouvrir votre éditeur. Choisissez l'outil le plus bête qui vous manque, celui qui rend l'heure, qui convertit une devise, qui lit un fichier, et faites-le exister ce soir. Un plugin trivial tient en quatre fichiers et deux cents lignes, et vous tenez déjà la recette en cinq étapes. Vous n'avez pas à savoir coder pour ça ; ce qu'il vous faut, c'est savoir ce que vous voulez ; le reste, l'assistant le compose. Tant que vous n'en avez pas fabriqué un de vos mains, ce chapitre reste une lecture ; le jour où le vôtre se charge dans LM Studio et répond, il devient un savoir-faire.

On monte d'un cran. Vibe-coder un plugin avec un assistant de code adossé à un modèle dans le cloud (Claude Code, Codex) est utile, mais le travail reste exposé : votre spec, vos retours d'erreur, le contenu de vos plugins transitent par un service tiers. Le chapitre suivant montre comment faire la même chose entièrement en local, avec ce qu'on appelle un ***harness*** — la pièce la plus dense du livret, et celle qui rend tout le reste cohérent.

Chapitre 7

Le harness · théorie et récit

C'est le chapitre central du livret. Pas par hasard : si vous deviez ne retenir qu'un concept de cette formation, ce serait celui du harness. Le mot ne fait pas encore partie du vocabulaire courant en français (on traduit littéralement par harnachement), mais c'est ce qui sépare un modèle local jouable d'un modèle local opérationnel. Ce chapitre vous explique ce que c'est, ce qu'il contient, pourquoi il vaut plus que la taille du modèle, et ce que j'ai appris en construisant le mien en local.

Le mot qui manquait

Un modèle, tout seul, ne fait pas grand-chose d'utile. Même un 70B. Vous lui posez une question, il complète. Vous lui demandez d'écrire un fichier, il vous renvoie du texte, pas un fichier. Vous lui demandez d'utiliser un outil, il ne sait pas où ce tool se trouve, ni comment l'appeler, ni quoi faire de son retour.

Ce qui rend un modèle opérationnel, c'est tout l'***harnachement*** autour. Un prompt système qui définit son rôle. Des descriptions de tools rédigées pour qu'il sache quand les utiliser. Une boucle d'exécution qui transforme ses réponses en actions. Une gestion du contexte qui ne déborde pas. Une mémoire entre sessions. Des garde-fous qui l'empêchent de boucler. De l'observabilité pour comprendre ce qui s'est passé. On appelle l'ensemble un ***harness***.

Tout ce que vous utilisez quotidiennement avec une IA est un harness. Claude Code est un harness. Codex est un harness. Cursor, Aider, OpenCode, Vibe, sont des harnesses. LM Studio est un harness simple, avec son système de plugins. La distinction qu'il faut faire désormais en local est celle-ci : ***je télécharge un modèle, mais ce que j'utilise est un harness***. C'est dans le harness que se trouve le savoir-faire.

> **Pour les débutants : le harness en une phrase**
>
> Le modèle, c'est la partie qui « réfléchit » et écrit du texte. Le harness, c'est tout ce qu'on installe autour pour qu'il ***agisse*** vraiment : des outils qu'il peut appeler (chercher sur le web, lire un fichier, calculer), une mémoire d'une session à l'autre, une méthode pour enchaîner les étapes, des garde-fous pour qu'il ne tourne pas en rond. Sans ce harnachement, le modèle ne fait que parler. Avec lui, il travaille. Pour reprendre l'image de la voiture : le modèle est le moteur, le harness est tout ce qui transforme cette puissance en un déplacement utile.

Pourquoi c'est plus important que la taille du modèle

Andrej Karpathy l'a formulé clairement : ***le modèle est le noyau cognitif, le reste est l'agent***. Prenez un moteur de Formule 1 : seul, il ne va nulle part. Le moteur, c'est la puissance, ce n'est pas lui qui fait rouler la voiture. Une voiture peut rouler sans moteur, en descente ou poussée; un moteur sans châssis, lui, reste cloué au sol. Le modèle est le moteur, le harness est le châssis, les roues et la direction. C'est l'ensemble qui avance.

