

École polytechnique de Louvain

Small Language Models sur ordinateur personnel : analyse, prototypage et intégration dans un outil institutionnel

Auteur: **Lucas FUENTES GONZALEZ**

Promoteur: **Yves DEVILLE**

Lecteurs: **Yves DEVILLE, Benoit RONVAL, Nicolas HUBERTY**

Année académique 2025–2026

Master [60] en sciences informatiques

Remerciements

Je tiens tout d’abord à exprimer ma sincère gratitude à mon promoteur, le Professeur Yves Deville, pour son accompagnement tout au long de ce travail. Depuis septembre 2025, il m’a guidé avec patience, en proposant des pistes de réflexion décisives qui ont largement contribué à orienter ce mémoire. Je le remercie également pour le temps qu’il a consacré à relire ce travail avec attention, ainsi que pour m’avoir mis en contact avec les membres de l’équipe Piccolo.

Je remercie chaleureusement les membres de l’équipe Piccolo pour leur disponibilité et les échanges enrichissants qui ont nourri la réflexion du chapitre consacré à l’intégration institutionnelle.

Je remercie également Benoit Ronval et Nicolas Huberty pour avoir accepté d’être les lecteurs de ce mémoire et pour l’intérêt qu’ils ont porté à ce travail.

Pour finir, je souhaite remercier mes parents pour leur soutien indéfectible et leur aide tout au long de mes études. Ce travail leur doit beaucoup.

Abstract

Depuis l’adoption massive des assistants IA par les étudiants, les services cloud commerciaux dominants soulèvent des questions liées aux données personnelles transmises hors d’Europe, aux coûts d’abonnement créant des inégalités d’accès, aux sollicitations permanentes de *data centers* ainsi qu’une forte dépendance à des modèles non européens.

Ce mémoire conçoit et implémente une application IA locale, capable de s’exécuter sur l’ordinateur personnel d’un étudiant sans aucune connexion réseau. L’analyse comparative de l’écosystème open source des *Small Language Models* (SLMs) montre que les modèles de la tranche 1–4 milliards de paramètres, notamment Llama-3.2-3B et Gemma-2-2B, atteignent aujourd’hui des performances qui nécessitaient des architectures bien plus massives il y a peu de temps. Pour les faire tenir dans la mémoire d’un PC portable étudiant (8–16 Go de RAM), l’application s’appuie sur la quantification post-entraînement au format GGUF avec le profil Q4_K_M via *llama.cpp*, qui réduit l’empreinte mémoire d’un facteur quatre tout en préservant la majorité des capacités du modèle d’origine. Pour optimiser l’utilisation de la mémoire, l’application intègre trois mécanismes de gestion des ressources. À savoir, un filtrage matériel proactif qui n’expose à l’utilisateur que les modèles compatibles avec sa machine, un dimensionnement adaptatif du KV cache et une fenêtre glissante sur l’historique de conversation qui évite les crashes en cours d’usage prolongé.

Dans un second temps, le mémoire pose les bases d’une intégration avec Piccolo, la plateforme IA institutionnelle de l’UCLouvain. L’architecture hybride proposée définit des critères de délégation vers le cloud tel que la saturation matérielle et le besoin de raisonnement avancé en plus des stratégies de compression de contexte, de *graceful degradation* et de routage prédictif.

L’application démontre qu’il est techniquement viable d’offrir une alternative locale, gratuite et utilisable hors ligne, répondant ainsi directement aux défis de souveraineté numérique, d’égalité d’accès et d’empreinte environnementale liés aux services cloud. Les limites actuelles du projet, comme l’absence de support GPU explicite, le manque de RAG pour interroger ses documents et l’absence de multimodalité, dessinent les possibilités de développements futurs.

Table des matières

1	Introduction	1
2	Fondement technique	3
2.1	Des Transformers aux Small Language Models	3
2.2	L'enjeu de l'exécution locale	4
2.3	L'existant	5
3	Analyse de l'écosystème Open Source	7
3.1	L'évolution des LLMs locaux : des LLMs aux SLMs	7
3.2	Modèles Base vs Instruct	8
3.3	Analyse comparative des architectures et performances	9
3.4	Optimisation matérielle et efficacité calculatoire	10
3.5	Fiabilité et biais comportementaux	10
3.6	La place des modèles européens	11
3.7	Le rôle de HuggingFace et des « Open Weights »	11
4	Optimisation de l'Exécution Locale : La Quantification	13
4.1	Le principe mathématique de la quantification numérique	13
4.2	Méthodes de quantification : PTQ et QAT	15
4.3	Le paradoxe de la vitesse : surmonter le "Memory Wall"	16
4.4	Impact de la quantification sur les performances	16
4.5	L'écosystème logiciel : GGUF, llama.cpp et les K-Quants	18

5	Conception et Implémentation d'un cœur IA local	20
5.1	Architecture logicielle	20
5.2	Choix technologiques	21
5.3	Gestion dynamique et préventive des ressources	23
5.4	Défis d'intégration : Gestion de la concurrence et du Streaming	27
6	Vers une intégration dans Piccolo	29
6.1	Piccolo actuellement	29
6.2	Défis d'intégration et conception de l'API	29
6.3	Choix technologiques pour la communication réseau .	30
6.4	Critères de délégation : Quand basculer sur l'API? . .	30
6.5	Compression du contexte avant délégation	31
6.6	Graceful degradation	32
6.7	Sensibilisation aux crédits : vers une UX transparente	32
6.8	Mécanismes de routage	33
6.9	Le futur de Piccolo	34
7	Evaluation	35
7.1	Retour sur les objectifs initiaux	35
7.2	Positionnement par rapport à l'existant	36
7.3	Limites et perspectives d'amélioration	37
8	Conclusion	39

1 Introduction

Depuis l'arrivée de ChatGPT fin 2022, les modèles de langage sont entrés dans le quotidien d'une grande partie des gens, et entre autre dans celui des étudiants. Rédaction d'un mail, débogage d'un bout de code, résumé d'un cours, reformulation d'un paragraphe. Tous ces usages sont devenus banals en moins de trois ans. À l'UCLouvain comme ailleurs, une partie des étudiants utilise régulièrement un assistant IA, le plus souvent via des services cloud commerciaux comme ChatGPT, Mistral, Claude ou Gemini.

Cette adoption massive soulève plusieurs enjeux. Le premier est la confidentialité des données. Quand un étudiant soumet un brouillon, un travail non publié ou simplement des notes de cours à ChatGPT, ces données traversent l'Atlantique et peuvent servir à entraîner de futurs modèles. Pour des contenus personnels ou des recherches non terminées, c'est loin d'être anodin. Le deuxième enjeu est l'accessibilité. La plupart des services performants reposent sur un abonnement ou un quota de crédits, ce qui crée une inégalité d'accès entre étudiants. Le troisième est écologique. De la phase d'entraînement de ces architectures massives [36] jusqu'à leur exécution quotidienne, l'infrastructure logicielle sollicite des data centers particulièrement énergivores et la multiplication de ces requêtes par des millions d'utilisateurs accentue cette empreinte environnementale. Enfin, le quatrième est une question de souveraineté numérique car une grande partie des modèles les plus utilisés sont développés hors d'Europe, dans des cadres réglementaires éloignés du RGPD ou de l'AI Act.

Deux réponses parallèles se sont développées ces dernières années pour y répondre. D'un côté, une partie de l'industrie a progressivement détourné son attention de la course aux modèles toujours plus gros pour développer les Small Language Models (SLMs) [19, 42], des modèles conçus pour être efficaces plutôt qu'imposants. Ces modèles rendent désormais possible une exécution directement sur un ordinateur personnel, sans serveur distant. De l'autre, plusieurs institutions, dont l'UCLouvain, ont mis en place leurs propres plateformes IA. C'est le cas de Piccolo, qui sert de passerelle entre la communauté universitaire et des modèles puissants hébergés sur des serveurs, avec une attention particulière portée à la souveraineté, à la protection des données et à un usage responsable.

Ces deux réponses sont souvent présentées comme alternatives. Pourtant, ce travail part du principe qu'elles sont complémentaires. Un cœur IA local installé sur le PC de

l'étudiant peut absorber l'essentiel des usages quotidiens, tout en préservant totalement la confidentialité. Pour les tâches plus complexes, une délégation vers Piccolo permettrait d'accéder à des modèles plus puissants. C'est cette architecture hybride qui structure l'ensemble du projet, même si la partie d'intégration avec Piccolo restera à l'état conceptuel.

Ce mémoire se concentre principalement sur la première brique de cette architecture : la conception et l'implémentation d'un cœur IA local fonctionnel, accessible aux étudiant, et conçu dès le départ pour pouvoir coopérer avec une infrastructure cloud institutionnelle. Il faut d'abord identifier les modèles open source réellement exécutables sur le matériel étudiant, concevoir une architecture logicielle qui masque la complexité de l'inférence derrière une interface simple, et poser dès maintenant les bases de l'intégration avec Piccolo.

2 Fondement technique

Le terme « LLM local » est devenu courant ces dernières années, mais cette expression recouvre toute une mécanique technique qu’il faut comprendre pour saisir les enjeux du reste de ce travail.

2.1 Des Transformers aux Small Language Models

Un modèle de langage, dans sa version la plus simple, est un système probabiliste. Il prend en entrée un texte et prédit le *token* suivant le plus probable. Un *token* est l’unité de base manipulée par le modèle. Ça peut être un mot entier, un fragment de mot ou d’un caractère, selon la façon dont le tokenizer du modèle a été entraîné. La génération de texte se fait token par token, chaque nouveau token venant s’ajouter au contexte pour le calcul du suivant.

Ce mécanisme repose aujourd’hui presque entièrement sur une architecture introduite en 2017 : le *Transformer* [41]. Sa nouveauté est son mécanisme d’*attention* qui permet au modèle de relier directement n’importe quel token du contexte à n’importe quel autre, sans la limitation de mémoire des architectures précédentes. Grâce à ce mécanisme, le *Transformer* est massivement parallélisable¹, ce qui rend l’entraînement sur des milliards de paramètres techniquement viable.

C’est cette combinaison qui a permis l’apparition des *Large Language Models* comme GPT-3 et ses 175 milliards de paramètres en 2020, puis la course aux échelles toujours plus grandes qui a suivi. Mais cette course a fini par atteindre ses limites. Entraîner et faire tourner un modèle de cette taille demande des infrastructures cloud massives, ce qui rend l’usage local impossible et concentre la technologie entre les mains de quelques acteurs. La tendance s’est progressivement inversée. A partir de 2023-2024, l’industrie a commencé à privilégier des modèles plus petits mais plus efficaces, regroupés sous le terme de *Small Language Models* (SLMs) [19, 42].

La distinction entre LLM et SLM est avant tout une question de taille. Il n’existe pas de seuil officiel, mais on considère généralement qu’un SLM se situe sous la barre des 7 milliards [42]. Ces modèles, malgré leur taille réduite, atteignent désormais des

1. Contrairement aux architectures séquentielles qui traitent les mots l’un après l’autre, le mécanisme d’attention permet d’analyser l’intégralité d’une séquence simultanément. Les calculs peuvent ainsi être répartis sur des milliers de cœurs de processeurs (GPU), ce qui réduit drastiquement le temps d’entraînement.

performances comparables à celles des LLMs d'il y a peu de temps grâce à des techniques comme la distillation de connaissances ou encore la quantification [42]. Le chapitre 3 reviendra sur les modèles qui dominent aujourd'hui ce segment.

2.2 L'enjeu de l'exécution locale

Faire tourner un modèle de langage en local, c'est-à-dire directement sur la machine de l'utilisateur, sans passer par un serveur distant, n'est pas une simple question de préférence technique. Plusieurs motivations convergent pour justifier cette approche : la confidentialité des données qui ne quittent jamais la machine, l'accessibilité, l'indépendance vis-à-vis des fournisseurs cloud et la réduction de l'empreinte écologique liée aux centres de données. Comme l'introduction l'a déjà souligné, ces motivations sont aussi très concrètes. Pour un public étudiant, qui manipule régulièrement des documents personnels ou des travaux non publiés, la confidentialité n'est pas un luxe.

Le problème étant que faire tourner un LLM en local pose des défis techniques importants. Un modèle est composé de paramètres qui sont des nombres à virgule flottante stockés en mémoire pour permettre les calculs d'inférence. Plus le modèle est gros, plus il faut de mémoire vive pour le charger. Concrètement, un modèle de 8 milliards de paramètres encodés en 16 bits (le format standard dans l'industrie) demande environ 16 Go de RAM uniquement pour charger ses poids [3]. À cela s'ajoute la mémoire supplémentaire nécessaire pour stocker le contexte de la conversation. Sur un PC portable étudiant typique, équipé de 8 ou 16 Go de RAM partagés avec le système d'exploitation et le navigateur, cette contrainte est éliminatoire.

À cela s'ajoute la question de la vitesse. Générer du texte token par token implique de relire l'intégralité des paramètres du modèle pour chaque nouveau token produit. Sur un processeur grand public, cette opération est physiquement limitée par la bande passante mémoire de la machine, ce qui peut rendre l'expérience utilisateur particulièrement lente si rien n'est fait pour optimiser le processus.

Pourtant, ces dernières années, l'exécution locale est passée du statut d'expérimentation à celui de réalité accessible et ce grâce à la course aux SLMs déjà mentionnée ainsi qu'à la *quantification*, une technique de compression mathématique qui permet de réduire drastiquement l'empreinte mémoire d'un modèle sans dégrader significativement ses performances [17]. C'est le sujet du chapitre 4.

Ce sont ces deux facteurs combinés qui expliquent pourquoi un projet comme celui-ci

est devenu envisageable aujourd'hui alors qu'il ne l'était pas il y a quelques années.

2.3 L'existant

L'IA locale grand public n'est pas un terrain vierge. Plusieurs outils permettent déjà à un utilisateur de faire tourner un modèle sur sa propre machine, chacun avec sa philosophie, son public et ses compromis. Avant de présenter les choix de conception qui ont guidé ce travail, il est utile de présenter les solutions disponibles.