Le modèle n'est donc pas ce qui fait le travail, même si c'est lui qui, au bout du compte, le fait. Pour atteindre la qualité d'un Claude Code, il faut une donnée qui n'existe nulle part sur le web : la trace de vrais développements, analysés, repris, corrigés. Cette donnée, c'est le harness qui la produit, en pilotant le modèle sur des tâches réelles et en gardant ce qui marche. Le cycle complet tient en trois temps : un modèle de base, un harness qui le met au travail et génère cette donnée comportementale, puis un entraînement qui la réinjecte dans le modèle. L'analyse au niveau du code source de Claude Code, publiée en 2026, a confirmé l'ampleur de cette couche : l'essentiel du savoir-faire agentique vit dans le harness, pas dans les poids. (analyse architecturale de Claude Code, MBZUAI)

Un exemple tout frais le montre. En juin 2026 sort VibeThinker-3B, un modèle de 3 milliards de paramètres qui, sur des concours de maths, rivalise avec des modèles bien plus gros (94,3 à l'AIME 2026). Je l'ai testé, c'est bluffant. Mais son raisonnement repose sur une prémisse intenable : pour calculer, il ne compte que sur sa propre tête. Demandez-lui des nombres premiers, et c'est sa capacité interne qui peine, là où une calculatrice donnerait le résultat sans risque d'erreur. C'est précisément là que le harness devient indispensable : il confronte le modèle au réel (un vrai outil de calcul, une vraie recherche) et il enrichit son contexte avec ce que le modèle ne peut pas produire seul. (VibeThinker-3B, rapport technique arXiv)

Conséquence directe pour le local : la prochaine bascule de capacité ne viendra pas d'un modèle deux fois plus gros. Elle viendra du harness que vous aurez construit, ou que la communauté construira, autour de votre modèle 14B. C'est là que se trouve la marge de progression la plus accessible.

C'est aussi la raison pour laquelle copier un modèle frontière en distillant ses réponses ne suffit pas. Quand Minimax a, selon Anthropic, créé des comptes pour distiller Claude, ils ont récupéré le comportement du système entier (modèle + harness), pas seulement les poids du modèle. (Anthropic, détection des attaques par distillation) Sans le harness, le savoir-faire s'évapore. Le harness n'est pas dans les poids, il est autour.

Anatomie d'un harness

À l'intérieur d'un harness, on trouve toujours deux couches.

Le cœur visible. Le prompt système qui définit le rôle. Les définitions des outils, avec leur schéma et leur description pédagogique. La boucle d'exécution qui enchaîne ***appel modèle → action → retour de l'action → relance du modèle*** jusqu'à la condition d'arrêt. C'est ce que la plupart des tutos appellent un agent loop.

L'infrastructure invisible. La gestion du contexte qui compresse, résume, fait glisser une fenêtre quand ça déborde. La mémoire persistante entre sessions (votre MEMORY.md de Claude Code, le projet long terme). Les sous-agents, qui permettent de déléguer une tâche isolée à un agent au scope étroit. Les garde-fous (limite de tours, file-lock après patch, abort sur réponse vide). L'observabilité turn-par-turn, qui rend le harness reproductible et debuggable.

Quand vous utilisez Claude Code ou Cursor, vous utilisez tout ça. Le modèle est dedans, mais l'essentiel du travail intéressant est dans la machinerie. C'est cette machinerie qu'il faut comprendre si vous voulez construire la vôtre, ou simplement bien utiliser celle des autres.

L'analogie qui rend tout clair

L'image que j'utilise volontiers en formation : vous ne demandez pas à un enfant de trois ans de conduire une Ferrari. Si vous le faites, il se plantera, et il se plantera bruyamment.

Mais vous pouvez prendre le meilleur pilote au monde, l'asseoir à côté du gamin, et le faire former pendant des années. Au bout de cette formation, le gamin pilote la Ferrari tout seul. Pas aussi bien que le maître, mais correctement, sur des circuits raisonnables.

C'est exactement la stratégie du local outillé. Le meilleur pilote, c'est Claude Code ou GPT-5, en cloud. La Ferrari, c'est votre tâche complexe (anonymiser un contrat, chercher sur le web, dériver une équation). Le gamin, c'est votre modèle 14B local. Le harness, ce sont les leçons que vous transférez du maître à l'élève.

Vous payez le maître une fois pour qu'il forme l'élève. Ensuite l'élève travaille tout seul, à coût marginal nul. C'est ce que mes plugins font. Claude Code a appris à mon LM Studio local comment chercher sur le web, comment anonymiser, comment dériver. Une fois la leçon transmise, je n'ai plus besoin du maître pour les usages courants. Je le réinvoque seulement pour la prochaine leçon.

Les six ingrédients d'un bon harness

Ces ingrédients, je ne les ai pas inventés. Je les ai découverts en regardant mon harness échouer pendant onze itérations sur une seule tâche (j'y reviens plus bas). Chaque échec a ajouté un ingrédient. C'est ça, l'expérience d'un harness : la collection des trous que vous avez bouchés.

Références inlinées, pas descriptives. Quand vous voulez que le modèle reproduise un pattern, montrez-lui un exemple complet, pas une description du pattern. Le modèle reproduit ce qu'il voit, il n'invente pas ce qu'on lui résume. La différence est énorme en sortie.

Descriptions de tools écrites pour le modèle. Un schéma JSON nu n'aide pas un LLM à décider quand utiliser un tool. Écrivez en prose, en vous adressant au modèle : «Utilise ce tool quand X. N'utilise pas ce tool quand Y. Voici un exemple d'appel correct. » C'est cent fois plus efficace.

Plusieurs filets de sécurité, pas un seul. Limite de tours, limite de répétitions, abort sur prose vide, file-lock après patch. Empilez-les. Si l'un saute, l'autre rattrape. Un seul filet finit toujours par lâcher au pire moment.

Smoke tests dans la boucle, pas en aval. Vérifiez après chaque écriture, exécutez le code avant que le modèle puisse déclarer la tâche terminée. Sans cela, le modèle apprend à ***prétendre*** qu'il
a fini sans avoir vérifié. C'est un travers structurel des LLM : ils répondent à la forme attendue, pas au fond.

Sous-agents pour casser les boucles. Si le modèle principal régresse trois fois sur la même erreur, lancez un sous-agent au scope étroit. Il reçoit l'erreur exacte, le fichier, et la mission de corriger. Il ne porte pas l'historique de la boucle, donc il ne reproduit pas la même bêtise. Vous verrouillez ensuite le fichier pour empêcher le principal d'écraser le patch.

Observabilité turn-par-turn. Un fichier de log par tour, avec en-tête structuré (action prise, résultat, état du contexte). C'est ce qui rend le harness reproductible et publiable comme matériel pédagogique. Sans logs, vous ne pourrez ni comprendre pourquoi ça a échoué, ni améliorer.

Deux patterns qui valent leur poids

Parmi les ingrédients ci-dessus, deux patterns méritent un détour parce qu'ils répondent à des problèmes très précis et qu'ils transforment radicalement les résultats.

Pattern · sous-agent + file-lock. Le modèle principal écrit un fichier. Le typecheck échoue. Le modèle relit l'erreur et réécrit le même fichier avec le même bug. Cela peut arriver trois, quatre, cinq fois d'affilée. Ce n'est pas qu'il est bête, c'est que son contexte est saturé par la formulation du bug et qu'il y retombe par cohérence statistique. Solution : à la troisième occurrence, vous lancez un sous-agent avec une mission unique («voici l'erreur, voici le fichier, fixe-la»). Il patche dans un contexte vierge, donc il réussit. Vous verrouillez ensuite le fichier pour trois tours pour empêcher le principal d'écraser le patch. C'est rustique et c'est miraculeusement efficace.

Pattern · smoke test contre le spec gaming. La spec dit : « le tool doit retourner des résultats ». Lemodèleécritfunction search() { return []; }.Letypecheckpasse.Lemodèledéclare la tâche terminée. La forme est juste, le fond est vide. C'est ce qu'on appelle le ***spec gaming*** : quand la mesure devient l'objectif, elle cesse d'être une bonne mesure. Solution : votre fonction de validation ne se contente pas du typecheck, elle exécute le tool sur un cas-test et vérifie que la sortie contient ***au moins N résultats non vides***. Coût : un cas-test à écrire. Bénéfice : le modèle ne peut plus tricher.

Récit · onze itérations en local

Voilà la partie où le livret devient honnête. En mai 2026, quelques jours avant cette formation, j'ai voulu vérifier une thèse : est-il possible de vibe-coder un plugin LM Studio, entièrement en local, en utilisant un modèle 14B comme moteur et un harness maison pour l'orchestrer ? Pas d'API cloud, pas de Claude Code, juste mon M5 (un MacBook Air 32 Go) et Qwen2.5-Coder-14B.

J'ai pris le plugin web-search comme cible, parce que je connaissais la solution finale (je l'avais écrit à la main avant). C'était un ***test de capacité***, pas une création.

J'ai lancé treize runs du harness sur cette seule tâche. Deux se sont effondrés sans produire la moindre ligne ; les onze autres ont chacune sorti un prototype que j'ai pu juger. C'est de ces onze itérations que je tire ce qui suit.

Itération 1 à 3. Le harness était minimal : prompt système, définitions de tools, boucle simple. Qwen partait dans tous les sens, oubliait le format, hallucinait des imports. Premier ingrédient ajouté : références inlinées complètes du plugin current-time.

Itération 4 à 5. Avec la référence, Qwen reproduisait la structure mais bouclait sur des bugs récurrents. Notamment, il importait toujours node-fetch alors que le runtime utilise le fetch natif. Trois fois le même bug. Ajout du pattern sous-agent + file-lock.