Les moteurs d'inférence avec API locale

La première catégorie regroupe les outils qui se concentrent sur le moteur d'inférence lui-même, sans interface utilisateur graphique intégrée. Ils sont conçus pour être appelés depuis du code, en ligne de commande ou via une API REST locale.

Le moteur d'inférence le plus connu est *Ollama* [25]. Il propose une syntaxe simple en ligne de commande pour télécharger et exécuter des modèles, et expose une API compatible avec celle d'OpenAI pour faciliter l'intégration dans d'autres outils. Son public cible est avant tout les développeurs qui veulent intégrer un LLM dans leurs propres projets. La force d'Ollama est sa simplicité d'usage pour ce public et la qualité de son écosystème de modèles préconfigurés.

À un niveau plus bas, *llama.cpp* [7] est la brique technique sous-jacente à beaucoup d'outils du marché, y compris à l'application développée dans ce travail (voir chapitre 5). C'est une bibliothèque C++ qui permet l'inférence efficace de modèles quantifiés sur du matériel grand public. *llama.cpp* peut être utilisée directement en ligne de commande, mais sa vocation principale est de servir de moteur à des couches logicielles supérieures.

Le point commun de ces outils est qu'ils ne s'adressent pas à un utilisateur final. Sans interface graphique, sans une grosse gestion des modèles, ils restent réservés à un public technique.

Les applications graphiques tout-en-un

La deuxième catégorie correspond aux applications de bureau qui combinent un moteur d'inférence avec une interface graphique soignée. L'objectif est de rendre l'IA locale accessible à un utilisateur qui ne veut pas ouvrir un terminal.

LM Studio [18] est probablement la solution la plus aboutie de cette catégorie. Elle propose une interface pour parcourir, télécharger et exécuter des modèles, avec un affichage de la consommation mémoire et des paramètres d'inférence. Le public visé est l'utilisateur curieux, à l'aise avec un ordinateur sans être nécessairement développeur. La principale limite de *LM Studio* est que c'est un logiciel propriétaire, ce qui empêche tout audit du code ou adaptation pour un contexte spécifique.

Jan [14] partage une philosophie similaire mais est *open source*. Il propose également une interface complète et embarque son propre moteur d'inférence. Sa principale différence avec *LM Studio* est le poids de son architecture, qui le rend plus lourd à installer et le destine plutôt à des utilisateurs déjà familiers de l'écosystème IA.

Les interfaces web auto-hébergées

La troisième catégorie rassemble les solutions pensées pour un déploiement de type serveur, accessibles via un navigateur web et souvent destinées à un usage multi-utilisateurs.

Open WebUI [26] en est le représentant le plus visible. Il propose une interface web complète, multi-utilisateurs, avec gestion des comptes, historique des conversations et système de plugins. Son installation typique passe par Docker et nécessite une base de données dédiée. Cette architecture en fait un excellent choix pour une équipe ou un département qui veut déployer un service LLM partagé en interne, mais elle est clairement surdimensionnée pour un usage individuel sur PC personnel.

AnythingLLM [23] suit une logique proche, avec une orientation plus marquée vers la création de *workspaces* privés intégrant du *Retrieval-Augmented Generation* sur des documents personnels. L'installation reste relativement lourde et le produit s'adresse plutôt à des utilisateurs qui veulent construire leur propre base de connaissances assistée par IA.

Bilan

Ces trois catégories couvrent l'essentiel du paysage actuel de l'IA locale. Les moteurs d'inférence ciblent les développeurs, les applications graphiques visent l'utilisateur final autonome, et les interfaces auto-hébergées s'adressent aux déploiements collectifs. Chacune de ces approches a ses qualités et répond à des besoins légitimes.

Pourtant, aucune de ces solutions n'est conçue spécifiquement pour le contexte qui motive ce travail c'est-à-dire un étudiant universitaire qui souhaite un assistant IA simple à installer, transparent dans son fonctionnement, et capable de coopérer à terme avec une infrastructure cloud institutionnelle.

3 Analyse de l'écosystème Open Source

L'apparition des modèles de langage de petite taille (Small Language Models - SLMs) marque un changement profond dans le domaine de l'intelligence artificielle générative. Historiquement, la performance des modèles était guidée par les lois d'échelle (*scaling laws*), qui établissaient qu'une augmentation massive du nombre de paramètres et des données d'entraînement garantissait une meilleure capacité cognitive mais la tendance s'inverse.

L'écosystème open source s'est récemment réorienté vers une optimisation radicale, visant à produire des modèles de petite taille capables d'être exécutés sur du matériel grand public, sans sacrifier leurs capacités de raisonnement [42]. Cette transition est une condition *sine qua non* pour le déploiement d'un cœur IA local sur des ordinateurs grand public, garantissant la confidentialité des données et l'accessibilité hors ligne.

En effet, le goulot d'étranglement majeur de l'inférence locale réside dans la mémoire vive (RAM ou VRAM). Pour qu'un modèle génère du texte rapidement, l'intégralité de ses poids doit être chargée en mémoire. Les modèles dépassant la barre des 7 milliards de paramètres requièrent des capacités qui dépassent largement celles des ordinateurs portables standards des étudiants. Sans l'application de techniques de compression ces modèles plus massifs et potentiellement plus performants demeurent inexploitable en local. C'est pourquoi cette analyse se concentre sur le segment *sub-7B*.

3.1 L'évolution des LLMs locaux : des LLMs aux SLMs

Le passage des LLMs aux SLMs ne se résume pas à une simple réduction de taille, il s'agit d'une optimisation de la densité d'information par paramètre [42]. Cette avancée s'explique par l'utilisation massive de techniques de *knowledge distillation*, où des modèles géants servent de "professeurs" pour transférer leurs capacités de raisonnement vers des architectures plus légères.

Les résultats de benchmarks récents montrent que des modèles comme Llama 3.2 3B

atteignent 63,4% sur MMLU² [42], un niveau qui nécessitait des architectures bien plus massives en 2022 [19].

3.2 Modèles Base vs Instruct

En ce qui concerne les modèles locaux, le choix des modèles pour ce projet ne s'est pas porté sur des modèles de base (*Base models*), mais sur leurs versions alignées (*Instruct/Chat models*). Cette distinction est fondamentale dans le monde des LLMs.

Alors que les modèles de base sont entraînés pour prédire statistiquement le token suivant à partir de grands ensembles textuels[27], les modèles *Instruct*³ passe par une phase supplémentaire d'alignement, souvent via le *Reinforcement Learning from Human Feedback*⁴, pour répondre aux intentions de l'utilisateur. Une étude fondamentale a démontré que l'alignement a un impact majeur : "In human evaluations on our prompt distribution, outputs from the 1.3B parameter InstructGPT model are preferred to outputs from the 175B GPT-3, despite having 100x fewer parameters."⁵ [27]. Cela justifie l'utilisation de SLMs « Instruct » pour obtenir un assistant performant sur du matériel grand public.

Cependant, cette transition vers les modèles *Instruct* impose des méthodes d'évaluation différentes. Les benchmarks classiques (MMLU, ARC⁶) mesurent les connaissances brutes, mais restent limités pour capturer la qualité de l'interaction avec les utilisateurs[27]. L'écosystème s'oriente donc vers des cadres comme **MT-Bench**, qui utilisent le concept de **LLM-as-a-judge** : des modèles plus grands et plus puissants (comme GPT-4) servent d'arbitres. Ils sont utilisés pour évaluer la pertinence et la cohérence des réponses des SLMs sur des conversations. [45].

Il est crucial de noter que ces deux approches d'évaluation sont complémentaires. Les benchmarks de base comme MMLU définissent le « potentiel cognitif » du modèle (son

2. Massive Multitask Language Understanding : Évalue les connaissances générales et le raisonnement du modèle à travers des questions à choix multiples couvrant 57 domaines de compétences.

3. Les modèles "Instruct" (ou "Chat") correspondent à la version conversationnelle des LLMs. Ils sont spécifiquement formatés pour interagir sous forme de dialogue fluide avec l'utilisateur, à l'image des LLMs grand public comme ChatGPT, Gemini ou Mistral.

4. RLHF : Apprentissage par renforcement à partir de retours humains

5. "Lors d'évaluations humaines, sur notre distribution de réponses rapides, les résultats du modèle InstructGPT à 1,3 milliard de paramètres sont préférés aux résultats du GPT-3 à 175 milliards de paramètres, malgré le fait qu'il possède 100 fois moins de paramètres."

6. AI2 Reasoning Challenge : Évalue la capacité de raisonnement scientifique à l'aide de questions à choix multiples tirées d'examens de sciences.

socle de connaissances brut et sa capacité de raisonnement) tandis que les benchmarks de conversation évaluent la manière dont ce potentiel est utilisé dans un dialogue. De plus, le maintien d’une évaluation sur les modèles de base permet de surveiller la « taxe d’alignement »⁷ (*alignment tax*)[27], permettant de s’assurer que l’optimisation pour la conversation n’a pas dégradé les capacités de raisonnement du modèle. Ainsi, bien que le projet vise une interaction fluide, les scores MMLU et ARC-Challenge, présentés dans les sections suivantes, restent des indicateurs indispensables du plafond de verre des performances atteignables par chaque architecture.

3.3 Analyse comparative des architectures et performances

L’analyse de l’écosystème montre une hiérarchie claire en fonction des capacités cognitives et de l’efficacité de l’architecture logicielle.

Leaders de la performance globale

Selon les benchmarks récents, la famille **Llama-3.2** de Meta constitue désormais le point de référence pour les modèles locaux actuels [42] :

- **Llama-3.2-3B** : Il s’impose comme le leader du segment sub-5B avec des scores de 63,4% sur *MMLU* et 78,6% sur *ARC-Challenge*. Ces résultats soulignent une capacité de raisonnement multidisciplinaire supérieure à la moyenne pour un modèle de cette taille.
- **Llama-3.2-1B** : Il donne des résultats surprenants vu sa taille. Sur *MMLU*, il gagne environ 20 points par rapport à *OLMo 1.2B*, un modèle pourtant comparable en nombre de paramètres. Dans le benchmark *SLM-Bench*, c’est aussi lui qui accumule le plus de premières places en correctness.

D’autres modèles, tels que **Gemma-2 (2B)** de Google, démontrent une capacité de généralisation remarquable avec un score de 57,8% sur *MMLU* et 76,7% sur *ARC*. Ses performances approchent celles de modèles 3B malgré un milliard de paramètres en moins, ce qui en fait un candidat sérieux [42].

7. désigne la perte potentielle de performances brutes ou de capacité de raisonnement d’un modèle qui survient lorsqu’on le contraint, via l’entraînement RLHF, à devenir plus poli ou plus sécurisé.

Spécialisation et raisonnement mathématique

Au-delà des connaissances générales, la logique et les mathématiques restent un vrai défi pour les SLMs. Les benchmarks classiques comme GSM8K montrent que le raisonnement en plusieurs étapes bloque rapidement les petits modèles. À ce sujet, une étude publiée à l'ACL apporte un éclairage très intéressant [13] : les SLMs ont souvent les connaissances nécessaires pour répondre, mais à cause de leur fonctionnement auto-régressif⁸, s'ils ratent la toute première étape du calcul, l'erreur se propage et fausse tout le reste du raisonnement. À l'inverse, si on les aide à bien formuler ce premier pas, leurs performances grimpent en flèche [13].

3.4 Optimisation matérielle et efficacité calculatoire

L'expérience utilisateur dépend directement du *throughput* (vitesse d'inférence).

- **Efficacité brute** : **GPT-Neo-1.3B** demeure une référence pour maximiser le débit par rapport aux ressources consommées, obtenant les meilleurs scores en *computation* dans les benchmarks de [28].
- **Innovations architecturales** : Le modèle **TinyLlama-1.1B** montre l'apport des optimisations comme *FlashAttention-2*⁹ et la *Grouped-query attention* (GQA) [44]. La GQA est particulièrement importante car elle réduit l'utilisation de la mémoire du KV Cache¹⁰, ce qui permet de gérer des contextes longs sur les machines des étudiants disposant de peu de RAM.

3.5 Fiabilité et biais comportementaux

Une étude approfondie sur les questions binaires révèle des vulnérabilités propres aux SLMs (1B–5B) qu'il est nécessaire de prendre en compte [37] :

- **Biais de validation** : Llama-3.2-1B présente un fort biais "TRUE" (probabilité ≈ 0.71) dans les formulations de type *True-First*, tendant à valider systématiquement l'affirmation présentée en premier.

8. Un modèle auto-régressif génère le texte mot par mot (ou token par token) : chaque nouveau mot produit est prédit en s'appuyant sur l'ensemble des mots précédemment générés.

9. optimise la parallélisation des calculs, accélérant considérablement l'inférence sur les contextes longs

10. Mécanisme de la mémoire interne du modèle permettant de stocker les calculs liés aux mots précédents de la conversation. Cela évite à l'IA de devoir recalculer tout l'historique à chaque nouveau mot généré.

- **Instabilité linguistique** : Phi-3.5-mini montre une forte variabilité multilingue ; s'il est performant en anglais ou allemand, ses réponses deviennent mauvaises ou incohérentes en polonais ou en afrikaans.

Ces exemples montrent que les modèles ont quand même certaines limitations dues à leur plus petite taille.

3.6 La place des modèles européens

Un constat frappant émerge de la littérature et des benchmarks récents : la sur-représentation des modèles américains (Llama de Meta, Gemma de Google) et asiatiques (Qwen d'Alibaba) au détriment des initiatives européennes. Bien que des acteurs européens de premier plan, comme EuroLLM ou encore le français Mistral AI, développent des modèles ouverts très performants (tel que Mistral-7B), l'écosystème autour des modèles de très petite taille reste dominé par les géants hors Europe.