Itération 6 à 7. Sous-agent en place, plus de boucle infinie sur les imports. Mais Qwen déclarait la tâche terminée alors que la fonction retournait un tableau vide. Spec gaming pur. Ajout du smoke test exécuté dans la fonction done().

Itération 8 à 9. Smoke test ajouté, Qwen passait enfin le test interne… mais sur DuckDuckGo en production, ça plantait. Mystère. Le harness ne voyait rien d'anormal : compile OK, types OK, smoke test OK, appel HTTP OK. Sauf que la réponse contenait un CAPTCHA, pas des résultats.

Itération 10 à 11. L'épiphanie est venue tardivement, après une heure de chantier. Curl passait, fetch était bloqué, depuis la même IP. C'était le User-Agent. Le Mozilla/5.0 tronqué que Qwen avait reproduit (parce qu'il est dans tous les snippets Stack Overflow) était en deny-list de DuckDuckGo. Mon plugin manuel, lui, utilisait un User-Agent ***brandé*** lms-plugin-`web-search/1.0 (+https:#/lmstudio.ai/zexigh/web-search),quipassaitsansencombre.` Qwen n'a jamais inventé ce détail. Il ne pouvait pas. Cette information ne s'apprend pas par lecture, elle s'éprouve.

Bilan : ~400 tours Qwen, ~100 minutes de compute sur M5, aucun run en autonomie complète n'a passé le test final. Et pourtant, le harness avait fait son boulot exactement. Il avait produit un prototype fonctionnel à chaque itération. Ce qu'il ne pouvait pas faire, c'était inventer ce qu'il n'avait jamais vu.

La frontière du harness

Cette session m'a appris la frontière exacte de ce que peut un harness rigoureux autour d'un modèle local modeste. Trois constats.

Le harness amplifie, il ne remplace pas. Petit modèle + harness rigoureux = prototype. Pas produit. La distance entre les deux, c'est la connaissance opérationnelle qui n'est dans aucun corpus d'entraînement.

La référence détermine ce qui sort. Si la référence ne contient pas un pattern, le modèle ne l'inventera pas. La qualité du harness tient pour l'essentiel à la qualité des références que vous inlinez.

Les petits modèles confabulent de manière stable. Le 14B local refait les mêmes erreurs à des endroits prévisibles. Le harness ne le rend pas plus intelligent, il amortit l'impact de ces erreurs. C'est déjà énorme, mais ce n'est pas une cure.

Le harness comme stratégie d'investissement

À l'horizon 2026, investir dans le harness est la meilleure chose qu'un praticien du local puisse faire. Pas parce que c'est facile (ça ne l'est pas), mais parce que c'est là que se trouve le levier le plus grand pour un coût raisonnable. Une fois construit, votre harness travaille pour vous indéfiniment.

Vous n'êtes pas obligé de construire à partir de zéro. La communauté commence à publier des harnesses ouverts (Aider, OpenCode), et LM Studio lui-même est un harness fonctionnel pour les usages courants. Mais comprendre ce qu'est un harness vous met dans une position critique différente : vous savez ce que vous achetez quand vous payez un service, vous savez ce qui manque dans un outil quand il déçoit, vous savez où chercher pour combler.

C'est aussi la compétence qui se monétise le mieux en interne en entreprise. Construire un harness adapté au métier d'une organisation (juridique, médical, commercial) est un projet de quelques semaines pour un praticien outillé, et il décuple la productivité d'une équipe entière. Mieux qu'acheter une licence GPT-5 pour vingt personnes, dans la plupart des cas.

Le harness, et sa frontière

Ce chapitre était le plus dense du livret, et il mérite qu'on en rassemble les fils. Un harness, c'est la machinerie autour du modèle, et c'est là que vit le savoir-faire, pas dans les poids : Karpathy le dit du noyau cognitif, le cycle modèle-harness-entraînement le confirme, VibeThinker et Minimax le montrent par l'absurde. Cette machinerie a une anatomie (un cœur visible fait de prompt, d'outils et de boucle ; une infrastructure invisible faite de mémoire, de sous-agents et de garde-fous) et une raison d'être que l'image du pilote et du gamin fixe une fois pour toutes : on paie le maître une fois pour former l'élève, qui travaille ensuite seul.

Pour la construire, vous tenez six ingrédients et deux patterns qui valent leur poids, le sous-agent qui casse les boucles et le smoke test qui interdit la triche. Et vous tenez surtout le récit de mes onze itérations, qui dit la vérité que les ingrédients ne disent pas : un harness rigoureux autour d'un 14B produit un prototype à tous les coups, jamais le produit. Ce qui manque entre les deux n'est pas une question de modèle plus gros ni de harness plus malin. C'est ce User-Agent brandé que Qwen ne pouvait pas inventer parce qu'il n'est écrit nulle part. Cette connaissance-là ne se lit pas, elle s'éprouve, et c'est la frontière exacte de tout ce que vous venez d'apprendre.

Le chapitre 8 termine le livret en revenant à la matière première du harness : ***les références*** — comment alimenter un local en savoir-faire structuré pour que le couple modèle + harness puisse y puiser, et rendre le tout durable.

Chapitre 8

Alimenter le local en savoir-faire

Vous arrivez au bout du livret. Les sept chapitres précédents vous ont donné une machine (chapitre 1), un logiciel (chapitre 2), trois usages (maïeutique, anonymisation, augmentation par outils), et la méthode pour fabriquer vos propres extensions (vibe-coding, harness). Il reste une dernière pièce, la plus simple en apparence et pourtant celle qui rend tout le reste durable : la matière. Les fiches markdown, les programmes structurés, les références que votre modèle local utilisera comme mémoire externe. C'est cette matière qui fait la différence entre un local impressionnant en démo et un local utile au quotidien.

Pourquoi la matière compte plus que le modèle

Reprenons le fil tendu par les chapitres précédents. Au chapitre 3, vous avez vu que la maïeutique fonctionne parce qu'un wiki de fiches alimente le modèle en notions structurées. Au chapitre 7, vous avez vu que le harness tient pour l'essentiel à la qualité des références inlinées. À chaque fois, ce qui fait la performance, c'est la ***matière*** qu'on donne au modèle, pas le modèle lui-même.

Cette observation a une conséquence stratégique. Quand vous démarrez avec le local, vous passez les premières semaines à comparer des modèles, à benchmarker, à ajuster des paramètres. C'est utile, mais c'est marginal. La vraie progression vient le jour où vous arrêtez de comparer les modèles et où vous commencez à construire votre matière. Un wiki d'une cinquantaine de fiches bien faites améliore davantage votre quotidien qu'un upgrade de 4B à 14B.

L'analogie : le modèle est un cuisinier compétent. Vous pouvez en choisir un meilleur, mais le saut sera marginal. Ce qui changera vraiment vos repas, c'est la qualité des produits que vous lui fournirez. Le local est l'art de bien faire son marché.

Trois types de matière

Toute matière qui alimente un local n'a pas la même nature. J'en distingue trois.

La matière conceptuelle. Des fiches qui définissent des notions, structurent un domaine. C'est ce qu'on a vu pour la maïeutique (raison, désir, liberté en philosophie). Mais c'est valable pour tout : une fiche par concept médical, par texte de loi, par principe comptable. Format compact : définition courte, exemples, contre-exemples, tensions, questions à poser. Trois à cinq pages par fiche maximum.

La matière procédurale. Des modes opératoires, des checklists, des templates. Comment je rédige un rapport client. Comment je structure une consultation. Comment je passe une commande fournisseur. Le modèle y puise des patterns d'action plutôt que des concepts. Format : procédure numérotée, exemples remplis, cas d'exception en fin.

La matière de référence. Des plugins existants à inliner pour le vibe-coding, des extraits de code canoniques, des configurations exemplaires. C'est ce qu'on a vu au chapitre 6 avec le plugin current-time qui sert de référence quand on veut en écrire un nouveau. Format : fichier complet, non résumé, sans commentaire ajouté.

Les trois types coexistent dans votre référentiel, mais ils sont rangés et utilisés différemment. La matière conceptuelle nourrit la maïeutique. La matière procédurale nourrit les workflows métier. La matière de référence nourrit la fabrication d'outils.

Le format markdown comme socle

Tout, dans cette matière, est en markdown. Pas par mode, par robustesse. Le markdown est lisible par n'importe quel humain, parseable par n'importe quel modèle, versionnable dans Git, indexable par n'importe quel moteur de recherche, et il survit aux changements de format pendant que les fichiers Word et les PDF se périment.

Vous ouvrez un fichier .md dans VS Code, dans Obsidian, dans Notion en import, dans un terminal. Cela suffit. Inutile d'investir dans un système propriétaire pour stocker votre savoir : la simplicité du markdown est ce qui rend la matière durable.

Ma règle pratique : si un fragment de savoir métier mérite d'être noté, il devient un fichier .md dans mon référentiel. Pas une note dans un outil propriétaire, pas un email retrouvé six mois plus tard, pas un signet vers une page web qui aura disparu. Un fichier .md chez moi, versionné, sauvegardé.

Structure d'un référentiel qui dure

Un référentiel mal organisé devient inutilisable dès qu'il dépasse une centaine de fichiers. Quelques principes que j'ai vu fonctionner.