Pour un projet comme celui-ci au sein de l'UCLouvain, se tourner vers des modèles développés en Europe a pourtant une importance stratégique. Au-delà des performances, l'utilisation de modèles "locaux" favorise la souveraineté numérique et l'indépendance aux technologies extra-européennes. De plus, les modèles européens sont conçus en intégrant une meilleure prise en compte culturelle propre à nos régions et une meilleure maîtrise des langues européennes. Cela permettrait de pallier directement les problèmes d'instabilité multilingue observés sur certains des modèles [37], tout en s'assurant que les modèles ont été conçus en respectant les règles européennes tels que le RGPD et l'AI Act. L'intégration future de modèles SLMs européens constitue donc une piste pour le projet et c'est pourquoi ce travail reste ouvert à leur intégration future, dès que des versions GGUF de qualité équivalente seront disponibles.

3.7 Le rôle de HuggingFace et des « Open Weights »

La plateforme **HuggingFace** joue un rôle clé dans l'écosystème de l'IA local en donnant accès aux modèles **Open Weights** [11]. Contrairement aux modèles "boîte noire" (API propriétaire), ces modèles sont cruciaux car ils permettent :

1. **L'audit de sécurité** complet et la vérification transparente des données.
2. **La quantification personnalisée** (ex : format 4-bit GGUF) pour optimiser les modèles en fonction du matériel des étudiants.

3. **L'évaluation via l'Open LLM Leaderboard** pour sélectionner les modèles les plus stables basés sur les retours de la communauté. [12].

Rank	Type	Model
1768		meta-llama/Llama-3.2-3B-Instruct 
1776		mistralai/Mistral-8B-Instruct-2410 
1905		meta-llama/Llama-3.1-8B-Instruct 
3089		stabilityai/stablelm-2-12b-chat 
3366		meta-llama/Llama-3.2-1B-Instruct 
3506		google/gemma-7b-it 
3524		mistralai/Mistral-7B-Instruct-v0.1 
4002		google/gemma-2b-it 
4077		rasyosef/Phi-1_5-Instruct-v0.1 
4236		EleutherAI/gpt-neo-1.3B 

FIGURE 1 – Hugging Face - LeaderBoard de la Communauté des modèles précédemment cités

Il est important de préciser que l'Open LLM Leaderboard compare l'intégralité des modèles open source disponibles, y compris des architectures massives dépassant la centaine de milliards de paramètres (comme Llama-3.1-405B). C'est pourquoi, sur la figure ci-dessus, les SLMs de cette analyse se positionnent à partir du rang 1768. De plus, les classement dans le Leaderboard peuvent changer avec le temps.

4 Optimisation de l'Exécution Locale : La Quantification

Faire tourner un LLM en local, surtout sur les ordinateurs des étudiants où le matériel est très variable et souvent limité, pose un problème évident : ces modèles sont extrêmement gourmands en ressources [34]. À la base, ils sont pensés pour être déployés sur des serveurs. Dans leur format natif, les poids du réseau de neurones sont stockés avec une grande précision. Historiquement, on utilisait du 32 bits (FP32)¹¹, mais aujourd'hui l'industrie s'est standardisée sur le 16 bits (FP16) [17].

Cela représente 2 octets par paramètre. Concrètement, un modèle standard de 8 milliards de paramètres nécessite mathématiquement 16 Go de mémoire vive (RAM ou VRAM) uniquement pour être chargé en mémoire. À cela s'ajoute la mémoire supplémentaire nécessaire pour gérer le contexte de la conversation. Évidemment, ces prérequis dépassent largement les capacités d'un PC portable classique [3]. Pour contourner cette contrainte matérielle, le projet s'appuie sur la quantification (*quantization*) [17, 34]. C'est une technique algorithmique de compression essentielle car c'est elle qui permet de réduire drastiquement le poids des LLMs pour les rendre compatibles avec des machines personnelles.

4.1 Le principe mathématique de la quantification numérique

Concrètement, l'idée de la quantification est de réduire la précision numérique des poids du modèle. Au lieu de manipuler des nombres à virgule flottante très lourds (en FP16), on vient les projeter sur des valeurs entières de plus faible précision, généralement sur 8 bits (INT8) ou même 4 bits (INT4). L'objectif est simple : garder une approximation fidèle de la distribution d'origine, tout en divisant l'occupation en mémoire.

La quantification affine : de la théorie à la pratique

D'un point de vue mathématique, la méthode la plus documentée est la quantification affine (aussi appelée asymétrique) [17]. Le principe est de prendre une plage continue de nombres décimaux et de la projeter sur une grille d'entiers beaucoup plus restreinte. Pour une quantification sur 8 bits par exemple, notre grille d'arrivée ne comporte que $2^8 = 256$ valeurs possibles (de 0 à 255).

11. Single-precision floating-point format

Pour que cette projection fonctionne sans modifier le modèle, il faut conserver deux paramètres clés pour chaque bloc de poids :

- Le **facteur d'échelle** (*Scale* ou S) : il sert à "compresser" la plage de nos données de départ pour qu'elle corresponde exactement à la nouvelle grille.
- Le **point de référence** (*Zero-point* ou Z) : c'est un entier qui permet de s'assurer que le "0" décimal correspond pile à une valeur entière dans le format quantifié. Ce détail est crucial, si le zéro est décalé, le modèle accumule des erreurs à chaque calcul matriciel et finit par générer des réponses incohérentes.

Prenons un exemple concret. Imaginons un ensemble de poids neuronaux qui s'étalent d'un minimum $\beta = -1,2$ à un maximum $\alpha = 1,8$. Pour calculer le facteur d'échelle S , on divise simplement la taille de la grille (255) par l'écart total de nos données :

$$S = \frac{255}{1,8 - (-1,2)} = \frac{255}{3,0} = 85,0$$

Ensuite, il faut trouver le point de référence Z pour aligner notre valeur réelle la plus basse $(-1,2)$ avec le zéro de la grille. En utilisant la formule de transformation affine¹² $x_q = x \times S + Z$, cela donne :

$$0 = -1,2 \times 85,0 + Z \implies Z = 102$$

Si l'on souhaite quantifier un poids précis du réseau, par exemple $x = -0,5$, le programme applique la formule :

$$x_q = -0,5 \times 85,0 + 102 = 59,5$$

Comme notre format d'arrivée ne gère que les nombres entiers, cette valeur est forcément arrondie à 60. C'est exactement cet arrondi imposé qui crée ce que l'on appelle l'erreur de quantification.

Plus tard, lors de l'inférence, le processeur effectue l'opération inverse : c'est la *dé-quantification*. Il récupère l'entier 60, réutilise S et Z , et tente de reconstituer le nombre d'origine :

$$x \approx \frac{60 - 102}{85,0} = -0,494$$

12. où x_q est la valeur quantifiée et x la valeur d'origine

On observe une légère perte de précision (la différence entre $-0,5$ et $-0,494$). Sur un seul paramètre, c'est négligeable. Mais multipliée par des milliards de poids dans le réseau, cette petite erreur peut dégrader les performances. Gérer et minimiser cette perte constitue justement tout l'enjeu des choix algorithmiques actuels.

Valeurs aberrantes et granularité

Le souci, c'est que si l'on appliquait un unique facteur d'échelle S à l'ensemble des poids d'une couche (*Per-layer quantization*), cela poserait rapidement problème. Il suffirait d'un seul poids avec une valeur extrême (*outlier*) pour étirer complètement l'amplitude $\alpha - \beta$. Résultat, tous les petits poids qui tournent autour de 0 finiraient écrasés sur la même valeur quantifiée, ce qui détruirait la finesse du modèle.

Pour éviter cela, les architectures modernes privilégient une granularité beaucoup plus fine, comme la quantification par bloc (*Block-wise quantization*). L'idée est de découper les matrices en petits groupes de 32, 64 ou 256 poids. Comme chaque bloc conserve ses propres métadonnées (S et Z), on arrive à préserver très fidèlement la distribution locale des informations.

En plus des poids, il faut également gérer les activations¹³. Le défi ici, c'est que les activations changent selon la requête de l'utilisateur et sont très sensibles aux valeurs aberrantes. C'est pourquoi plusieurs moteurs d'inférence font un choix stratégique : ils compressez agressivement les poids (en 4 bits) mais gardent les activations sur une précision un peu plus haute (en 8 bits ou en FP16). Cela évite que l'IA ne perde le fil de son raisonnement en plein milieu d'une phrase.

4.2 Méthodes de quantification : PTQ et QAT

En pratique, la quantification peut s'appliquer à différents moments. La première option, c'est la Quantification Post-Entraînement (*Post-Training Quantization* - PTQ) [17]. Le principe est assez direct : on prend un modèle qui a déjà été entraîné en FP16, et on vient simplement compresser ses poids après coup. L'avantage, c'est que c'est extrêmement rapide et que cela demande peu de puissance de calcul. C'est l'approche standard pour la majorité des modèles open source.

13. Résultats intermédiaires calculés par les neurones

La deuxième option, c'est l'entraînement Sensible à la Quantification (*Quantization-Aware Training* - QAT). Là, le processus est beaucoup plus lourd. L'idée est de simuler les défauts de la compression directement pendant la phase d'entraînement. Même si cette méthode permet de descendre à des niveaux de compression extrêmes (2 ou 3 bits) avec de très bonnes performances, elle demande énormément de ressources matérielles et de temps. Étant donné que ce type de quantification est peu présent dans les modèles open source, l'approche PTQ a logiquement été retenue dans le cadre de ce travail.

4.3 Le paradoxe de la vitesse : surmonter le "Memory Wall"

Intuitivement, on pourrait se dire que rajouter une étape de dé-quantification (la conversion de INT vers FP avant chaque opération matricielle) va forcément ralentir le programme. Pourtant, c'est l'inverse. La quantification accélère énormément l'inférence. L'explication se trouve dans un problème matériel bien connu : le fameux *Memory Wall*.

Quand on fait tourner un LLM, l'étape la plus lente n'est pas le calcul brut à l'intérieur du processeur (CPU). Le vrai point de blocage, c'est le trajet des données entre la RAM et le processeur. Pour générer un seul token, le système doit lire la quasi-totalité des milliards de paramètres du modèle. Or, cette vitesse de transfert est physiquement bridée par la bande passante de la carte mère. En divisant la taille du modèle par 4, on divise par 4 le volume de données qui doit traverser ce goulot d'étranglement. Finalement, le petit délai causé par la conversion mathématique est largement compensé par le gain de temps obtenu sur le transfert physique. C'est ce mécanisme qui permet d'avoir une génération fluide sur des PC classiques.

4.4 Impact de la quantification sur les performances

Une question légitime se pose : est-ce que réduire aussi drastiquement la taille du modèle ne va pas dégrader son raisonnement ? Les recherches sur les lois d'échelle de l'inférence (*Inference Scaling Laws*) [3] sont plutôt rassurantes. Pour mesurer cela, on observe souvent la *cross-entropy*. Concrètement, elle évalue à quel point le modèle est sûr de lui quand il doit prédire le mot suivant. Si l'entropie grimpe en flèche, c'est le signe d'une IA confuse qui se met à halluciner.

En pratique, la littérature scientifique [17] montre que le passage en 8 bits permet de conserver plus de 99% des performances de base. Les travaux fondateurs sur les lois d'échelle de l'inférence [3] établissent que le compromis idéal (sweet spot) se situe

autour de 4 bits [3, 17], préservant la quasi-totalité de la justesse du modèle d’origine tout en divisant son empreinte par quatre. En revanche, dès que l’on descend à 3 bits, la mécanique atteint un point de rupture empirique où l’entropie décroche nettement [3].

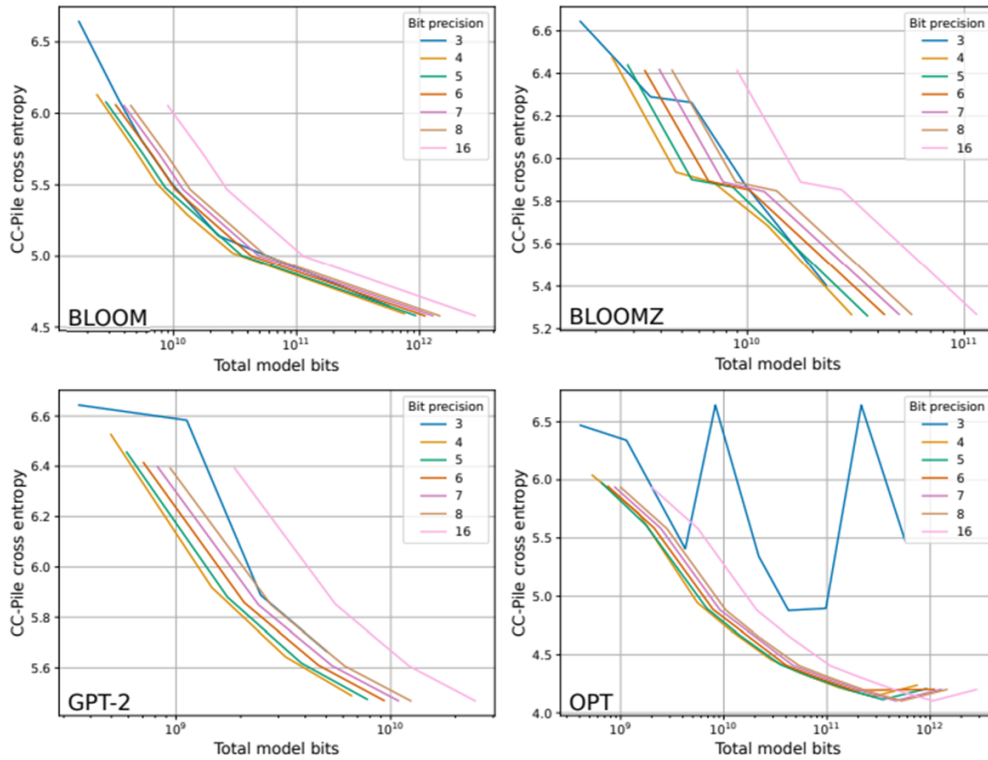


FIGURE 2 – Lois d’échelle de l’inférence : Comparaison des performances en fonction de la taille du modèle et de la précision des poids pour une empreinte mémoire constante [3].