Un dossier par domaine, pas un dossier par projet. Vos clients passent, vos domaines de compétence restent. Organisez par ***philosophie***, ***fiscalité***, ***RGPD***, pas par ***client Dupont***, ***projet Acme 2024***. La connaissance domaine se réutilise, le projet ne se réutilise pas.

Un fichier par notion, pas un fichier par bloc temporel. Évitez le ***journal-2026-04.md*** qui ac- cumule des entrées hétéroclites. Préférez ***raison.md***, ***intentionnalité.md***, ***empirisme.md***. Chaque fichier traite une chose, et il vit aussi longtemps que la notion vit.

Un index par dossier. Un fichier index.md qui liste les notions du domaine, avec une ligne par fichier. C'est ce que le modèle lira en premier pour savoir où chercher. C'est aussi ce qu'un humain lira pour s'orienter.

Un journal append-only. Un seul fichier log.md à la racine du référentiel, où vous notez les changements. ***Le 12 mars, j'ai ajouté une fiche sur la phénoménologie. Le 14 mars, j'ai mis à jour raison.md suite à une discussion.*** Cela vous donne l'historique sans gonfler chaque fiche.

Pas de hiérarchie profonde. Trois niveaux maximum : domaine, sous-domaine éventuel, fichier. Au-delà, on se perd. Si une fiche mérite un sous-thème, créez plutôt un fichier dédié et liez les deux.

Cette structure tient à 50 fiches, à 200 fiches, à 1 000 fiches. C'est la même, juste avec plus de fichiers. Pas de réorganisation complète à prévoir tous les six mois.

Le pattern wiki-link entre fiches

Le secret qui rend un référentiel vivant, c'est de lier les fiches entre elles. Convention courante : [[nom-de-la-fiche]] au milieu du texte, qui devient un lien navigable dans Obsidian, dans certains rendus markdown, et qui reste lisible partout ailleurs.

Quand vous écrivez la fiche raison, vous citez naturellement [[*empirisme*]] et [[*rationalisme*]] quand vous parlez des deux écoles. Si ces fiches n'existent pas encore, le lien marque l'intention de les écrire un jour. Si elles existent, vous tissez le réseau.

À six mois, vous obtenez un graphe : chaque notion pointe vers les voisines, le tout se navigue par sauts plutôt que par recherche linéaire. Cette structure de graphe est aussi ce que le modèle exploite quand il lit votre wiki : il peut suivre les liens, élargir le contexte, trouver des analogies.

Calibrer la qualité des fiches

Une bonne fiche n'est pas longue, elle est précise. Trois critères que j'applique pour calibrer.

Une seule notion par fiche. Si vous vous surprenez à parler de deux choses, scindez. La fusion de notions dans une seule fiche est le premier signe que le référentiel va se dégrader.

Des exemples concrets, pas seulement des définitions. Une fiche qui dit ***«la raison est la faculté de connaître»*** n'aide personne. Une fiche qui ajoute ***«Descartes : le doute méthodique, Kant : l'impératif catégorique, Spinoza : la connaissance du troisième genre »*** devient utile pour le modèle qui en composera des dialogues.

Des contre-exemples ou des tensions. Une notion sans contre-exemple est mal cernée. La fiche raison doit dire ce que la raison ***n'est pas*** : la passion, l'imagination, l'opinion. Les tensions sont ce qui rend la matière utilisable en maïeutique, parce qu'elles donnent des angles d'attaque.

Une fiche calibrée tient en une à trois pages écran. Au-delà, vous décomposez. En dessous, vous étoffez ou vous fusionnez avec une voisine.

La maintenance · ce qu'il faut savoir

Un référentiel n'est pas un projet à terminer, c'est une infrastructure à entretenir. Le travail de maintenance est plus léger que l'investissement initial, mais il n'est pas nul.

Ajout au fil de l'eau. Vous tombez sur une notion intéressante (lecture, discussion, projet), vous créez la fiche. Pas plus tard, pas après avoir réuni cinq notions sur le même thème. Maintenant, vite fait, quitte à étoffer plus tard.

Révision sur déclenchement. Quand vous relisez une fiche pour un usage et qu'elle vous paraît obsolète ou incomplète, vous la mettez à jour à ce moment-là. C'est le bon timing parce que vous êtes en contexte. Forcer une révision mensuelle systématique n'a jamais marché chez moi.

Suppression rare. Une fiche périmée se déplace dans un dossier archives/, elle ne se supprime pas. Vous garderez l'historique, et un jour la notion redeviendra peut-être utile.

Index synchronisé. Quand vous ajoutez ou renommez une fiche, vous mettez à jour l'index.md du dossier dans le même geste. Cette discipline coûte trente secondes par fichier et garantit que le référentiel reste navigable.

À mon échelle (plusieurs centaines de fiches sur dix ans), je consacre une à deux heures par mois à la maintenance pure. Au-delà de deux ans, c'est largement rentabilisé.

Programmes officiels et corpus publics

Pour certains domaines, vous n'avez pas besoin de tout écrire vous-même. Des corpus structu- rés existent en open source et constituent un démarrage rapide.

Programmes scolaires. Le ministère de l'éducation publie les programmes officiels en PDF. Vous les convertissez en markdown, vous découpez par notion, vous obtenez une base sérieuse. Pour le bac philo, le repo github.com/scanderiaFR (que je maintiens) couvre déjà les 17 notions officielles du programme, citations vérifiées à l'appui.

Textes juridiques. Légifrance publie le code général des impôts, le code du travail, le code civil en libre accès. Un script de conversion en markdown structuré (un fichier par article) prend un après-midi. Vous obtenez un référentiel juridique exploitable en local.

Documentation technique. La plupart des SDK et des frameworks publient leur doc en markdown ou en RST sur GitHub. Cloner le dépôt vous donne une référence canonique pour le vibe - coding.

Wikipédia. Pour les notions générales, l'export markdown d'une page Wikipédia est une base correcte, à condition de la retravailler (Wikipédia est verbeuse, votre fiche doit être dense).

Le travail consiste à ***adopter*** ces corpus dans votre structure : les renommer, les indexer, leur ajouter vos propres compléments, les lier au reste. C'est de l'agrégation plus que de la création.

Le wiki au service du harness

C'est la boucle qui ferme le livret. Vos fiches markdown alimentent trois usages distincts, exactement les trois que ce livret a couverts.

Pour la maïeutique (chapitre 3) : le wiki conceptuel sert de mémoire externe au modèle qui joue le maître. Il sait quoi demander parce que vous lui avez préparé le terrain.

Pour le vibe-coding (chapitre 6) : les plugins de référence et les exemples de SDK servent à l'assistant de code qui génère vos nouveaux plugins. Il reproduit ce qu'il voit.

Pour le harness (chapitre 7) : les références inlinées dans le prompt système sont la matière première du harness. Sans elles, il n'amplifie rien.

Vous voyez la cohérence : le wiki n'est pas un usage de plus, c'est la fondation commune des autres. Ce n'est pas un hasard si je le mets en dernier dans le livret. C'est la pièce qu'on perçoit comme accessoire et qui se révèle centrale.

La promesse tenue

Refermez le livret et regardez le chemin parcouru. Vous êtes parti d'une question d'inquiet, «est-ce que ça va tourner chez moi?», et vous tenez maintenant de quoi y répondre : une méthode pour calibrer une machine, un LM Studio installé, un modèle qui tourne. Entre-temps, le local a cessé d'être une curiosité pour devenir un outil de tous les jours. Un maître socratique qui vous fait accoucher de vos réponses au lieu de les dicter. Un anonymiseur qui préfère manquer un nom plutôt que d'en inventer un, et qui ne laisse rien sortir de chez vous. Un modèle qui, branché sur un outil, va chercher dans le réel ce que ses poids ignorent.

Puis vous êtes passé de l'autre côté, du côté de celui qui fabrique. Un plugin vibe-codé en une soirée, parce que la barre est plus basse qu'on ne le croit. Un harness compris dans sa mécanique et, surtout, dans sa frontière : il amplifie un petit modèle, il ne le transforme pas en modèle frontière. Et cette matière markdown, ces fiches qu'on perçoit comme accessoires et qui se révèlent la fondation de tout le reste. Le fil qui relie ces huit chapitres est simple : en local, ce qui compte n'est jamais le modèle seul, c'est ce que vous mettez autour.

Le local est un travail long, plus que ce que la communication enthousiaste laisse entendre. Mais c'est aussi un travail dont les bénéfices se cumulent. Chaque fiche que vous ajoutez à votre référentiel reste utile dans un an. Chaque plugin que vous vibe-codez fonctionnera encore quand le prochain modèle frontière sortira. Chaque ingrédient de votre harness s'amortit sur tous les usages suivants.

Ce qui ne se cumule pas, c'est l'abonnement cloud. Il se renouvelle, indéfiniment, sans laisser de trace chez vous quand vous arrêtez. Le local est l'inverse exact. C'est la raison stratégique pour laquelle ce livret existe.

Maintenant, à vous.

Glossaire

Les termes techniques rencontrés au fil du livret, expliqués simplement et rassemblés ici pour s'y reporter à tout moment.