La Figure 2 montre bien ce compromis. Elle compare quatre familles de modèles open source (BLOOM, BLOOMZ, GPT-2 et OPT) sur la sous-partie *CommonCrawl* du dataset *The Pile*¹⁴. L’axe des abscisses représente la taille totale du modèle en bits (échelle logarithmique), ce qui permet de comparer différentes architectures à empreinte mémoire équivalente. L’axe des ordonnées affiche la *cross-entropy* obtenue sur ce corpus (notée « CC-Pile » sur le graphique, pour *CommonCrawl-Pile*). Plus la valeur est basse, meilleure est la capacité prédictive du modèle. Chaque courbe correspond à un niveau

14. *The Pile* est un corpus textuel libre de 800 Go couramment utilisé pour l’entraînement et l’évaluation des grands modèles de langage. Il agrège 22 sources hétérogènes, dont la sous-partie *CommonCrawl* (une archive publique du web) utilisée ici comme benchmark d’évaluation.

de précision différent (de 3 à 16 bits par poids), ce qui permet de visualiser directement l'impact de la quantification.

Ces courbes montrent qu'à empreinte mémoire égale, les modèles quantifiés entre 4 et 6 bits se comportent presque comme ceux en 8 ou 16 bits. Seule la courbe en 3 bits décroche nettement, ce qui confirme le point de rupture sous les 4 bits.

Un nouveau paradigme de sélection

Ces observations amènent à une véritable règle d'or pour l'exécution locale : à quantité de RAM allouable égale, la taille globale du modèle compte bien plus que sa précision mathématique pure [3].

Prenons un exemple concret, illustré par la Figure 2. Si l'ordinateur d'un étudiant ne dispose que de 8 Go de RAM, deux options s'offrent à lui. Soit il charge un petit modèle de 3 milliards de paramètres sans compression (en FP16). Soit il opte pour un gros modèle de 12 milliards de paramètres, mais fortement quantifié (en INT4).

Les tests prouvent systématiquement que le modèle de 12B en 4 bits surpasse largement celui de 3B en 16 bits, que ce soit pour la logique ou la rédaction. Finalement, l'idée n'est pas de sacrifier les performances pour s'adapter au matériel, mais plutôt d'arriver à intégrer le maximum "d'intelligence" dans chaque octet de mémoire disponible.

4.5 L'écosystème logiciel : GGUF, llama.cpp et les K-Quants

Il existe pas mal d'autres méthodes de quantification très performantes, comme le *Generalized Post-Training Quantization* (GPTQ) ou le *Activation-Aware Weight Quantization* (AWQ) [17]. Le souci, c'est que ces algorithmes sont conçus pour tourner presque exclusivement sur des cartes graphiques (GPU) dédiées. Le matériel des étudiants étant composé majoritairement de CPU classiques, ces formats n'étaient pas adaptés au projet.

Pour contourner ces limites matérielles, l'architecture s'appuie sur la bibliothèque `llama.cpp` [7] et exploite les modèles au format GGUF¹⁵ [6, 10]. C'est un format « tout-en-un » très pratique qui embarque absolument tout ce qu'il faut pour l'inférence [10]. De plus, il permet à l'OS de charger le modèle en RAM de façon quasi instantanée en utilisant le *Memory Mapping* (mmap)¹⁶.

15. GGML Universal File

16. mmap est un appel système qui mappe un fichier directement dans l'espace d'adressage virtuel. Cela

Au sein du format GGUF, il a fallu choisir un profil de quantification précis. Puisque le 4 bits représente le compromis idéal (*sweet spot*) entre consommation mémoire et justesse, il restait à déterminer de quelle manière l'appliquer. Le choix s'est finalement porté sur la famille des K-Quants, et plus précisément sur le profil Q4_K_M [34].

Pour comprendre ce choix, il faut comparer cette méthode aux anciens formats standards (comme le profil *Legacy* Q4_0). Les méthodes classiques appliquent une réduction linéaire très basique sur de petits blocs de 32 paramètres [34]. C'est rapide, mais la distribution mathématique d'origine est fortement écrasée. Les K-Quants, à l'inverse, s'appuient sur une structure hiérarchique en « super-blocs » de 256 paramètres, eux-mêmes divisés en sous-blocs de 32 poids disposant chacun de leur propre facteur d'échelle [7, 34]. Pour limiter le coût en métadonnées, les facteurs d'échelle des sous-blocs sont eux aussi quantifiés à partir d'une super-échelle stockée au niveau du super-bloc [34]. Cette organisation permet de respecter beaucoup mieux la courbe naturelle des poids du modèle qu'un simple découpage linéaire.

Ensuite, la déclinaison « Medium » (_M) a été préférée à la déclinaison « Small » (_S). Là où un profil Q4_K_S force l'intégralité du réseau en 4 bits purs pour économiser un maximum d'espace, le paramètre « Medium » applique une stratégie hybride [7]. La grande majorité des poids est bien compressée sur 4 bits, mais les zones les plus sensibles du réseau sont sauvegardées sur 5 ou 6 bits [10, 34]. C'est précisément cette protection ciblée des couches critiques qui permet de faire tourner de plus gros modèles récents sans ressentir de perte de « logique » à l'usage.

Enfin, l'écosystème continue d'évoluer, notamment avec l'arrivée récente des IQ-Quants [34]. Toutefois, cette approche n'a pas été retenue pour le projet. En effet, encore peu de modèles *open source* sont distribués sous ce format à ce jour, ce qui réduit considérablement le nombre de modèles disponibles.

permet au système d'exploitation de charger dynamiquement les données uniquement lorsque le processeur en a besoin.

5 Conception et Implémentation d'un cœur IA local

L'un des plus gros défis techniques du projet a été de masquer la complexité de l'exécution locale des SLM, pour pouvoir offrir à l'utilisateur une interface qui reste fluide et réactive. Habituellement, avec des solutions basées sur des API comme OpenAI ou Claude, la puissance de calcul est totalement déportée sur le cloud. À l'inverse, l'inférence en local utilise directement les ressources de la machine hôte. Comme ces ressources matérielles sont limitées, cela exige une gestion beaucoup plus fine et dynamique. Ce chapitre détaille donc l'architecture logicielle qui a été mise en place. Il présente les choix technologiques adoptés pour contourner ces contraintes, ainsi que les solutions développées pour bien gérer la mémoire et les accès concurrents. Il est à noter que le lien [github](#) permettant d'accéder à l'intégralité du code source de cette implémentation se trouve en annexe ainsi que des screenshot du prototype.

5.1 Architecture logicielle

Au lieu de tout coder dans un seul grand fichier monolithique, l'application a été découpée en plusieurs parties pour que le code reste lisible et plus facile à modifier par la suite. L'idée était de bien séparer les rôles de chaque élément (en suivant le principe de *Separation of Concerns*)¹⁷. Concrètement, le code est divisé en quatre modules principaux qui échangent des informations grâce à un état global centralisé.

Le Singleton pour la gestion de l'état global

Comme la génération de texte par le modèle peut prendre un certain temps, il fallait éviter les bugs ou les blocages si l'utilisateur interagit avec l'interface au même moment. Pour gérer cela, le fichier `state.py` s'inspire du design pattern *Singleton* [33]. Il sert à centraliser les gestionnaires principaux (*Engine* et *Chat Manager*) pour qu'ils ne soient instanciés qu'une seule fois. Il maintient aussi un dictionnaire avec l'état actuel de l'application : on y retrouve des informations comme l'identifiant de la conversation en cours, le modèle chargé en mémoire, et surtout un booléen qui sert de verrou (`is_generating`). Ce verrou empêche tout simplement de lancer une nouvelle

17. Principe fondamental d'architecture logicielle consistant à diviser un programme en modules distincts, chacun ayant une responsabilité unique et bien définie.

requête quand le programme est déjà en train d'en traiter une, ce qui évite les problèmes de concurrence.

La persistance (Chat Manager)

Puisque l'un des objectifs principaux du projet est de garantir la confidentialité, toutes les données devaient être sauvegardées localement, sans rien envoyer sur le réseau. Pour stocker les historiques de discussion, le format JSON a été privilégié par rapport à une base de données classique (comme SQLite). Ce choix est assez naturel quand on regarde les données : les conversations avec un LLM sont déjà structurées sous forme de listes de messages (avec les rôles *system*, *user*, et *assistant*), ce qui correspond parfaitement au fonctionnement du JSON. C'est simple à lire, à sauvegarder, et ça consomme peu de ressources. Enfin, pour que l'application s'installe proprement sur n'importe quel ordinateur, le code va chercher dynamiquement le dossier standard des données d'application de l'OS (comme *AppData*) pour y ranger les historiques.

5.2 Choix technologiques

Développer un cœur IA demande de faire le lien entre deux mondes : d'un côté, celui de la recherche en IA, qui utilise beaucoup Python avec de grosses bibliothèques comme PyTorch ; de l'autre, celui du déploiement local, qui exige plutôt du code bas niveau bien optimisé.

L'interface Llama.cpp : le pont Python/C++

Comme on l'a vu dans la section précédente sur la quantification, il est intéressant d'utiliser des modèles quantifiés au format GGUF [5, 6, 10]. Pour gérer ce format en pratique, l'application s'appuie sur la bibliothèque *llama-cpp-python*. Cette dépendance a deux grands avantages. D'abord, elle offre des *bindings*¹⁸ vers la bibliothèque native *llama.cpp* codée en C/C++. Cela permet de s'assurer que les gros calculs matriciels, indispensables pour générer les *tokens*, tournent avec les performances d'un langage compilé. Ensuite, elle propose une interface de haut niveau très standard. C'est un vrai

18. Code servant de "passerelle" pour permettre à un programme écrit dans un certain langage (ici Python) d'appeler et d'utiliser une bibliothèque écrite dans un autre langage plus bas niveau (ici C/C++).

confort pour le développement de l'interface, car cela évite d'avoir à gérer manuellement la complexité des pointeurs et de la mémoire en C++.

Réactivité asynchrone et interface Web locale

Pour l'interface graphique (UI), il a fallu bien comparer les options disponibles. De nombreuses architectures d'interface classiques reposent par défaut sur des boucles d'événements synchrones. Le souci, c'est que si on les associe à des processus d'inférence IA, l'interface finit inévitablement par geler (*freeze*) pendant que le modèle calcule sa réponse.

Il existe de nombreuses solutions en Python spécialement conçues pour l'IA, mais elles posaient majoritairement problème :

- **Chainlit** : C'était le premier choix de développement car il gère l'asynchrone de base, ce qui évite de figer l'écran pendant les requêtes. Le souci, c'est son côté "boîte noire" : c'était beaucoup trop rigide pour faire des personnalisations poussées. De plus, il fallait mettre en place des contournements techniques assez lourds afin d'éviter que le modèle ne se recharge complètement en mémoire à chaque nouveau message [2].
- **Streamlit et Gradio** : Streamlit a un gros défaut pour faire tourner un modèle en local : le moindre changement sur l'interface relance la totalité du script. Quant à Gradio, bien que très simple à utiliser, il a été écarté car le rendu final n'était vraiment pas assez ergonomique [8, 35].
- **Open WebUI et Chatbot UI** : Ce sont des solutions "full stack" très complètes, mais qui s'apparentent à de vraies usines à gaz pour un projet de cette taille. Elles demandent d'installer des bases de données (MongoDB ou Supabase avec Docker) et divers outils externes. En pratique, c'est pensé pour être déployé sur des serveurs d'entreprise, pas du tout pour une exécution locale sur le PC d'un étudiant [26].

Le choix s'est finalement porté sur NiceGUI [43], découvert en cherchant des alternatives légères. C'est un framework construit par-dessus *FastAPI* et *Vue.js*. C'est une solution hybride qui génère une interface web locale. Son gros point fort, c'est sa flexibilité : comme NiceGUI n'impose aucun design préconçu, il laisse une liberté totale.

Autre avantage crucial : comme tout repose sur *asyncio* [29], chaque interaction (un

clic, du texte tapé) lance des coroutines asynchrones¹⁹. L'interface communique avec le backend Python via des WebSockets²⁰. Ça permet de mettre à jour seulement certaines parties de l'écran en temps réel, sans jamais avoir à recharger la page complète. C'est indispensable pour afficher la réponse de l'IA mot par mot de façon bien fluide, tout en gardant le modèle LLM chargé en tâche de fond.

5.3 Gestion dynamique et préventive des ressources

Toute la complexité de cette implémentation se concentre vraiment dans le module `llm_engine.py`. En effet, contrairement à un programme classique, un modèle d'IA ne peut pas se rabattre sur le fichier d'échange du disque dur (le Swap ou Paging)²¹ quand la mémoire vive est saturée. Si cela se produit, la vitesse de génération s'effondre littéralement et l'application devient inutilisable. C'est pour cette raison que le logiciel est obligé d'analyser les capacités de la machine hôte en direct, au moment de l'exécution.

Analyse Hardware et filtrage des modèles

Au démarrage, le coeur IA commence par analyser la configuration matérielle de la machine. Pour ça, il utilise la bibliothèque `psutil` [31, 32] afin de récupérer la quantité totale de mémoire vive. En parallèle, il vérifie le nombre de cœurs logiques disponibles en faisant appel à `multiprocessing.cpu_count()`.