GPU

Graphics Processing Unit · processeur graphique

Le composant spécialisé dans les calculs massivement parallèles. C'est lui qui fait tourner un modèle d'IA rapidement, parce que l'inférence est avant tout de la multiplication de matrices, exactement le genre de calcul pour lequel un GPU est conçu. Sur un PC, c'est la carte graphique ; sur un Mac récent, le GPU est intégré à la puce. Voir aussi VRAM, mémoire unifiée.

VRAM

Video RAM · la mémoire de la carte graphique

La mémoire vive embarquée directement sur la carte graphique, juste à côté du GPU. Elle est séparée de la RAM système et bien plus rapide d'accès pour le GPU. Sur un PC à carte graphique dédiée, un modèle doit tenir dans cette VRAM pour tourner vite ; s'il déborde, il bascule dans la RAM système, beaucoup plus lente. Voir aussi mémoire unifiée.

RAM système

la mémoire vive de la carte mère

La mémoire vive principale de l'ordinateur, utilisée par le processeur central. Sur un PC à carte graphique dédiée, elle sert de réserve lente quand la VRAM est pleine : le modèle continue de tourner, mais la vitesse s'effondre. Voir aussi mémoire unifiée.

Mémoire unifiée

un seul pool de mémoire partagé (Apple Silicon)

Sur les puces Apple Silicon (M1 à M5), le processeur et le GPU partagent un seul et même pool de mémoire. Il n'y a pas de VRAM séparée : la RAM système ***est*** la mémoire GPU. C'est pourquoi un Mac à 16 Go peut parfois charger des modèles qu'une carte dédiée à 8 Go de VRAM refuse.

Paramètres (B)

les coefficients appris, comptés en milliards

Les valeurs numériques apprises par le modèle pendant son entraînement. On les compte en milliards : un modèle «8B» en a huit milliards (B pour ***billion***, milliard en anglais). Plus un modèle a de paramètres, plus il est capable, mais aussi plus il occupe de mémoire et plus il décode lentement. Voir aussi quantization, MoE.

Quantization

la compression des poids du modèle

Réduire la précision numérique des paramètres pour gagner de la mémoire. En FP16, chaque paramètre prend deux octets ; en Q4, environ un demi-octet. La Q4 perd quelques pour cent de qualité par rapport à la pleine précision, mais le gain mémoire est tel que c'est presque toujours le bon choix en local.

MoE (mixture-of-experts)

une architecture à experts spécialisés

Une architecture où seule une fraction des paramètres, les «experts», est active à chaque token. Le modèle stocke beaucoup de paramètres sur le disque mais n'en active qu'une partie à l'inférence : rapide à exécuter, lourd à stocker. C'est pourquoi un modèle MoE pèse plus que son nombre de paramètres actifs ne le laisse croire.

Token

l'unité de texte du modèle

Le morceau de texte élémentaire que manipule un modèle : un mot court, un bout de mot long, un signe de ponctuation. Le modèle lit et écrit token par token. La vitesse d'un modèle local se mesure en ***tokens par seconde***. Voir aussi contexte, KV-cache.

Contexte (fenêtre de contexte)

ce que le modèle garde sous les yeux

La quantité de texte que le modèle garde présente en mémoire pendant un échange : votre question, l'historique de la conversation, les documents que vous lui donnez. Elle se mesure en tokens. Plus la fenêtre est grande, plus le KV-cache consomme de mémoire.

KV-cache

la mémoire du contexte en cours

La mémoire que le modèle alloue pour traiter les tokens du contexte en cours (KV pour ***keys and values***). Elle grossit avec la taille de contexte : sur une grande fenêtre, le KV-cache peut peser autant que le modèle lui-même. C'est la raison pour laquelle agrandir le contexte coûte de la mémoire en plus du poids du modèle.

Scanderia

MASTERCLASS

Ce livret est un résumé de la formation live “Installez votre IA en local” avec Idriss Aberkane et Philippe Anel sur le média citoyen Scanderia fondé par Idriss Aberkane.

Rendez-vous sur scanderia.com pour visionner nos autres masterclasses, rester informé et développer votre esprit critique grâce au catalogue complet de Scanderia.

IDRISS ABERKANE, PH.D

Écrivain, essayiste, conférencier et consultant international.
Expert pluridisciplinaire reconnu mondialement et brillant vulgarisateur (biomimétisme, économie, géopolitique et intelligence artificielle).
Titulaire de trois doctorats en Littérature Comparée, en Neurosciences Cognitives et Économie de la Connaissance, ainsi qu’en Diplomatie et Noopolitique.

Ancien Visiting Scholar (Maths 2006) de l'Université de Stanford, Docteur de l'Ecole Polytechnique, il a été invité à donner plus de 700 conférences dans le monde, en trois langues et est l’auteur de plusieurs best-sellers.

Design graphique et mise en page : AMATHEUS Studio