Cette analyse matérielle répond à deux vrais besoins. D'abord, il permet de répartir la charge sur le processeur de façon sécurisée. Pour que le modèle puisse tourner sans figer l'ordinateur, le code applique une règle de calcul assez simple : $\max(1, \text{cpu_count} - 2)$. L'idée est de ne jamais laisser le modèle monopoliser la totalité du CPU. On réserve systématiquement deux cœurs. Le premier sert à garder le système d'exploitation fluide, et le second s'occupe de la boucle d'événements asynchrone de NiceGUI. C'est une sécurité indispensable. Sans ça, lancer le programme sur un PC portable un peu limite ferait très certainement planter la machine.

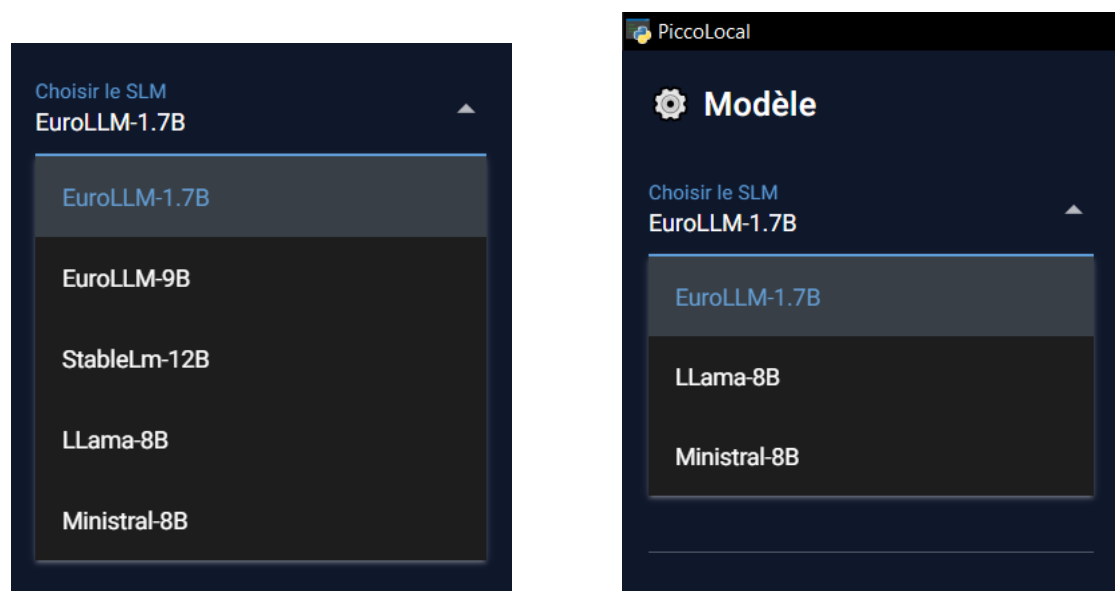
19. Fonctions capables de se "mettre en pause" pendant qu'elles attendent un résultat (comme un téléchargement ou un calcul), permettant au processeur d'exécuter d'autres tâches en attendant sans figer l'application.

20. Permet un échange de données bidirectionnel et en temps réel entre un client (ici l'UI) et un serveur (ici le coeur IA). Contrairement au protocole HTTP classique, la connexion reste ouverte en permanence, permettant l'envoi instantané d'informations sans que le client ait besoin de formuler une nouvelle requête.

21. Espace sur le disque dur utilisé par le système d'exploitation comme "mémoire vive de secours" lorsque la vraie RAM physique est pleine. Son accès est extrêmement lent comparé à la RAM.

Le second intérêt de cette analyse, c'est de mettre en place un filtrage matériel proactif des modèles proposés (*Hardware-Aware Filtering*). Au lieu de laisser l'utilisateur lancer le téléchargement d'un modèle beaucoup trop lourd (comme ceux de 8 ou 12 milliards de paramètres, qui dépassent souvent les 6 Go) pour finalement planter au chargement par manque de mémoire, le système vérifie la viabilité en amont.

L'algorithme compare tout simplement la RAM totale de l'ordinateur avec le poids du modèle quantifié, en n'oubliant pas d'ajouter la fameuse marge de sécurité pour le système d'exploitation. Si la machine n'a pas la capacité nécessaire, le modèle est purement et simplement retiré des options de l'interface. L'utilisateur ne voit donc que ce qui peut réellement tourner sur son PC, ce qui évite la frustration et les mauvaises surprises.



(a) Choix disponible sur un ordinateur possédant 16 GB de mémoire RAM

(b) Choix disponible sur un ordinateur possédant 8 GB de mémoire RAM

FIGURE 3 – Choix de modèles disponibles sur différentes configurations

Dimensionnement du *KV Cache*

L'un des défis les plus techniques de l'implémentation a été la gestion de la fenêtre de contexte (*Context Window*)²². Lors de l'inférence, le système s'appuie sur la structure de

22. La quantité maximale de texte (mesurée en tokens) que le modèle peut analyser et "garder en mémoire" en une seule fois pour générer sa réponse.

clés et de valeurs (*Keys* et *Values*) introduite par l'architecture Transformer [41]. Pour éviter de recalculer inutilement ces matrices pour l'intégralité des mots passés à chaque nouveau token généré, l'historique de la conversation est sauvegardé dynamiquement dans une mémoire tampon appelée le *Key-Value Cache* (KV Cache). Plus la conversation s'allonge, plus ce cache consomme de la mémoire vive, de manière quadratique²³. Si l'on assigne une fenêtre de contexte trop grande (par exemple 32 000 tokens) sur une machine avec peu de RAM, le chargement du modèle provoquera une erreur fatale de type *Out Of Memory*, faisant crasher l'application.

Pour régler ça, on a intégré au code une fonction de dimensionnement dynamique (`_calculate_safe_context`). La fonction part d'un calcul simple : RAM totale moins le poids du modèle GGUF, moins 3 Go de marge pour l'OS (basé sur les informations données par Microsoft [21, 22]). Ce qui reste donne le budget mémoire disponible pour le KV Cache.

En fonction de la mémoire qu'il reste (*available for context*), le système détermine le nombre maximum de tokens qu'il peut allouer. L'allocation s'adapte automatiquement à la machine : ça va de 1024 tokens pour les PC les plus limités, jusqu'à 32768 tokens pour les PC plus performants.

Ce petit mécanisme fait toute la différence. Il garantit un taux d'échec de chargement presque nul, ce qui rend l'application robuste et capable de tourner de manière fiable, peu importe la configuration matérielle de l'ordinateur de l'étudiant.

Gestion de la mémoire par fenêtre glissante dynamique

Ajuster dynamiquement le *KV Cache* permet d'éviter le fameux crash (*Out Of Memory*) au moment de charger le modèle, mais ça ne fait pas tout. Un autre problème arrive sur la durée, quand la conversation s'allonge. Par exemple, si un étudiant passe plusieurs heures à échanger avec le modèle, l'historique va logiquement continuer de grossir. À un moment donné, la taille cumulée des messages finira forcément par saturer la limite matérielle qu'on lui a fixée (par exemple, 2048 tokens).

Pour éviter que la mémoire ne finisse par saturer, l'application utilise un système de

23. Cela fait référence à la complexité spatiale de l'algorithme ($O(N^2)$). Dans un modèle d'IA classique, chaque nouveau mot généré doit calculer sa relation avec absolument tous les mots stockés dans l'historique. En résumé : si la conversation devient deux fois plus longue, les ressources mémoire nécessaires pour analyser cet historique sont multipliées par quatre.

fenêtre glissante (*Sliding Window*). Le principe est simple :

1. **Tokenisation préventive** : Le contenu de chaque message de l'historique est converti en *tokens* via le tokenizer du modèle chargé.
2. **Priorisation du contexte** : Le *prompt système* est systématiquement conservé pour que le modèle garde toujours ses instructions de base.
3. **Suppression des vieux messages** : Le code parcourt l'historique en partant des messages les plus récents et les accumule. Pour éviter de remplir totalement la mémoire, le poids cumulé ne doit jamais dépasser la limite de `_calculate_safe_context` à laquelle on soustrait une marge de génération (`max_new_tokens`). Cette marge garantit qu'il restera toujours de la place physique pour que le modèle puisse formuler sa nouvelle réponse.

Les messages plus anciens sont donc temporairement "oubliés" par le contexte actif du modèle. Bien sûr, ils ne sont pas perdus pour autant puisqu'ils restent sauvegardés dans le fichier JSON pour que l'utilisateur puisse relire toute sa conversation. C'est une méthode assez standard quand on développe des chatbots. Elle permet non seulement de garder un temps de réponse stable, mais surtout d'éviter que l'application ne plante à cause d'un contexte trop lourd pour la machine.

Gestion explicite de la mémoire (*Garbage Collector*)

Normalement, en Python, c'est le garbage collector [30] qui gère la libération de la mémoire de façon totalement transparente. Mais dans notre cas, les modèles LLM pèsent plusieurs gigaoctets. Comme ils sont chargés via *llama.cpp*[7], ils occupent des espaces mémoire directement alloués en C, ce qui échappe en grande partie au contrôle de Python.

À titre d'exemple, si l'utilisateur veut changer de modèle en cours de route, il ne suffit pas de réaffecter la variable Python. Ça ne vide pas la RAM assez vite et on risque de saturer la mémoire au moment d'essayer de charger le second modèle.

Pour éviter ça, Il a été nécessaire de développer une fonction de déchargement explicite (`unload_model`). Elle force d'abord la fermeture du processus côté C++ (`model.close()`), détruit la référence en Python (`del`), puis déclenche manuellement et immédiatement un passage du garbage collector (`gc.collect()`). C'est le genre de

gestion manuelle de mémoire qu'on ne s'attend pas à faire en Python, mais les bindings C de llama.cpp ne laissent pas le choix.

5.4 Défis d'intégration : Gestion de la concurrence et du Streaming

Au niveau de l'architecture, une autre difficulté a été de faire cohabiter l'interface web avec le moteur d'inférence C++. Le problème, c'est qu'ils fonctionnent de manière diamétralement opposée : l'interface est asynchrone et réagit aux événements, tandis que le code C++ est par nature synchrone et bloquant. Si on ne gère pas la communication entre ces deux mondes, l'application finit inévitablement par geler complètement pendant que le modèle calcule.

La concurrence pour la génération de texte

Quand l'utilisateur envoie une requête, le modèle quantifié ne répond pas instantanément. Il met parfois plusieurs secondes avant de réussir à produire le tout premier mot (*Time To First Token*)²⁴, puis génère la suite mot à mot.

Le piège, ce serait que la fonction de l'interface graphique appelle directement le moteur de l'IA. Si on faisait ça de façon classique, le cœur IA se mettrait en pause le temps du calcul. Il deviendrait totalement incapable de réagir à la moindre interaction (comme un simple clic sur un autre menu) tant que la génération du texte n'est pas complètement terminée.

Pour éviter ce blocage, une gestion de la concurrence a été mise en place. L'idée étant de combiner le *Multithreading* natif de Python avec l'asynchrone d'*Asyncio* déjà présenté précédemment.

Trois grandes étapes ont donc été mises en place :

1. **Thread Worker** : La prédiction des *tokens* par llama.cpp est déportée dans un *Thread* en arrière-plan. Ce *Worker* encaisse toute la charge lourde du processeur, ce qui laisse le fil d'exécution principal totalement libre.
2. **Queue** : Il fallait ensuite un moyen de récupérer les mots générés depuis ce *Thread* isolé. Pour ça, l'application s'appuie sur une queue. Queue, une structure qui a

24. Temps de latence mesuré entre le moment où l'utilisateur valide sa requête et l'apparition du tout premier mot généré par le modèle à l'écran.

l'avantage d'être *Thread-Safe*²⁵. À chaque fois que le modèle produit un nouveau mot, le *Worker* l'envoie dans cette file d'attente grâce à l'instruction `yield`.

3. **Asyncio** : Du côté de l'interface, une boucle asynchrone `while True` surveille cette file d'attente en permanence. Dès qu'un mot arrive, elle le récupère et l'affiche à l'écran en temps réel (*Streaming*). Si la file est vide, elle rend la main au processeur pendant un très court instant (avec `await asyncio.sleep(0.05)`). C'est ce petit délai qui laisse le temps à NiceGUI de traiter les autres actions de l'utilisateur, comme faire défiler la page ou afficher proprement une animation de chargement.

C'est toute cette organisation du code qui permet de garder l'application fluide. L'utilisateur a l'impression d'avoir une interface réactive et multitâche, alors qu'en réalité, les calculs qui tournent en dessous sont lourds.

Flux asynchrones pour le téléchargement de poids

La même logique asynchrone a été gardée pour gérer les téléchargements depuis le *Hugging Face Hub*. Étant donné qu'un modèle GGUF pèse facilement entre 1,5 et 8 Go, lancer un téléchargement de manière classique aurait complètement figé l'interface pendant de longues minutes. L'utilisateur se serait retrouvé coincé devant un écran qui ne répond plus du tout.

Pour éviter ça, l'application fait ce qu'on appelle du streaming de données réseau. Au lieu d'essayer de charger un fichier géant d'un seul bloc en mémoire, la requête HTTP est configurée pour le récupérer petit à petit (`stream=True`). Le fichier est téléchargé et écrit sur le disque dur par petits morceaux (*chunks*) de 1 Mo, en utilisant un générateur.

Bien sûr, tout ce processus tourne de son côté, dans son propre *Thread*. Pour communiquer avec l'interface, ce *Thread* envoie sa progression dans une deuxième *Queue*. L'interface lit ensuite ces informations en asynchrone pour mettre à jour une barre de progression en direct. C'est indispensable pour l'expérience utilisateur (*UX*), ça permet de voir clairement que le téléchargement avance et d'être rassuré sur le fait que le programme n'a pas planté.

25. Structure de données ou un bout de code qui peut être manipulé simultanément par plusieurs processus (threads) sans provoquer de corruption de données ou de plantage.

6 Vers une intégration dans Piccolo

Jusqu'à présent, l'application développée se concentre exclusivement sur une exécution locale ce qui permet garantir une confidentialité totale et un fonctionnement hors ligne. Cependant, une évolution logique du projet consisterait à créer une passerelle entre ce 'cœur IA' local et l'infrastructure cloud institutionnelle de l'UCLouvain, baptisée Piccolo. L'idée, c'est donc de ne pas opposer local et cloud, mais de les faire coopérer intelligemment.

6.1 Piccolo actuellement

Il est important de préciser que Piccolo n'est pas un modèle de langage en soi, mais plutôt une *interface web institutionnelle* [38] couplée à un *backend sécurisé* développé en interne par l'UCLouvain [39]. Cette plateforme sert de passerelle entre la communauté universitaire (authentifiée via les identifiants classiques de l'université) et des gros modèles hébergés sur des serveurs sécurisés (permettant l'accès à des modèles de pointe comme ceux de Mistral AI).

Le rôle de ce 'cœur IA' institutionnel est de donner accès à l'intelligence artificielle tout en gardant le contrôle sur les données (ce que l'UCLouvain appelle la souveraineté numérique). L'infrastructure s'assure que les flux de données sont chiffrés et anonymisés vis-à-vis des fournisseurs externes. Les requêtes ne sont jamais utilisées pour ré-entraîner les LLMs commerciaux. L'objectif de cette intégration théorique serait donc d'offrir une architecture hybride idéale : utiliser la gratuité, la rapidité et la confidentialité du moteur local pour les tâches quotidiennes, et la puissance de calcul du cloud de l'université uniquement lorsque les limites matérielles du PC de l'étudiant sont atteintes.

6.2 Défis d'intégration et conception de l'API

Faire communiquer l'application locale avec Piccolo suppose de créer une API côté serveur, séparée du backend interne de l'université pour ne pas exposer les données existantes.

Pour illustrer cela, tout reposerait sur une clé API. L'étudiant générerait cette clé depuis l'interface web de Piccolo et la renseignerait dans l'application locale. Cette clé servirait d'identifiant unique à chaque requête, permettant au serveur de décompter

les crédits²⁶ alloués à l'utilisateur. Du côté des bases de données de l'université, l'API nécessiterait la création d'une table dédiée exclusivement à la "consommation". Afin de respecter la philosophie du projet local, le contenu textuel des requêtes et des réponses ne serait jamais enregistré côté serveur. La mémoire des conversations resterait sur le disque dur de l'utilisateur.

L'API serait donc stateless et c'est l'application locale qui renverrait tout le contexte de la conversation à chaque requête. Du point de vue sécurité, c'est un avantage, si le serveur est compromis, il n'y a rien à voler puisque les conversations n'y transitent que le temps d'une requête.

6.3 Choix technologiques pour la communication réseau

Pour implémenter cette communication HTTP, il est crucial de sélectionner des bibliothèques adaptées à l'architecture existante de l'application (basée sur NiceGUI et asyncio). Le vrai goulot d'étranglement n'est pas le réseau mais llama.cpp. Comme la bibliothèque C++ est fondamentalement bloquante, l'architecture thread + queue était de toute façon obligatoire pour l'inférence locale. S'il était exécuté sur le fil principal, le code bloquerait l'interface graphique le temps de recevoir la réponse du cloud. La solution naturelle consiste donc à réutiliser exactement le même *Design Pattern*, c'est-à-dire déporter la requête HTTP synchrone dans un `threading.Thread` en arrière-plan et transmettre la réponse (streamée mot par mot) à l'interface via une queue. C'est pour cette raison que le choix se porterait sur Requests plutôt que sur HTTPX. Ce dernier pourrait tout à fait être utilisé pour la partie réseau, mais cela introduirait deux patterns de concurrence différents dans le même projet sans rien simplifier pour autant.

6.4 Critères de délégation : Quand basculer sur l'API?

La pertinence d'une architecture hybride repose sur sa capacité à n'utiliser le cloud que lorsque c'est nécessaire. L'appel à l'API Piccolo ferait office "d'accélérateur" dans certains cas d'usage :

1. **Les limites matérielles (RAM/VRAM) :** L'application gère la saturation du KV Cache via une fenêtre glissante décrite dans le chapitre précédent. Ce mécanisme évite le plantage, réduit une partie de la conversation. Déléguer vers Piccolo dans

26. Piccolo alloue un quota de crédits à chaque utilisateur. Au-delà de la régulation des coûts, ce mécanisme vise à sensibiliser à l'impact environnemental réel de chaque requête envoyée au cloud. [40].

ce cas permet de conserver l'intégralité de l'historique, même si, comme on le verra en section 6.5, une compression du contexte reste conseillée pour ne pas utiliser inutilement des crédits.

2. **Le besoin de haut raisonnement** : Le SLM local s'en sort bien pour de la synthèse ou de la classification simple, mais dès qu'on passe à des maths avancées ou de l'algorithmique, il décroche comme le montrent les benchmarks présentés en section 3.3. Dans ces cas-là, déléguer à l'API permet d'accéder à des modèles bien plus puissants.
3. **L'économie d'énergie et gestion thermique** : L'inférence locale pousse le CPU à 100%. Sur un portable sur batterie, ça chauffe, le ventilateur s'emballe, et l'autonomie fond. C'est une conséquence directe du Memory Wall décrit en section 4.3 car générer des tokens exige de lire en permanence des gigaoctets de paramètres depuis la RAM.
4. **La vitesse de génération (Throughput)** : Pour les requêtes demandant une sortie textuelle très longue un processeur grand public reste limité par sa bande passante mémoire comme décrit en section 4.3. Il est donc raisonnable d'anticiper que des serveurs dédiés offriront un meilleur temps de génération, même si cette différence n'a pas été mesurée dans le cadre de ce projet.

6.5 Compression du contexte avant délégation

L'un des défis souvent négligés dans une architecture hybride est le coût de la délégation elle-même. Concrètement, envoyer à Piccolo l'intégralité d'une longue conversation consomme des crédits de manière disproportionnée par rapport à la tâche réellement demandée.

Pour pallier ce problème, une stratégie élégante consisterait à utiliser le SLM local comme un outil de compression de prompt (*prompt compression*). Avant chaque envoi vers le cloud le modèle local analyserait la *self-information* (ou perplexité) de chaque élément de l'historique pour éliminer sélectivement les tokens redondants ou hautement prévisibles. Un travail fondateur LLMingua [15] montre qu'il est possible d'élaguer la taille d'un contexte d'un facteur de 4x à 20x avec une perte quasi négligeable de la qualité des réponses du modèle distant. Ce prompt condensé et filtré remplacerait l'historique brut dans la requête envoyée à Piccolo, réduisant ainsi le nombre de tokens.

De plus, cette compression a un effet secondaire intéressant pour la vie privée. Au lieu d'envoyer le texte brut de toute la conversation à Piccolo, on n'envoie qu'un prompt condensé avec potentiellement des informations concernant l'utilisateur en moins. Cependant, le problème c'est que le prompt condensé n'est pas toujours fiable. Sur un échange très technique, un SLM local risque de perdre des détails importants en condensant, et le modèle cloud se retrouverait à travailler avec un contexte trop vague.

6.6 Graceful degradation

Travailler dans le train ou dans une bibliothèque mal couverte est une réalité quotidienne pour beaucoup d'étudiants et c'est exactement le genre de situation où une requête vers Piccolo risque de ne pas passer. Ce problème de connectivité est classique dans les architectures de *split computing*, où le calcul est réparti entre une machine locale et un serveur distant. Les déconnexions temporaires et les variations de latence sont parmi les défis les plus documentés de ce type d'architecture [20].

L'architecture envisagée intégrerait donc un mécanisme de *graceful degradation*²⁷. Cela implique qu'avant chaque envoi, le logiciel effectuerait un simple *ping* vers l'endpoint de Piccolo avec un timeout court. Si la réponse n'arrive pas, l'application bascule sur le moteur local et notifie l'utilisateur.

C'est d'ailleurs pour ça que la logique de routage doit rester côté client. Si elle était côté serveur, une panne réseau bloquerait tout. Avec le client qui décide, le moteur local prend le relais automatiquement et la conversation continue sans interruption.

6.7 Sensibilisation aux crédits : vers une UX transparente

L'une des particularités de Piccolo est son système de crédits [39]. Chaque requête envoyée au cloud consomme un certain nombre de tokens, qui se traduit en une déduction sur le quota de l'étudiant. Le souci, c'est que dans la plupart des interfaces existantes, cette consommation est totalement invisible.

L'architecture hybride proposée ici est l'occasion de concevoir une UX plus honnête. Avant de déléguer une requête vers Piccolo, le logiciel pourrait estimer le coût approximatif en crédits et afficher un message de confirmation si ce coût dépasse un seuil configurable. D'abord, ça responsabilise l'étudiant, il prend conscience du coût réel et peut décider

27. Principe selon lequel un système continue de fonctionner de manière réduite mais cohérente lorsqu'une partie de ses ressources devient indisponible.

en connaissance de cause. Ensuite, ça l’incite à n’utiliser le cloud que quand ça en vaut vraiment la peine, ce qui rejoint directement l’objectif de responsabilité numérique de Piccolo [40].

6.8 Mécanismes de routage

Les quatre critères définis en section 6.4 (limites matérielles, besoin de haut raisonnement, gestion thermique, throughput) posent le cadre conceptuel. Mais dans la pratique, comment le système les détecte-t-il automatiquement, sans intervention manuelle de l’étudiant à chaque requête ? C’est là qu’intervient le moteur de routage.

Ce problème est bien identifié dans la littérature et des solutions concrètes ont déjà été validées expérimentalement. HybridLLM [4] est la référence la plus directe. Ce framework propose exactement l’architecture envisagée ici c’est-à-dire un petit modèle sur l’appareil de l’utilisateur combiné à un grand modèle cloud, avec un routeur qui décide dynamiquement lequel solliciter selon la difficulté estimée de la requête. Les auteurs démontrent qu’il est possible de réduire de 22 à 40 % les appels au cloud avec moins de 1 % de perte de qualité. L’architecture théorisée pour Piccolo s’inspire de cette logique. La différence, c’est le contexte. Ici, pas des API commerciales, mais un cloud institutionnel avec un système de crédits.

La question centrale est donc : comment estimer la difficulté d’une requête avant même d’y répondre ?

HybridLLM [4] propose un routeur basé sur un encodeur compact de type DeBERTa²⁸, entraîné à prédire l’écart de qualité attendu entre le petit modèle local et le grand modèle cloud pour une requête donnée. Si cet écart est faible la requête reste en local. Dans le cas contraire, elle est déléguée.

1. Le routage par seuil de qualité

C’est l’approche centrale de HybridLLM [4]. Le routeur prédit un score de difficulté pour chaque requête et le compare à un seuil configurable. Ce seuil est ajustable dynamiquement. En l’abaissant, on délègue moins au cloud et on économise des crédits

28. DeBERTa (Decoding-Enhanced BERT with Disentangled Attention) est un modèle de compréhension du langage développé par Microsoft [9]. Il ne produit pas de texte comme les LLMs génératifs, il calcule uniquement un score de classification en un seul passage, ce qui le rend quasi instantané sur un processeur classique.

tandis qu'en le relevant, on privilégie la qualité au détriment du coût. Dans le contexte de Piccolo, ce seuil pourrait prendre la forme d'un simple curseur dans l'interface. L'utilisateur choisit lui-même jusqu'où il veut solliciter le cloud.

2. Le routage manuel comme filet de sécurité

Aussi performant soit-il, un routeur automatique peut se tromper. Bien qu'HybridLLM [4] évite d'enchaîner automatiquement plusieurs modèles pour ne pas surcharger la puissance de calcul, l'approche la plus pragmatique pour Piccolo consiste à impliquer l'utilisateur dans la décision. Dans ce scénario, si le modèle local répond en premier mais que le résultat produit est jugé incomplet ou incohérent par l'étudiant, l'interface proposerait un bouton de « Régénération via le cloud ». Ce mécanisme simple de fallback manuel permet de ne solliciter les ressources distantes qu'en cas d'échec avéré et validé par l'humain, offrant un complément indispensable au routage automatique pour les cas limites où le score du routeur oscille près du seuil.

3. Le routage déterministe comme pré-filtre léger

Le routeur automatique n'a pas besoin d'intervenir pour tout. Certaines requêtes sont trivialement complexes. Par exemple, demander un algorithme de tri, un calcul différentiel, ou une sortie au format JSON. Un simple scan de mots-clés peut les détecter rapidement. Ces règles déterministes agissent comme un filtre rapide. Elles traitent les cas clairs immédiatement et laissent le routeur automatique s'occuper du reste.

6.9 Le futur de Piccolo

Bien que ce mémoire se concentre sur une application de bureau indépendante, ce travail sur le routage hybride s'inscrit dans une réflexion plus large sur l'informatique à l'UCLouvain. Actuellement pensé pour un usage ciblé, Piccolo a vocation à devenir modulaire.

À moyen terme, l'infrastructure pourrait intégrer un proxy open source tel que LiteLLM. Ce type de solution permet de standardiser toutes les API d'intelligence artificielle sous un format unique (le format OpenAI). Pour le logiciel local développé ici, cela signifierait qu'une fois le connecteur HTTP développé, il deviendrait compatible

avec n'importe quel autre modèle que l'université déciderait de brancher en arrière-plan, sans avoir à modifier une seule ligne de code côté client.

7 Evaluation

Après avoir détaillé l'architecture logicielle et posé les bases de l'intégration avec Piccolo, il est temps de prendre du recul. Ce chapitre propose un retour critique sur le travail réalisé : est-ce que l'application atteint les objectifs fixés au départ ? Où se situe-t-elle par rapport aux outils existants ? Et surtout, quelles sont ses limites ?

7.1 Retour sur les objectifs initiaux

Dès le départ, le projet a été pensé autour de plusieurs enjeux majeurs : la confidentialité des données, l'accessibilité, la réduction de l'empreinte écologique et la souveraineté numérique. Reste à voir comment ces ambitions se sont traduites concrètement dans l'implémentation.

La **confidentialité** était l'enjeu principal, et le contrat est rempli. Les poids des modèles sont téléchargés une seule fois depuis *HuggingFace*, puis toute l'inférence tourne directement sur la machine de l'étudiant. Les historiques, eux, sont sauvegardés au format JSON dans le dossier applicatif classique du système d'exploitation. Rien ne quitte le disque local. Le choix du JSON plutôt qu'une base de données distante renforce ce principe car les fichiers restent lisibles, facilement auditable et sous le contrôle total de l'utilisateur.

L'**accessibilité** montre parfaitement l'utilité des mécanismes présentés au chapitre 5. En combinant le filtrage matériel, le dimensionnement du *KV Cache* et le système de fenêtre glissante, l'application s'adapte d'elle-même à la configuration de la machine hôte. Un étudiant avec seulement 8 Go de RAM ne se voit proposer que les modèles capables de tourner sur son ordinateur, ce qui évite les plantages et pas mal de frustration.

Comme le soulignait l'analyse de l'écosystème (chapitre 3), cette accessibilité est une avancée récente, rendue possible par l'évolution des SLMs et l'arrivée à maturité des techniques de quantification.

L'**empreinte écologique**, reste une préoccupation du projet. L'application tournant exclusivement sur le matériel que l'étudiant possède déjà, on évite de solliciter des *data centers* distants à la moindre requête. Aucun entraînement n'est fait ici. Tout se limite à

de l'inférence en local, sur des modèles de petite taille. C'est une approche qui s'aligne avec la vision de responsabilité numérique portée par *Piccolo* [40], en allant même un peu plus loin, puisque l'utilisation quotidienne ne consomme plus aucune ressource côté serveur.

La **souveraineté numérique**, quant à elle, est respectée. Le code étant open source et publié sous licence MIT, l'application n'enferme l'utilisateur dans aucun écosystème propriétaire. N'importe qui peut modifier le logiciel, ajouter de nouveaux modèles à la liste de ceux auxquels il a accès et choisir ceux qu'il préfère.

7.2 Positionnement par rapport à l'existant

Le chapitre 2 a déjà présenté les principales solutions d'IA locale disponibles aujourd'hui. Ces outils sont matures, largement utilisés et il aurait été tout à fait défendable de partir de l'un d'entre eux plutôt que de réinventer la roue. La question légitime est donc de savoir ce que cette application fait que les autres ne font pas ?

Une accessibilité matérielle pensée pour l'étudiant

Le filtrage matériel et le dimensionnement du *KV Cache*, détaillés au chapitre 5, sont absents de la plupart des solutions existantes. *Ollama* laisse l'utilisateur télécharger un modèle de 8B sur une machine de 8 Go de RAM, et le constate au moment du crash. *LM Studio* fait un peu mieux avec un système de tags indicatifs, mais ne va pas jusqu'à filtrer la liste des modèles proposés en fonction du matériel détecté.

On passe d'un produit destiné à un public déjà formé (qui sait ce qu'est un modèle 4B ou 8B) à un produit utilisable par un étudiant qui n'a aucune connaissance du sujet.

Une transparence que les solutions propriétaires ne permettent pas

LM Studio est sans doute la solution la plus proche en termes de public cible car il a une interface graphique soignée pour faire tourner des modèles locaux. Mais c'est un logiciel propriétaire. Impossible d'auditer le code, impossible de vérifier ce qui est fait des prompts, impossible d'adapter le comportement à un contexte particulier.

Pour un projet destiné à s'intégrer dans un environnement universitaire, ça rend la chose impossible. L'UCLouvain ne peut pas recommander à ses étudiants un outil

dont le fonctionnement interne est opaque, surtout dans un contexte où les questions de souveraineté numérique et de protection des données sont centrales.

Une intégration institutionnelle inédite

C'est probablement le point le plus original du projet. Aucun d'entre eux n'a été pensé pour coopérer avec un cloud institutionnel. *Ollama* et *LM Studio* se positionnent comme des alternatives au cloud commercial, pas comme des partenaires d'un cloud universitaire.

Synthèse

Aucun des outils existants ne couvre simultanément les différents axes cités ci-dessus. C'est cette combinaison particulière qui justifie le développement d'un outil dédié plutôt que la réutilisation d'un logiciel existant.

7.3 Limites et perspectives d'amélioration

Il serait malhonnête de présenter l'application sans en assumer les limites. Plusieurs choix de conception, bien que justifiés dans leur contexte, méritent d'être discutés.

Absence de support GPU explicite

La bibliothèque `llama.cpp` supporte nativement l'accélération matérielle via CUDA [24], Metal [1] ou Vulkan [16]. Or, l'application actuelle repose uniquement sur une inférence CPU. Ce choix a été motivé par le fait que tous les étudiants n'ont pas un PC équipé d'une carte graphique dédiée. Le souci, c'est que pour les étudiants qui en ont, cela représente une perte de performance qu'une détection automatique du *backend* au démarrage viendrait combler.

Dépendance forte à llama.cpp

Toute l'architecture repose sur *llama-cpp-python*. Si cette bibliothèque venait à être abandonnée ou à évoluer dans une direction incompatible, une refonte importante serait nécessaire. Ce couplage était toutefois difficilement évitable car `llama.cpp` reste aujourd'hui la solution de référence pour l'inférence CPU de modèles GGUF, et les

alternatives (comme *mlx* côté Apple ou *candle* côté Rust) restent soit beaucoup moins matures, soit plus spécialisées.

Absence de mécanismes RAG

L'application ne propose aucune fonctionnalité de *Retrieval-Augmented Generation*. Concrètement, un étudiant qui souhaite interroger ses propres documents de cours doit copier-coller manuellement le contenu dans le contexte, ce qui atteint vite les limites de la fenêtre de contexte du modèle. C'est probablement le manque le plus visible pour un outil à vocation académique.

Pas de support multimodal

Les modèles actuellement supportés sont exclusivement textuels. Or, les SLMs multimodaux comme *Llava* ou *MiniCPM-V* commencent à devenir accessibles en version quantifiée au format GGUF. Pouvoir soumettre un schéma ou une photo de tableau apporterait une vraie valeur ajoutée, mais cela dépassait le cadre initial du projet.

Évaluation utilisateur absente

Aucune étude utilisateur n'a été menée dans le cadre de ce travail. Les choix d'interface ont été guidés à partir d'heuristiques classiques d'UX et de d'intuitions, sans validation auprès d'un panel d'étudiants. Certaines décisions de conception mériteraient d'être confrontées à des retours réels.

8 Conclusion

Au terme de ce travail, il est temps de prendre un peu de recul sur le chemin parcouru. Le projet est parti d'un constat assez simple. L'IA générative est devenue un outil quotidien pour les étudiants, mais les solutions dominantes posent de sérieuses questions de confidentialité, d'accessibilité, d'impact écologique et de souveraineté. À partir de là, deux objectifs ont été fixés. D'abord, vérifier que l'on peut techniquement construire un assistant IA local utilisable, robuste et accessible à un public étudiant. Ensuite, poser les bases d'une architecture hybride dans laquelle ce cœur local pourrait coopérer avec une infrastructure cloud institutionnelle comme Piccolo.

Sur le premier objectif, la réponse est plutôt positive, même s'il faut la nuancer. L'analyse de l'écosystème open source a montré que les SLMs de la tranche 1B-4B atteignent aujourd'hui des performances qui auraient nécessité des modèles bien plus massifs il y a quelques années à peine. La quantification, et plus particulièrement le format GGUF, permet ensuite de faire tenir ces modèles dans la mémoire d'un PC portable étudiant sans sacrifier l'essentiel de leurs capacités. Mis bout à bout, ces deux leviers rendent l'inférence locale clairement viable. Le prototype développé en est la preuve concrète, à condition d'investir dans la gestion dynamique des ressources. Cela passe par un filtrage *hardware-aware* des modèles proposés, un dimensionnement adaptatif du *KV Cache*, une fenêtre glissante sur l'historique et une gestion explicite de la mémoire allouée.

Sur le second objectif, le travail reste à un stade plus conceptuel. Le chapitre 6 propose une architecture concrète pour l'intégration avec Piccolo, qui couvre les critères de délégation, la compression du contexte avant envoi, les mécanismes de routage inspirés de HybridLLM, la *graceful degradation* et la sensibilisation aux crédits.

Au-delà de ces deux objectifs, plusieurs limites ont été identifiées et discutées au chapitre 7, notamment l'absence de support GPU explicite, l'absence de fonctionnalités RAG pour interroger ses propres documents, et l'absence de support multimodal. Ces manques ne sont pas anodins, mais ils annoncent les prochaines étapes en cas de développement ultérieur. À plus long terme, l'évolution la plus stratégique reste l'intégration concrète avec Piccolo.

Pour finir, ce travail défend une idée simple. Il est possible, en 2026, de proposer aux étudiants un assistant IA respectueux de leurs données, accessible sur leur matériel

existant, indépendant des géants du cloud, et qui reste néanmoins capable de coopérer avec les services institutionnels lorsque c'est pertinent. Les briques techniques existent, il ne reste qu'à les assembler.

Bibliographie

- [1] APPLE INC. *Metal — Apple Developer Documentation*. <https://developer.apple.com/metal/>. Avril 2026. (visité en mars et avril 2026).
- [2] CHAINLIT TEAM. *Overview - Chainlit Documentation*. <https://docs.chainlit.io/get-started/overview>. 2026. (visité en décembre 2025).
- [3] Tim DETTMERS et Luke ZETTLEMOYER. “The case for 4-bit precision : k-bit Inference Scaling Laws”. In : *Proceedings of the 40th International Conference on Machine Learning*. Sous la dir. d’Andreas KRAUSE et al. T. 202. Proceedings of Machine Learning Research. PMLR, 23–29 Jul 2023, p. 7750-7774. URL : <https://proceedings.mlr.press/v202/dettmers23a.html>.
- [4] Dujian DING et al. “Hybrid LLM : Cost-Efficient and Quality-Aware Query Routing”. In : *The Twelfth International Conference on Learning Representations*. 2024. URL : <https://openreview.net/forum?id=02f3mUtqnM>.
- [5] Georgi GERGANOV et CONTRIBUTORS. *ggml : Tensor library for machine learning*. <https://github.com/ggml-org/ggml>. 2023. (visité en janvier 2026).
- [6] Georgi GERGANOV et CONTRIBUTORS. *ggml : Tensor library for machine learning*. <https://github.com/ggml-org/ggml/blob/master/docs/gguf.md>. 2023. (visité en janvier 2026).
- [7] Georgi GERGANOV et CONTRIBUTORS. *llama.cpp : LLM inference in C/C++*. <https://github.com/ggml-org/llama.cpp>. Janvier 2026. (visité en janvier 2026).
- [8] GRADIO TEAM. *Build machine learning apps in Python*. <https://www.gradio.app/>. 2026. (visité en décembre 2025).

- [9] Pengcheng HE et al. “{DEBERTA} : {DECODING}-{ENHANCED} {BERT} {WITH} {DISENTANGLED} {ATTENTION}”. In : *International Conference on Learning Representations*. 2021. URL : <https://openreview.net/forum?id=XPZiaotutsD>.
- [10] HUGGING FACE. *GGUF - Hugging Face Hub documentation*. <https://huggingface.co/docs/hub/gguf>. 2024. (visité en janvier 2026).
- [11] HUGGING FACE. *Hugging Face — The AI community and model hub*. 2026. URL : <https://huggingface.co/>. (visité en octobre et novembre 2025).
- [12] HUGGING FACE. *Open LLM Leaderboard — Hugging Face Space*. lien vers le leaderborad direct (ne fonctionne pas parfois) : https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard#/. 2026. URL : <https://huggingface.co/open-llm-leaderboard>. (visité en octobre, novembre et décembre 2025).
- [13] Kushal JAIN et al. “First-Step Advantage : Importance of Starting Right in Multi-Step Math Reasoning”. In : *Findings of the Association for Computational Linguistics : ACL 2025*. Sous la dir. de Wanxiang CHE et al. Vienna, Austria : Association for Computational Linguistics, juill. 2025, p. 766-778. ISBN : 979-8-89176-256-5. DOI : 10.18653/v1/2025.findings-acl.42. URL : <https://aclanthology.org/2025.findings-acl.42/>.
- [14] JAN. *Jan : An open source alternative to ChatGPT that runs 100% offline on your computer*. <https://github.com/janhq/jan>. Site officiel : <https://jan.ai>. 2026. (visité en octobre 2025 et mars 2026).
- [15] Huiqiang JIANG et al. “LLMLingua : Compressing Prompts for Accelerated Inference of Large Language Models”. In : *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Sous la dir. d’Houda BOUAMOR, Juan PINO et Kalika BALI. Singapore : Association for Computational Linguistics, déc. 2023, p. 13358-13376. DOI : 10.18653/v1/2023.emnlp-main.825. URL : <https://aclanthology.org/2023.emnlp-main.825/>.
- [16] KHRONOS GROUP. *Vulkan — Cross platform 3D Graphics and Compute API*. <https://www.vulkan.org/>. Spécification : <https://registry.khronos.org/vulkan/>. 2026. (visité en mars et avril 2026).

- [17] Jiedong LANG, Zhehao GUO et Shuyu HUANG. “A Comprehensive Study on Quantization Techniques for Large Language Models”. In : *2024 4th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*. 2024, p. 224-231. DOI : 10.1109/ICAIRC64177.2024.10899941.
- [18] LM STUDIO. *LM Studio : Discover, download, and run local LLMs*. <https://lmstudio.ai/>. 2026. (visit  en septembre 2025 et mars 2026).
- [19] Zhenyan LU et al. *Small Language Models : Survey, Measurements, and Insights*. 2025. arXiv : 2409.15790 [cs.CL]. URL : <https://arxiv.org/abs/2409.15790>.
- [20] Yoshitomo MATSUBARA, Marco LEVORATO et Francesco RESTUCCIA. “Split Computing and Early Exiting for Deep Learning Applications : Survey and Research Challenges”. In : *ACM Comput. Surv.* 55.5 (d c. 2022). ISSN : 0360-0300. DOI : 10.1145/3527155. URL : <https://doi.org/10.1145/3527155>.
- [21] MICROSOFT. *Windows 10 system requirements*. <https://support.microsoft.com/en-us/windows/windows-10-system-requirements-6d4e9a79-66bf-7950-467c-795cf0386715>. 2025. (visit  en f vrier 2026).
- [22] MICROSOFT. *Windows 11 requirements*. <https://learn.microsoft.com/en-us/windows/whats-new/windows-11-requirements>. 2024. (visit  en f vrier 2026).
- [23] MINTPLEX LABS. *AnythingLLM : The all-in-one AI productivity accelerator*. <https://github.com/Mintplex-Labs/anything-llm>. Site officiel : <https://anythingllm.com>. 2026. (visit  en octobre 2025 et mars 2026).
- [24] NVIDIA CORPORATION. *CUDA Toolkit Documentation*. <https://docs.nvidia.com/cuda/>. Site officiel : <https://developer.nvidia.com/cuda-toolkit>. 2026. (visit  en mars et avril 2026).
- [25] OLLAMA. *Ollama : Get up and running with large language models locally*. <https://github.com/ollama/ollama>. Site officiel : <https://ollama.com>. 2026. (visit  en novembre 2025 et avril 2026).
- [26] OPEN WEBUI CONTRIBUTORS. *Open WebUI : User-friendly WebUI for LLMs*. <https://github.com/open-webui/open-webui>. 2026. (visit  en d cembre 2025).

- [27] Long OUYANG et al. *Training language models to follow instructions with human feedback*. 2022. URL : <https://proceedings.neurips.cc/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html>.
- [28] Nghiem Thanh PHAM et al. *SLM-Bench : A Comprehensive Benchmark of Small Language Models on Environmental Impacts*. Sous la dir. de Christos CHRISTODOULOPOULOS et al. Nov. 2025. DOI : 10.18653/v1/2025.findings-emnlp.1165. URL : <https://aclanthology.org/2025.findings-emnlp.1165/>.
- [29] PYTHON SOFTWARE FOUNDATION. *asyncio — Entrées/Sorties asynchrones*. <https://docs.python.org/fr/3/library/asyncio.html>. 2026. (visité en décembre 2025).
- [30] PYTHON SOFTWARE FOUNDATION. *gc — Interfaces pour le ramasse-miettes*. <https://docs.python.org/fr/3/library/gc.html>. 2026. (visité en février 2026).
- [31] Giampaolo RODOLÀ. *psutil documentation*. <https://psutil.readthedocs.io/>. 2026. (visité en janvier 2026).
- [32] Giampaolo RODOLÀ et CONTRIBUTORS. *psutil : Cross-platform lib for process and system monitoring in Python*. <https://github.com/giampaolo/psutil>. 2026. (visité en janvier 2026).
- [33] Alexander SHVETS. *Singleton Design Pattern*. <https://refactoring.guru/design-patterns/singleton>. 2026. (visité en janvier 2026).
- [34] LORENZ SPARRENBURG et al. “Small and Fast LLMs on Commodity Hardware : Post-Training Quantization in llama. cpp”. In : *2025 IEEE 12th International Conference on Data Science and Advanced Analytics (DSAA)*. 2025, p. 1-10. DOI : 10.1109/DSAA65442.2025.11247985.
- [35] STREAMLIT TEAM. *Main concepts - Streamlit Documentation*. <https://docs.streamlit.io/get-started/fundamentals/main-concepts>. 2026. (visité en décembre 2025).

- [36] Emma STRUBELL, Ananya GANESH et Andrew McCALLUM. “Energy and Policy Considerations for Deep Learning in NLP”. In : *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Sous la dir. d’Anna KORHONEN, David TRAUM et Lluís MÀRQUEZ. Florence, Italy : Association for Computational Linguistics, juill. 2019, p. 3645-3650. DOI : 10.18653/v1/P19-1355. URL : <https://aclanthology.org/P19-1355/>.
- [37] Houcemeddine TURKI et al. “Evaluating the Behavior of Small Language Models in Answering Binary Questions”. In : *3rd International Workshop on Generalizing from Limited Resources in the Open World (GLOW@IJCAI 2025)*. 2025. URL : https://www.researchgate.net/publication/392589019_Evaluating_the_Behavior_of_Small_Language_Models_in_Answering_Binary_Questions.
- [38] UCLouvain. *Piccolo — Interface de conversation*. 2025. URL : <https://piccolo.uclouvain.be/Chat>. (visité en novembre 2025).
- [39] UCLouvain. *Piccolo — La plateforme d’intelligence artificielle de l’UCLouvain*. 2025. URL : <https://www.uclouvain.be/fr/ia/piccolo>. (visité en mars et avril 2026).
- [40] UCLouvain. *Piccolo, l’outil IA de l’UCLouvain pour un usage responsable*. 2024. URL : <https://www.uclouvain.be/fr/news/piccolo-l-outil-ia-de-l-uclouvain-pour-un-usage-responsable>. (visité en avril 2026).
- [41] Ashish VASWANI et al. “Attention is All you Need”. In : *Advances in Neural Information Processing Systems*. Sous la dir. d’I. GUYON et al. T. 30. Curran Associates, Inc., 2017. URL : https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.
- [42] Fali WANG et al. “A Comprehensive Survey of Small Language Models in the Era of Large Language Models : Techniques, Enhancements, Applications, Collaboration with LLMs, and Trustworthiness”. In : *ACM Trans. Intell. Syst. Technol.* 16.6 (nov. 2025). ISSN : 2157-6904. DOI : 10.1145/3768165. URL : <https://doi.org/10.1145/3768165>.

- [43] ZAUBERZEUG. *NiceGUI : Create web-based user interfaces with Python*. <https://github.com/zauberzeug/nicegui>. 2026. (visité en décembre 2025 et février 2026).
- [44] Peiyuan ZHANG et al. *TinyLlama : An Open-Source Small Language Model*. 2024. arXiv : 2401.02385 [cs.CL]. URL : <https://arxiv.org/abs/2401.02385>.
- [45] Lianmin ZHENG et al. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In : *Advances in Neural Information Processing Systems*. Sous la dir. d’A. OH et al. T. 36. Curran Associates, Inc., 2023, p. 46595-46623. URL : https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

Annexe

Code source

L'intégralité du code source développé dans le cadre de ce mémoire a été rendue publique afin de garantir la reproductibilité du projet et d'encourager d'éventuels développements futurs.

L'application, qui comprend l'interface utilisateur (UI) et le moteur de génération en arrière-plan (*backend*), est distribuée sous la licence *open source* MIT.

Le dépôt de code contient les différents modules présentés dans le Chapitre 5 ainsi qu'un fichier README .

Le dépôt est accessible à l'adresse suivante :
github.com/LucasFueGo/SLMs-sur-ordinateur-personnel

Screenshot de l'implémentation

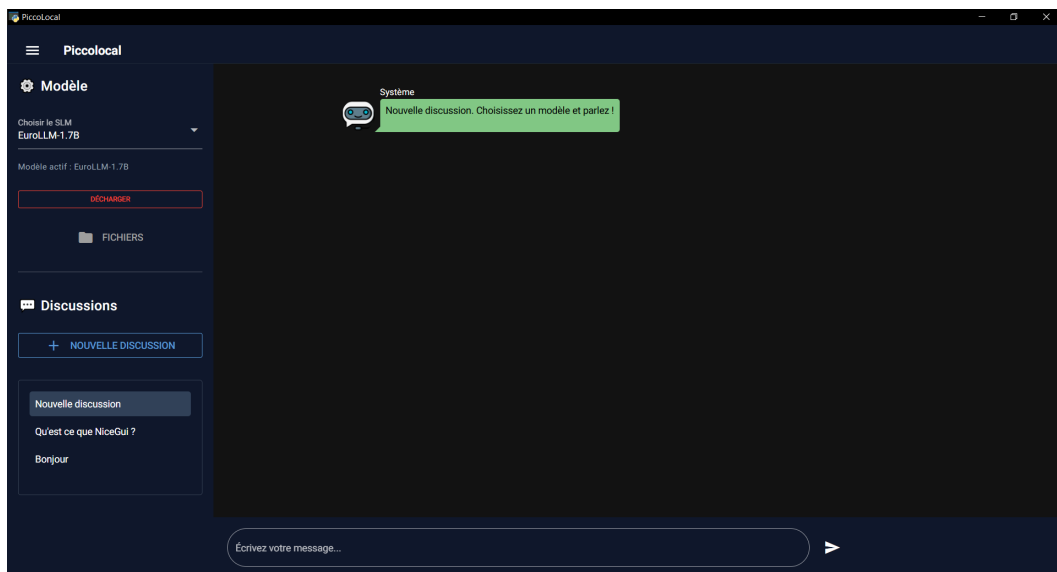


FIGURE 4 – Interface générale du prototype.

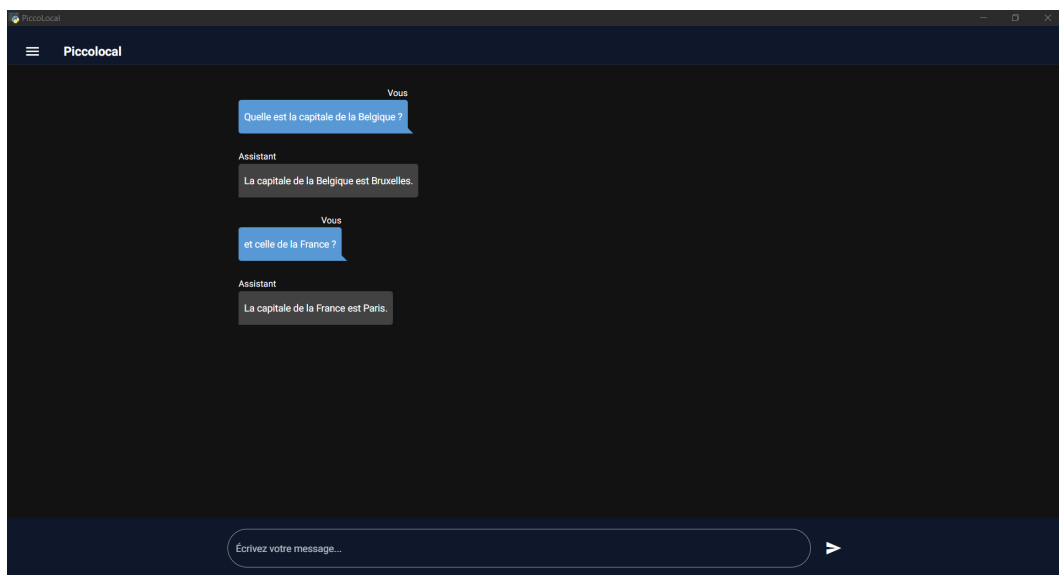


FIGURE 5 – interface de discussion avec le SLM.

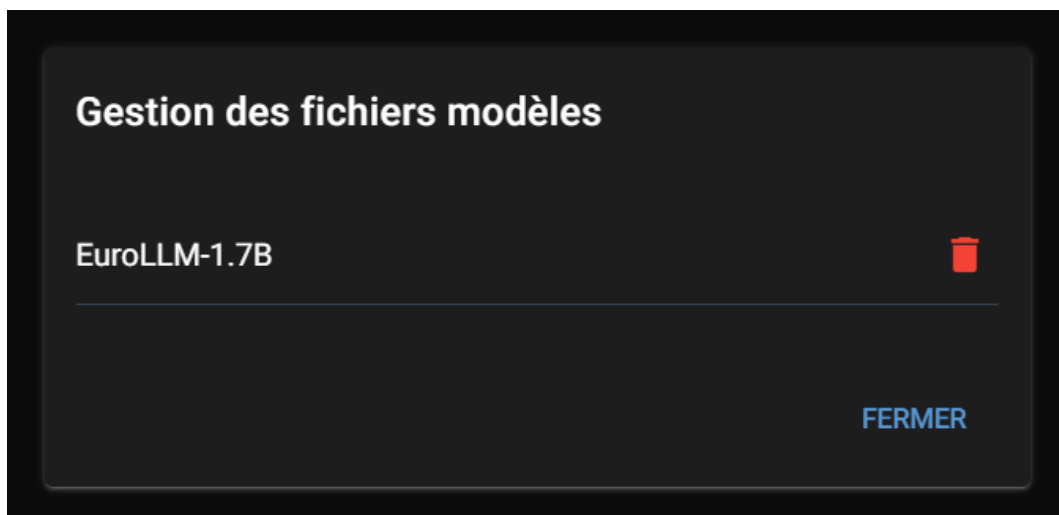


FIGURE 6 – Interface de gestion des modèles installés sur la machine de l'utilisateur.

Utilisation de l'intelligence artificielle

Dans le cadre de la rédaction de ce mémoire, j'ai utilisé des outils d'intelligence artificielle générative de manière accessoire, conformément aux lignes directrices de l'EPL relatives à l'usage responsable de ces technologies.

J'ai eu recours à ces outils à des fins d'exploration et de brainstorming : en amont de la rédaction, je les ai consultés pour identifier des pistes de réflexion, structurer des idées initiales ou orienter certaines recherches bibliographiques. Cette utilisation est comparable à celle d'un moteur de recherche et n'a en aucun cas remplacé la démarche documentaire que j'ai menée de façon rigoureuse à partir des bases de données académiques pertinentes. Toutes les sources citées dans ce travail ont été consultées, vérifiées et analysées directement par mes soins. D'autre part, j'ai utilisé ponctuellement des outils d'assistance linguistique pour la relecture orthographique et grammaticale de passages que j'avais rédigés moi-même, sans qu'aucun contenu ne soit ajouté ou modifié sur le fond.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl