Une image contenant texte, Police, capture d’écran, nombre

Le contenu généré par l’IA peut être incorrect.Chatbot intelligent pour le diagnostic de pannes électroniques via NLP

Travail de Bachelor réalisé en vue de l’obtention du Bachelor HES

par :

Corentin ROMANIELLO

Conseiller au travail de Bachelor :

Fredrik LAHODE, Chargé de cours HES

< Lieu, date de dépôt >

Haute École de Gestion de Genève (HEG-GE)

Filière Informatique de gestion

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l’examen final de la Haute école de gestion de Genève, en vue de l’obtention du titre Bachelor of Science HES-SO en Informatique de gestion.

L’étudiant a envoyé ce document par email à l'adresse remise par son directeur de mémoire afin qu’il l’analyse à l’aide du logiciel de détection de plagiat COMPILATIO.

L’étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J’atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à < indiquer le lieu >, le < date >

< Votre prénom et votre nom >

< Signez la déclaration ici >

Remerciements

Si vous avez des remerciements à formuler, à l’entreprise ou à toute autre personne qui a pu vous aider dans la réalisation du travail.

Les remerciements sont rédigés dans le style « **Corps de texte** ».

Résumé

Ce travail de Bachelor porte sur le développement d’un chatbot capable d’aider un utilisateur à identifier une panne électronique en formulant simplement une question en langage naturel. L’objectif principal était d’évaluer si une approche basée sur la récupération de documents (RAG - Retrieval-Augmented Generation) pouvait suffire pour fournir des réponses pertinentes, sans devoir entraîner un modèle de langage de zéro.

Pour cela, j’ai utilisé deux types de sources : d’une part les guides structurés disponibles sur le site iFixit, et d’autre part des échanges techniques issus du forum Reddit (r/techsupport), qui représentent des cas concrets et des formulations variées. Ces données ont été traitées, découpées, puis indexées avec FAISS pour permettre leur recherche rapide selon les besoins de l’utilisateur.

Le système s’appuie sur l’outil LangChain, qui permet de connecter les différentes étapes du pipeline : récupération des documents, génération du prompt, puis création de la réponse par un modèle de langage. Plusieurs modèles ont été testés, notamment Mistral, DeepSeek et GPT-4.1, afin de comparer leurs performances dans ce contexte. Certains se sont montrés plus efficaces que d’autres, en particulier sur la capacité à exploiter correctement les documents fournis.

Au fil du projet, plusieurs améliorations ont été intégrées : génération automatique de variantes de la question (query expansion), ajout dynamique des étapes détaillées des guides iFixit, et reformulation du prompt pour éviter les réponses hors sujet. Le tout a été regroupé dans un prototype fonctionnel, accessible via une interface web locale, permettant à l’utilisateur de poser une question et de visualiser les sources utilisées pour la réponse.

Ce projet m’a permis d’acquérir une meilleure compréhension du fonctionnement des modèles de langage, ainsi que des moyens concrets pour les adapter à des cas pratiques, sans recourir à des techniques lourdes comme le fine-tuning.

Table des matières

[Déclaration 1](#_Toc200291207)

[Remerciements 2](#_Toc200291208)

[Résumé 3](#_Toc200291209)

[Liste des tableaux 7](#_Toc200291210)

[Liste des figures 7](#_Toc200291211)

[1. Introduction 9](#_Toc200291212)

[1.1 Contexte et enjeux 9](#_Toc200291213)

[1.2 Objectifs du projet 9](#_Toc200291214)

[1.3 Problématiques, hypothèses et question de recherche 9](#_Toc200291215)

[1.4 Démarche méthodologique 10](#_Toc200291216)

[1.5 Plan du document 11](#_Toc200291217)

[2. Conception du système 12](#_Toc200291218)

[2.1 Planification et suivi 12](#_Toc200291219)

[2.2 Outils et configuration utilisée 12](#_Toc200291220)

[2.3 Architecture globale du prototype 13](#_Toc200291221)

[2.4 Dépendances techniques principales 13](#_Toc200291222)

[2.4.1 LangChain 14](#_Toc200291223)

[2.4.2 FAISS 14](#_Toc200291224)

[2.4.3 Ollama 14](#_Toc200291225)

[2.4.4 OpenAI API 15](#_Toc200291226)

[2.4.5 Hugging Face Embeddings 15](#_Toc200291227)

[2.4.6 PRAW 15](#_Toc200291228)

[2.5 Suivi et livrables 16](#_Toc200291229)

[3. Revue de littérature 16](#_Toc200291230)

[3.1 Chatbots et évolution historique 16](#_Toc200291231)

[3.2 Grands modèles de langage (LLM) 17](#_Toc200291232)

[3.3 Fine-tuning des LLM 18](#_Toc200291233)

[3.4 Retrieval-Augmented Generation (RAG) 19](#_Toc200291234)

[4. Source de données 20](#_Toc200291235)

[4.1 Guides iFixit 20](#_Toc200291236)

[4.2 Posts Reddit Tech Support 21](#_Toc200291237)

[5. Construction de la solution 22](#_Toc200291238)

[5.1 Structuration des documents et indexation 22](#_Toc200291239)

[5.1.1 Indexation des guides iFixit 22](#_Toc200291240)

[5.1.2 Indexation des posts Reddit Tech Support 24](#_Toc200291241)

[5.2 Influence de la langue sur les résultats 25](#_Toc200291242)

[5.3 Comparaison avec un modèle statistique 26](#_Toc200291243)

[5.3.1 Temps d’indexation 27](#_Toc200291244)

[5.4 Comprendre la différence entre vector store et retriever 28](#_Toc200291245)

[5.5 Construction du pipeline RAG 28](#_Toc200291246)

[5.5.1 Étape 1 - Indexation unifiée et création du retriever 28](#_Toc200291247)

[5.5.2 Étape 2 - Définition du LLM local via Ollama 31](#_Toc200291248)

[5.5.3 Étape 3 - Construction du prompt (version initiale) 31](#_Toc200291249)

[5.5.4 Étape 4 - Chaînage des composants avec create\_rag\_chain 32](#_Toc200291250)

[5.5.5 Étape 5 - Exécution du pipeline avec .invoke() 33](#_Toc200291251)

[5.6 Modules spécifiques : intégration dynamique des étapes iFixit 34](#_Toc200291252)

[5.6.1 Implémentation d’une fonction spécifique 35](#_Toc200291253)

[5.6.2 Modification de format\_documents() 35](#_Toc200291254)

[5.6.3 Adaptation du prompt 36](#_Toc200291255)

[5.6.4 Résultat et analyse 36](#_Toc200291256)

[6. Expérimentation et évaluation 38](#_Toc200291257)

[6.1 Constat initial 38](#_Toc200291258)

[6.2 Comparaison entre plusieurs LLM 38](#_Toc200291259)

[6.2.1 Modèle 1 - llama3.1:8b 39](#_Toc200291260)

[6.2.2 Modèle 2 - mistral:7b 39](#_Toc200291261)

[6.2.3 Modèle 3 - deepseek-r1:32b 40](#_Toc200291262)

[6.2.4 Modèle 4 - gpt-4.1 (API OpenAI) 42](#_Toc200291263)

[6.3 Tableau comparatif 43](#_Toc200291264)

[6.4 Analyse générale 43](#_Toc200291265)

[7. Amélioration du pipeline RAG 44](#_Toc200291266)

[7.1 Post-récupération : un premier pas vers le RAG avancé 44](#_Toc200291267)

[7.2 Refonte du prompt pour limiter les hallucinations 44](#_Toc200291268)

[7.3 Changement de modèle d’embeddings 45](#_Toc200291269)

[7.4 Implémentation complète de la Query Expansion 45](#_Toc200291270)

[7.4.1 Objectif 45](#_Toc200291271)

[7.4.2 Étape 1 - Génération des variantes de la question 46](#_Toc200291272)

[7.4.3 Étape 2 - Récupération multi-requêtes 47](#_Toc200291273)

[7.4.4 Étape 3 - Filtrage par score de similarité 47](#_Toc200291274)

[7.4.5 Résultat final 48](#_Toc200291275)

[7.5 Synthèse 49](#_Toc200291276)

[8. Discussion 49](#_Toc200291277)

[8.1 Récupération - Le retriever suffit-il à isoler les bons documents ? 50](#_Toc200291278)

[8.2 Modèle génératif - Tous les LLMs peuvent-ils exploiter le contexte ? 50](#_Toc200291279)

[8.3 Formulation de la requête - Impact sur la récupération et la génération 50](#_Toc200291280)

[8.4 Structuration du prompt - Peut-on vraiment éviter les hallucinations ? 51](#_Toc200291281)

[8.5 Performance globale - Un bon compromis sans fine-tuning ? 51](#_Toc200291282)

[9. Présentation du prototype final 52](#_Toc200291283)

[9.1 Vue d’ensemble 52](#_Toc200291284)

[9.2 Architecture générale 53](#_Toc200291285)

[9.3 Fonctionnement du pipeline intégré 54](#_Toc200291286)

[9.4 Interface utilisateur 55](#_Toc200291287)

[9.5 Déploiement et configuration 56](#_Toc200291288)

[9.6 Documentation associée 56](#_Toc200291289)

[10. Conclusion 57](#_Toc200291290)

[10.1 Bilan et apports techniques 57](#_Toc200291291)

[10.2 Retours personnels 57](#_Toc200291292)

[10.3 Ouvertures possibles 58](#_Toc200291293)

[10.3.1 Ajout de sources supplémentaires 58](#_Toc200291294)

[10.3.2 Déploiement cloud + CI/CD 58](#_Toc200291295)

[10.3.3 Multilingue 58](#_Toc200291296)

[10.3.4 Interaction avec l’utilisateur 58](#_Toc200291297)

Liste des tableaux

[Tableau 1 : Planification globale 12](#_Toc200291298)

[Tableau 2 - Comparatif des modèles génératifs testés 43](#_Toc200291299)

Liste des figures

[Figure 1 : Modèle de récupération des guides iFixit basé sur embeddings 23](#_Toc200291300)

[Figure 2 : Modèle de récupération des posts Reddit basé sur embeddings 25](#_Toc200291301)

[Figure 3 : Test du modèle de récupération des guides iFixit basé sur embeddings en français 26](#_Toc200291302)

[Figure 4 : Test du modèle de récupération des guides iFixit basé sur embeddings en anglais 26](#_Toc200291303)

[Figure 5 : Modèle de récupération des guides iFixit basé sur statistiques 27](#_Toc200291304)

[Figure 6 : Retriever embeddings 28](#_Toc200291305)

[Figure 7 : Retriever « BM25 » 28](#_Toc200291306)

[Figure 8 : Modèle de récupération des posts et guides dans le pipeline RAG 30](#_Toc200291307)

[Figure 9 : Classe LangChain dérivée pour utiliser un LLM via Ollama 31](#_Toc200291308)

[Figure 10 : Prompt du LLM dans la chaîne RAG 32](#_Toc200291309)

[Figure 11 : Formatage des documents retourné par « retriever » et fonction création pipeline RAG 33](#_Toc200291310)

[Figure 12 : Exécution pipeline RAG 34](#_Toc200291311)

[Figure 13 : Fonction retournant les étapes d’un guide iFixit 35](#_Toc200291312)

[Figure 14 : Formatage des documents retourné par « retriever » avec étapes pour les guides iFixit 36](#_Toc200291313)

[Figure 15 : Utilisation étapes guides iFixit dans prompt du LLM dans la chaîne RAG 36](#_Toc200291314)

[Figure 16 : Contenu guide iFixit retourné par « retriever » avec étapes données au contexte 37](#_Toc200291315)

[Figure 17 : Sortie pipeline RAG avec étapes guides iFixit 38](#_Toc200291316)

[Figure 18 : Sortie pipeline RAG avec modèle « llama3.1:8b » 39](#_Toc200291317)

[Figure 19 : Temps exécution pipeline RAG avec modèle « llama3.1:8b » 39](#_Toc200291318)

[Figure 20 : Sortie pipeline RAG avec modèle « mistral:7b » 40](#_Toc200291319)

[Figure 21 : Temps exécution pipeline RAG avec modèle « mistral:7b » 40](#_Toc200291320)

[Figure 22 : Sortie pipeline RAG avec modèle « deepseek-r1:32b » 41](#_Toc200291321)

[Figure 23 : Réflexion modèle « deepseek-r1:32b » pipeline RAG 41](#_Toc200291322)

[Figure 24 : Temps exécution pipeline RAG avec modèle « deepseek-r1:32b » 42](#_Toc200291323)

[Figure 25 : Sortie pipeline RAG avec modèle « gpt-4.1 » 42](#_Toc200291324)

[Figure 26 : Temps exécution pipeline RAG avec modèle « gpt-4.1 » 43](#_Toc200291325)

[Figure 27 : Modèle « gpt-4.1 » instancié via « ChatOpenAI » 43](#_Toc200291326)

[Figure 28 : Prompt du LLM dans la chaîne RAG avancé 45](#_Toc200291327)

[Figure 29 : Prompt LLM « Query Expansion » 46](#_Toc200291328)

[Figure 30 : Chaîne de génération des questions variantes 46](#_Toc200291329)

[Figure 31 : Générations des questions variantes 46](#_Toc200291330)

[Figure 32 : Chaîne de récupération 47](#_Toc200291331)

[Figure 33 : Fonction retournant une liste de documents uniques à partir de listes imbriquées 47](#_Toc200291332)

[Figure 34 : Configuration du score de similarité minimal pour filtrer les documents retournés par le « retriever » 47](#_Toc200291333)

[Figure 35 : Exécution pipeline RAG avancé 48](#_Toc200291334)

[Figure 36 : Sortie pipeline RAG avancé (1) 48](#_Toc200291335)

[Figure 37 : Sortie pipeline RAG avancé (2) 49](#_Toc200291336)

[Figure 38 : Schéma de l’architecture du prototype 54](#_Toc200291337)

[Figure 39 : Capture d’écran de l’interface du prototype en fonctionnement 56](#_Toc200291338)

# Introduction

## Contexte et enjeux

De nos jours, on utilise tous des appareils électroniques au quotidien : téléphones, ordinateurs, tablettes, etc. Le problème, c’est qu’on a tendance à les remplacer très vite, même quand ils fonctionnent encore. Résultat, on produit énormément de déchets électroniques. Ça pose évidemment des questions environnementales importantes, notamment sur la manière dont on pourrait limiter cet impact.

Dans ce cadre, on entend de plus en plus parler d’économie circulaire. L’idée, c’est de sortir du modèle classique « je consomme, je jette » pour aller vers quelque chose de plus durable. Par exemple, en encourageant la réparation plutôt que le remplacement. Mais réparer un appareil, ça suppose de savoir ce qui ne va pas, et là, c’est souvent compliqué pour le grand public.

C’est justement à ce niveau que l’intelligence artificielle peut apporter quelque chose. Grâce au traitement du langage naturel (NLP), il devient possible de créer des outils capables de comprendre les problèmes décrits par un utilisateur et de proposer des solutions. Dans ce mémoire, je me suis intéressé à la création d’un assistant basé sur un modèle de langage, qui pourrait guider une personne dans le diagnostic d’une panne et l’aider à la résoudre.

## Objectifs du projet

L’objectif principal de ce projet est de concevoir un assistant intelligent, sous forme de chatbot, capable de comprendre les questions des utilisateurs liées à une panne et de leur fournir des pistes concrètes pour la résoudre.

Pour ça, l’outil s’appuie sur des données déjà disponibles, comme les guides de réparation d’iFixit ou les discussions techniques trouvées sur Reddit. L’idée, c’est de ne pas réinventer la roue, mais de rendre accessibles ces informations à travers un système simple d’utilisation.

## Problématiques, hypothèses et question de recherche

Ce projet se situe à la croisée de deux approches bien différentes pour la génération automatisée de réponses techniques :

* d’un côté, le fine-tuning, qui consiste à réentraîner un modèle de langage pour l’adapter à un domaine spécifique,
* de l’autre, l’architecture RAG (Retrieval-Augmented Generation), qui permet de garder un modèle généraliste, mais de lui fournir des documents pertinents au moment de la génération.

L'enjeu est de savoir si une architecture RAG avancée peut suffire à elle seule à produire des réponses utiles et fiables, sans passer par une phase d'entraînement du modèle.

Cela a mené à la formulation d’une hypothèse centrale :

Hypothèse : Il est possible de produire des réponses pertinentes à des questions techniques, sans entraîner un modèle, en construisant un pipeline RAG avancé qui exploite correctement des documents externes.

Pour tester cette hypothèse, cinq sous-questions de recherche ont été posées :

1. Récupération : Est-ce qu’un bon système de récupération (retriever + embeddings) permet réellement d’identifier les documents les plus utiles pour une question donnée, sans intervention manuelle ?
2. Modèle génératif : Tous les modèles de langage sont-ils capables d’exploiter efficacement les documents injectés dans un prompt RAG ?
3. Formulation de la requête : Dans quelle mesure la formulation de la question utilisateur impacte-t-elle la qualité des documents récupérés et donc de la réponse générée ?
4. Structuration du prompt : Est-ce qu’un prompt bien formulé suffit à garantir que le LLM s’appuie uniquement sur les documents fournis, sans générer d’informations erronées ?
5. Performance globale : Quel équilibre entre précision, vitesse et coût peut-on atteindre avec un pipeline RAG local ou hybride, sans fine-tuning, mais avec des modules avancés comme la query expansion ou la post-récupération ?

Ces cinq aspects forment la colonne vertébrale du mémoire et structurent l’ensemble des expérimentations, comparaisons et améliorations présentées dans les chapitres suivants.

## Démarche méthodologique

Pour gérer le projet, j’ai utilisé la méthode SCRUM, qui est assez répandue dans le domaine du développement. Elle permet de travailler par étapes, qu’on appelle des « sprints », avec à chaque fois des objectifs clairs. À la fin de chaque sprint, on peut faire le point sur ce qui a été accompli et ajuster si besoin.

D’un point de vue technique, j’ai exploré deux grandes approches :

* Le fine-tuning, qui consiste à réentraîner un modèle sur un domaine spécifique. C’est une méthode puissante, mais elle demande beaucoup de ressources (temps, données, puissance de calcul), ce que je n’avais pas forcément à disposition.
* L’approche Retrieval-Augmented Generation (RAG), qui permet d’utiliser un modèle généraliste en lui fournissant des documents pertinents à consulter au moment de la réponse. C’est plus léger, plus flexible, et adapté à un contexte où les données sont variées et pas forcément centralisées.

Au final, c’est l’approche RAG que j’ai choisi, non pas parce qu’elle est meilleure par défaut, mais parce qu’elle correspondait mieux aux contraintes du projet. Ce choix sera expliqué plus en détail dans la suite du mémoire, à travers les essais que j’ai réalisés et les résultats obtenus.

## Plan du document

Ce mémoire est structuré en dix chapitres :

* Le chapitre 2 explique comment le projet a été organisé et présente les outils et l’architecture du système.
* Le chapitre 3 est une revue de littérature sur les chatbots, les modèles de langage, le fine-tuning et l’approche RAG.
* Le chapitre 4 présente les sources utilisées : les guides iFixit et les posts Reddit.
* Le chapitre 5 décrit comment ces données ont été préparées, structurées et indexées.
* Le chapitre 6 montre comment la chaîne RAG a été construite et comment elle fonctionne.
* Le chapitre 7 présente les tests réalisés pour évaluer les performances du système.
* Le chapitre 8 propose des améliorations basées sur les résultats obtenus.
* Le chapitre 9 décrit le prototype final et la documentation associée.
* Le chapitre 10 conclut en résumant ce qui a été fait et en ouvrant sur d’éventuelles perspectives.

Une section est aussi dédiée aux dépendances techniques, et des annexes sont présentes à la fin avec du code, des schémas, etc.

# Conception du système

## Planification et suivi

Pour structurer l’avancement du projet, j’ai découpé le travail en sept sprints, chacun avec un objectif clair :

Tableau  : Planification globale

|  |  |  |
| --- | --- | --- |
| **Sprint** | **Dates** | **Objectifs principaux** |
| 1 | 3 mars - 15 mars | Étude de la littérature sur les chatbots, le NLP, les LLM et les différentes approches |
| 2 | 17 mars - 29 mars | Collecte des sources de données (iFixit, Reddit, etc.) et structuration |
| 3 | 31 mars - 12 avril | Indexation des données et préparation du pipeline |
| 4 | 14 avril - 3 mai | Développement du premier prototype du chatbot, intégration des modèles NLP/LLM |
| 5 | 5 mai - 24 mai | Optimisations, ajustements, amélioration de la pertinence des réponses |
| 6 | 26 mai - 7 juin | Création d’une interface utilisateur simple pour utiliser le chatbot |
| 7 | 9 juin - 20 juin | Finalisation, tests globaux et préparation de la soutenance |

(Planification)

## Outils et configuration utilisée

Pour mener à bien ce projet, j’ai utilisé plusieurs outils, chacun pour un aspect bien précis du développement.

* LangChain : pour construire la chaîne RAG et organiser les étapes du pipeline (retrieval, prompt, génération).
* Ollama : pour exécuter localement les modèles de langage (comme Mistral, Llama 3, etc.) sans dépendre d’une API payante.
* FAISS : utilisé pour l’indexation et la recherche de documents à partir des embeddings.
* Flask : pour mettre en place une API simple, accessible via le navigateur.
* API iFixit : pour récupérer des guides de réparation détaillés.
* PRAW : pour extraire des posts techniques à partir du subreddit r/techsupport.

Concernant la configuration de la machine utilisée pour tout le développement :

* Processeur : Intel Core i5 (12ᵉ génération)
* Mémoire RAM : 32 Go
* Carte graphique : NVIDIA RTX 4060 avec 8 Go de VRAM

Cette configuration s’est avérée suffisante pour faire tourner les modèles d’IA de taille moyenne (jusqu’à 8B) en local, même si certains modèles plus lourds, comme LLaMA 70B, restaient inexploitables avec mes ressources.

## Architecture globale du prototype

Le système mis en place repose sur une architecture en plusieurs couches, relativement simple, mais efficace pour tester différentes combinaisons de modèles.

1. Collecte des données : via les API iFixit et Reddit. Les contenus sont ensuite stockés en JSON pour être utilisés lors de l’indexation.
2. Préparation des documents : découpe des textes en fragments, génération des embeddings, et stockage dans FAISS.
3. Création d’un retriever : capable de chercher les fragments les plus pertinents selon la question de l’utilisateur.
4. Construction du pipeline RAG : intégration du contexte dans un prompt et envoi au LLM pour générer une réponse.
5. Interface utilisateur : simple interface HTML connectée à une API Flask locale, permettant à l’utilisateur de poser sa question.

Ce découpage m’a permis de tester chaque bloc séparément et de faire évoluer certains composants (comme le modèle de LLM) sans avoir à tout recommencer à zéro.

## Dépendances techniques principales

Pour construire le pipeline RAG de manière modulaire et maintenable, plusieurs dépendances logicielles ont été intégrées au projet. Cette section décrit les principales bibliothèques utilisées, ainsi que leur rôle spécifique dans le système.

### LangChain

LangChain[[1]](#footnote-1) constitue le socle de l’orchestration du pipeline. Il a permis de chaîner plusieurs étapes (retrieval, prompt, génération) en une suite cohérente, pilotée de manière modulaire. Par exemple, le PromptTemplate est instancié via langchain.prompts[[2]](#footnote-2) pour formater dynamiquement la requête adressée au LLM. J’ai également utilisé RunnablePassthrough (de langchain.schema.runnable[[3]](#footnote-3)) pour relayer certains éléments tels quels entre les étapes, et StrOutputParser (via langchain\_core.output\_parsers[[4]](#footnote-4)) pour structurer proprement la sortie textuelle du modèle.

LangChain a aussi été essentiel pour intégrer les retrievers (comme FAISS ou BM25) dans le pipeline, et pour manipuler l’API du LLM de façon centralisée.

### FAISS

FAISS (Facebook AI Similarity Search) est utilisé comme base vectorielle dans laquelle sont indexés tous les documents textuels (guides iFixit, posts Reddit). Instanciée via langchain\_community.vectorstores[[5]](#footnote-5), cette dépendance permet de retrouver rapidement les documents les plus proches sémantiquement d’une requête utilisateur. Elle est alimentée par des vecteurs générés avec des modèles d’embeddings et interrogée par le retriever via des requêtes simples ou multiples (notamment après expansion de requête).

### Ollama

Ollama[[6]](#footnote-6) m’a permis de faire tourner localement des modèles de langage open source comme llama3.1:8b ou mistral:7b. Grâce à la dépendance ollama-python[[7]](#footnote-7), j’ai pu instancier mes modèles de génération via un serveur local, évitant ainsi d’utiliser une API payante. Cela a été très utile pour faire des tests sur machine personnelle, en conservant un contrôle total sur les performances, la langue de génération et le comportement du LLM.

Ce système est compatible avec LangChain, et j’ai pu l’intégrer comme source directe de génération dans le pipeline.

### OpenAI API

Dans la dernière phase du projet, j’ai utilisé l’API d’OpenAI[[8]](#footnote-8) pour appeler le modèle GPT-4.1. Ce modèle a été sélectionné suite à une série de tests comparatifs qui ont démontré sa capacité à exploiter correctement les documents injectés, à répondre en français, et à structurer les réponses de manière claire et pertinente.

L’appel au modèle GPT-4.1 a été effectué via la classe ChatOpenAI de langchain\_openai[[9]](#footnote-9), qui facilite l’intégration de l’API dans un flux LangChain. Cette dépendance m’a permis de bénéficier d’un LLM très performant sans avoir à le faire tourner localement, ce qui a fortement réduit les temps de traitement et augmenté la qualité des réponses finales.

### Hugging Face Embeddings

Pour transformer mes documents en vecteurs utilisables dans FAISS, j’ai utilisé la classe HuggingFaceEmbeddings, instanciée via langchain\_huggingface.embeddings[[10]](#footnote-10). Deux modèles ont été testés : all-mpnet-base-v2 (modèle robuste, mais lent) et MiniLM-L6-v2 (modèle plus léger mais presque aussi pertinent). Ce choix a permis de comparer l’efficacité des embeddings dans les tests de récupération.

Le module permet un appel rapide aux modèles hébergés sur Hugging Face, sans avoir besoin d’entraîner quoi que ce soit.

### PRAW

Pour la récupération des données depuis Reddit (r/techsupport), j’ai utilisé la bibliothèque PRAW[[11]](#footnote-11). Elle permet de récupérer facilement les titres, corps et commentaires des posts. Ces contenus, une fois nettoyés, ont été transformés et ajoutés à l’index vectoriel FAISS au même titre que les guides techniques. Cette couche communautaire complète bien les sources plus formelles comme iFixit.

L’ensemble de ces dépendances forme l’ossature technique du système développé. Chacune a été choisie pour sa compatibilité avec l’écosystème Python, sa capacité à s’intégrer dans un pipeline LangChain, et la richesse de sa documentation. Leur combinaison a permis de concevoir un pipeline RAG avancé, réactif, flexible, et modifiable sans réécriture lourde.

## Suivi et livrables

* Mémoire : Document de synthèse expliquant la démarche, les choix technologiques et les résultats obtenus.
* Archive contenant le code source de l’application : Contenu du dépôt GitHub contenant le code source de l’application.
* Documentation technique : Explication détaillée des modèles NLP, du pipeline de données et des API utilisées.
* Notebooks Jupyter au format PDF et Python.

Ce cadre méthodologique structuré assure une gestion efficace du projet, en garantissant une progression régulière et en facilitant les ajustements nécessaires pour maximiser la pertinence des solutions développées.

# Revue de littérature

## Chatbots et évolution historique

Les chatbots existent depuis longtemps, bien avant les modèles modernes. Le tout premier qui est souvent cité est ELIZA, conçu dans les années 1960. Ce programme simulait un psychothérapeute, en posant des questions ouvertes, ce qui lui permettait de masquer ses limites. Comme l’explique Zemčík (2019, p. 15), ELIZA « simulates a role of a Rogerian psychotherapist - she asks open questions with which she also answers - thus she diverts attention from herself to the user. »

Un peu plus tard, PARRY proposait une autre approche. Au lieu de se comporter comme un thérapeute, il simulait un patient atteint de schizophrénie paranoïde, pour obtenir des réponses plus développées. « It tries to provoke controversies and thus make the participant give more elaborate answers. » (Zemčík 2019, p. 15)

Dans les années 80-90, on voit apparaître des outils comme Racter, capable de générer du texte en anglais de manière presque aléatoire, et Dr. Sbaitso, qui avait la particularité de pouvoir synthétiser de la parole. Même si leurs capacités restaient limitées, ils ont marqué une étape importante dans la construction des premiers agents conversationnels.

C’est dans les années 2010 que les chatbots deviennent plus accessibles au grand public, avec Siri, Google Now ou Cortana. Ils comprenaient mieux la voix, mais restaient souvent limités aux commandes simples. D’autres chatbots sociaux, comme Xiaoice, ont aussi vu le jour. Celui-ci, programmé pour agir comme une jeune fille de 17 ans sur les réseaux chinois, visait à créer un lien émotionnel avec les utilisateurs (Zemčík 2019, p. 14). Microsoft a aussi développé Tay, un chatbot qui apprenait en ligne, mais qui a été retiré à cause de dérives liées à l’apprentissage non supervisé (Zemčík 2019, p. 15).

Ces outils posaient déjà certains problèmes : ils avaient du mal à comprendre les nuances du langage ou à suivre un fil de discussion cohérent. C’est à partir de là que sont apparus les grands modèles de langage, pour essayer de mieux gérer ces situations complexes.

## Grands modèles de langage (LLM)

Les LLM, ou grands modèles de langage, ont complètement changé la donne. Ils sont capables de comprendre et de produire du texte avec une qualité bien supérieure aux anciens systèmes. Le point de départ de cette avancée, c’est l’architecture Transformer, présentée par Vaswani et al. (2023, p. 1) : « We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. »

Cette architecture permet une exécution plus rapide et surtout plus efficace, car elle est entièrement basée sur l’attention. Vaswani explique que le Transformer « allows for significantly more parallelization and can reach a new state of the art in translation quality… » (Vaswani et al. 2023, p. 2)

Un exemple marquant de LLM est GPT-3, avec ses 175 milliards de paramètres. Brown et al. (2020, p. 5) expliquent que ce modèle peut faire des traductions, répondre à des questions, faire du raisonnement, parfois même avec très peu d’exemples : « GPT-3 also displays one-shot and few-shot proficiency at tasks designed to test rapid adaption or on-the-fly reasoning… »

Les modèles fondation, comme le définit Bommasani et al. (2022, p. 3), sont des modèles polyvalents, qu’on peut adapter à différents usages. Ils parlent d’émergence, où le comportement n’est pas programmé, mais apparaît tout seul à l’entraînement : « Emergence means that the behavior of a system is implicitly induced rather than explicitly constructed. »

Une autre notion clé est l’homogénéisation : on applique la même méthode (les transformers) à des domaines différents (Bommasani et al. 2022, p. 3). Ces deux concepts permettent de comprendre pourquoi les LLM sont devenus un standard dans l’IA aujourd’hui.

L'évolution a aussi été marquée par une montée en puissance des capacités. Zhao et al. (2025, p. 3) soulignent que les LLM affichent désormais des compétences inédites : « LLMs display some surprising emergent abilities that may not be observed in previous smaller PLMs. » Et ces modèles s’utilisent souvent via une interface de type prompting, ce qui les rend beaucoup plus flexibles : « The major approach to accessing LLMs is through the prompting interface. »

Dans l’étude de Dam et al. (2024, p. 1), on retrouve cette idée : les LLM sont devenus essentiels pour traiter le langage humain, que ce soit pour la génération de texte, la compréhension ou la recherche. « Thanks to their exceptional ability to understand, generate, and manipulate human language… » Ils sont utilisés dans des domaines aussi variés que la recherche, l’éducation ou la santé.

Mais tout n’est pas parfait. Deux problèmes majeurs ressortent souvent : la difficulté à mettre à jour les connaissances d’un modèle, et le phénomène d’hallucination.

Sur le premier point, Dam et al. (2024, p. 13) soulignent que « Maintaining up-to-date knowledge presents a notable challenge… » Les LLM sont figés au moment de leur entraînement, ce qui limite leur pertinence pour des informations récentes.

Le deuxième problème est l’hallucination. Il y a deux types : intrinsèque (l’information générée contredit des sources fiables) et extrinsèque (l’information ne peut pas être vérifiée). « LLM-based chatbots face a notable challenge in producing factual texts due to hallucination… » (Dam et al. 2024, p. 14)

Ces limites sont importantes à connaître avant de choisir une méthode comme le fine-tuning ou le RAG.

## Fine-tuning des LLM

Le fine-tuning, c’est le fait de reprendre un modèle existant et de le réentraîner sur un domaine spécifique. Cela permet au modèle d’être plus précis dans un contexte donné. Par exemple, dans ce projet, on aurait pu l’entraîner avec des questions sur des pannes électroniques et leurs solutions.

Aïcha (2024) définit le fine-tuning comme un « procédé d'enrichissement du LLM […] pour qu'il puisse comprendre et générer du langage humain de manière pertinente et cohérente dans un contexte donné. »

Pour faire cela, il faut un dataset bien construit, avec des exemples concrets. Dans notre cas, des paires question - réponse sur des pannes matérielles auraient été idéales. Mais cela demande du temps, des ressources, et beaucoup de données bien annotées.

C’est pour cette raison que le fine-tuning n’a pas été retenu ici. À la place, j’ai choisi d’utiliser la méthode RAG, qui est plus souple et plus légère à mettre en place, tout en restant efficace.

## Retrieval-Augmented Generation (RAG)

La méthode RAG (Retrieval-Augmented Generation) permet de générer une réponse en s’appuyant sur des documents externes. Contrairement au fine-tuning, elle ne modifie pas le modèle, mais l’alimente avec du contexte pertinent.

Barkova (2023) explique que cette méthode permet d’utiliser un LLM dans un contexte spécifique « sans engendrer des coûts liés à l’entraînement… »

Le principe est simple : les documents sont encodés avec un modèle d’embedding, puis stockés dans une base vectorielle. Lorsqu’un utilisateur pose une question, elle est transformée en vecteur, comparée aux documents, et les meilleurs sont injectés dans le prompt du LLM.

Selon Gao et al. (2024, p. 1), cette méthode améliore les performances des LLM en allant chercher des documents pertinents dans une base de données externe : « RAG enhances LLMs by retrieving relevant document chunks from external knowledge base… »

Ils décrivent trois types de RAG :

* Le RAG naïf, avec un fonctionnement simple basé sur la suite indexation → récupération → génération.
* Le RAG avancé, qui ajoute des optimisations (meilleure formulation des requêtes, compression de contexte…).
* Le RAG modulaire, encore plus flexible, où on peut interchanger certains composants, ou utiliser des modules spécialisés.

Le RAG modulaire permet aussi des recherches itératives, où le système affine la réponse en fonction de ce qu’il découvre, et même des recherches adaptatives, comme dans Flare ou Self-RAG (Gao et al. 2024, p. 5).

Enfin, ils comparent RAG et fine-tuning :

« RAG excels in dynamic environments by offering real-time knowledge updates… FT is more static, requiring retraining for updates but enabling deep customization… »

Dans mon projet, la méthode RAG a été privilégiée pour toutes ces raisons. Elle permet de s’appuyer sur des contenus actualisés (comme les guides iFixit ou Reddit) et de les intégrer directement dans la réponse générée par le modèle.

# Source de données

## Guides iFixit

Pour alimenter le système avec des données fiables et bien structurées, j’ai utilisé les guides proposés par le site iFixit[[12]](#footnote-12). Ce site est une vraie référence en matière de réparation. Il propose des tutoriels détaillés pour réparer une grande variété d’appareils électroniques : smartphones, ordinateurs, consoles, etc.

L’intérêt principal d’iFixit, c’est que les contenus sont très bien organisés. Chaque guide se concentre sur un problème ou une opération spécifique (remplacement de batterie, changement d’écran, etc.) et suit une structure claire. On y trouve des étapes numérotées, des titres, des images, des descriptions d’outils nécessaires, des niveaux de difficulté, et même des avertissements.

L’API publique d’iFixit[[13]](#footnote-13) permet d’accéder librement à ces guides. Elle fournit les informations au format JSON, ce qui est pratique pour les exploiter dans une application. Voici un exemple de structure retournée par l’API pour un guide :

* title : le titre du guide (ex. « How to replace a MacBook battery »)
* subject : Le sujet du guide
* guideid : L’identifiant du guide
* dataType : Le type de document
* steps : la liste des étapes à suivre
* summary : un résumé global du guide
* category : La catégorie du guide
* url : L’URL du guide
* type : Le type de maintenance

Dans ce projet, j’ai récupéré plusieurs centaines de guides iFixit à l’aide de cette API. Ensuite, je les ai transformés en documents exploitables par le système RAG.

Ce choix de source était stratégique. D’une part, les guides sont bien rédigés et accessibles. D’autre part, ils couvrent une large gamme de produits, ce qui augmente les chances que l’utilisateur trouve une réponse adaptée.

Enfin, comme les guides iFixit sont régulièrement mis à jour, cela permet d’avoir une base de données toujours actuelle - ce qui est particulièrement intéressant dans une approche comme RAG, où les documents externes jouent un rôle essentiel.

## Posts Reddit Tech Support

En plus des guides officiels, j’ai choisi d’intégrer des posts issus du forum Reddit, plus précisément du subreddit r/techsupport[[14]](#footnote-14). Ce forum est dédié à l’entraide technique, et on y trouve de nombreux cas réels, parfois très proches de ce que vivent les utilisateurs de tous les jours.

L’objectif ici était de récupérer des problèmes concrets, posés en langage naturel, et les réponses données par la communauté. Contrairement aux guides iFixit qui sont structurés et validés, les posts Reddit sont plus désordonnés, mais ils reflètent mieux la réalité de ce qu’un utilisateur peut écrire dans un chatbot.

Pour extraire ces données, j’ai utilisé la bibliothèque PRAW (Python Reddit API Wrapper). Elle permet de se connecter à l’API Reddit et de récupérer facilement des publications et leurs commentaires.

Voici ce que j’ai récupéré pour chaque post :

* le titre, qui contient généralement un résumé du problème,
* le contenu du post (souvent une description plus longue, parfois avec du contexte, des erreurs rencontrées, ce qui a été testé, etc.),
* les commentaires, qui peuvent inclure des suggestions, des conseils, voire des diagnostics.
* l’URL du post original

Chaque post a ensuite été transformé en un document, utilisable pour l’indexation dans FAISS.

Ce corpus complémentaire permet d’enrichir le système avec des formulations très diverses. Il aide aussi à faire le lien entre la manière dont un utilisateur pourrait formuler un problème, et la manière dont il pourrait être résolu, ce qui est essentiel dans une approche NLP.

# Construction de la solution

## Structuration des documents et indexation

### Indexation des guides iFixit

La première étape pour construire une chaîne RAG fonctionnelle a consisté à structurer les données issues de l’API iFixit. L’objectif était d’obtenir une base exploitable, avec des documents bien découpés, enrichis en métadonnées et indexés efficacement.

Pour cela, chaque guide iFixit a été transformé en un objet Document[[15]](#footnote-15) de LangChain, avec deux éléments principaux :

* Le champ page\_content, dans lequel j’ai concaténé plusieurs informations textuelles : le type de document, le type de maintenance, le sujet, le titre et l’URL.
* Le champ metadata, contenant les mêmes éléments de manière structurée avec le guideid, la category et summary en plus.

J’ai ensuite utilisé le RecursiveCharacterTextSplitter[[16]](#footnote-16) avec les paramètres suivants : chunk\_size = 1000 et chunk\_overlap = 200. Le chevauchement permet de garder une continuité dans le contenu, ce qui est important quand un fragment est coupé au milieu d’une explication.

Figure  : Modèle de récupération des guides iFixit basé sur embeddings

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect. (retrieve.ipynb)

Une fois les fragments générés, j’ai utilisé le modèle all-mpnet-base-v2 de Hugging Face pour produire des embeddings. Ce modèle a été choisi pour sa précision sur les textes multilingues et son intégration simple avec LangChain. Les embeddings ont ensuite été stockés dans une base FAISS (Facebook AI Similarity Search), qui permet une recherche rapide par similarité vectorielle.

Le choix de FAISS repose sur un benchmark technique évoqué dans l’article « Comparing RAG Part 2: Vector Stores; FAISS vs Chroma » (Stepkurniawan 2024), où FAISS est plus rapide et plus performant que Chroma dans des contextes similaires.

### Indexation des posts Reddit Tech Support

Les données Reddit proviennent du subreddit r/techsupport. Contrairement aux guides iFixit, les posts Reddit présentent un contenu plus informel et non structuré, mais extrêmement utile pour refléter les formulations réelles des utilisateurs.

À l’aide de la bibliothèque PRAW, chaque post a été récupéré avec :

* Le titre (souvent une synthèse du problème),
* Le corps du message (détails, erreurs, démarches déjà effectuées),
* Les commentaires (conseils, solutions proposées),
* L’URL du post.

Ces informations ont été regroupées et formatées dans un objet Document, avec :

* page\_content : intégration du titre et du post complet.
* metadata : contenant les mêmes éléments de manière structurée avec les commentaires et l’URL en plus.

Les documents Reddit ont été stockés dans un second index FAISS.

Figure  : Modèle de récupération des posts Reddit basé sur embeddings

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect. (retrieve.ipynb)

## Influence de la langue sur les résultats

Une fois l’indexation en place, j’ai réalisé une série de tests pour évaluer la sensibilité du système à la langue utilisée dans la requête.

En gardant le même modèle d’embedding (all-mpnet-base-v2), j’ai formulé une requête en français puis en anglais pour comparer les résultats. La question choisie était :

* FR : « Comment réparer un téléviseur Samsung ? »
* EN : « How to fix a Samsung television? »

En français, les documents retournés étaient peu pertinents, sans réel rapport avec le sujet. En revanche, la version anglaise a permis de retrouver des guides en lien direct avec des pannes d’écran ou des modèles Samsung.

Figure  : Test du modèle de récupération des guides iFixit basé sur embeddings en français

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.(retrieve.ipynb)

Figure  : Test du modèle de récupération des guides iFixit basé sur embeddings en anglais

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect. (retrieve.ipynb)

Ces tests montrent que malgré la nature multilingue du modèle all-mpnet-base-v2, la langue d’interrogation influence fortement la qualité de la récupération, surtout quand le corpus est majoritairement anglophone.

## Comparaison avec un modèle statistique

Pour aller plus loin, j’ai comparé l’approche par embeddings à une approche plus traditionnelle : un retriever basé sur les statistiques, plus précisément BM25.

J’ai utilisé la classe BM25Retriever[[17]](#footnote-17) de LangChain, qui repose sur l’algorithme TF-IDF pour classer les documents selon la fréquence des mots.

J’ai utilisé la requête suivante (en anglais cette fois, pour garder la cohérence avec les tests précédents) :

« How to repair a Samsung TV? »

Les résultats montrent une différence nette :

* BM25 retourne des documents contenant les termes « How to repair » ou encore « How to », mais pas des documents pertinents au sens global.
* FAISS avec embeddings renvoie des guides réellement liés à la réparation d’écrans, même si tous les mots ne sont pas présents dans le texte exact.

Figure  : Modèle de récupération des guides iFixit basé sur statistiques

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect. (retrieve.ipynb)

Ces tests m’ont permis de comprendre que les embeddings sont plus adaptés à des requêtes formulées en langage naturel, alors que BM25 reste efficace uniquement si les mots-clés sont très bien choisis.

### Temps d’indexation

L’un des inconvénients majeurs de l’approche par embeddings est son temps de traitement. Sur ma machine, l’indexation via FAISS avec all-mpnet-base-v2 a duré environ 20 minutes, contre 1,5 seconde pour l’approche BM25.

Ce constat souligne que même si les performances de récupération sont meilleures avec les embeddings, elles ont un coût non négligeable en termes de ressources et de temps de calcul.

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.Une image contenant texte, Police, capture d’écran, Graphique

Le contenu généré par l’IA peut être incorrect.(retrieve.ipynb)

Figure  : Retriever embeddings

Figure  : Retriever « BM25 »

(retrieve.ipynb)

## Comprendre la différence entre vector store et retriever

Pendant le développement, j’ai découvert qu’il fallait bien différencier deux concepts souvent confondus dans la documentation LangChain :

* Le vector store (comme FAISS) est une base de données qui stocke les vecteurs, mais ne sait pas les interroger de manière intelligente.
* Le retriever[[18]](#footnote-18) est l’objet qui permet de faire le lien entre une question et les documents à retourner, à l’aide du vector store ou d’une autre méthode.

C’est le retriever qui est utilisé dans le pipeline RAG, et non directement la base FAISS.

Cette distinction m’a permis de mieux organiser mon code et de comprendre pourquoi certains composants de LangChain sont interchangeables.

## Construction du pipeline RAG

Maintenant que les données sont récupérées et indexées dans un vector store, utilisé par un retriever, il est temps de mettre en place un premier pipeline RAG dans un notebook Jupyter. L’objectif est de récupérer à la volée les documents les plus pertinents pour une question donnée, puis de les injecter dans le prompt d’un LLM local pour générer une réponse contextualisée.

### Étape 1 - Indexation unifiée et création du retriever

Dans un premier temps, les guides iFixit et les posts Reddit avaient été indexés dans deux vector stores distincts. J’ai ensuite refactorisé cela avec une fonction unique index\_data\_embeddings, qui :

* transforme les guides et posts en objets Document LangChain,
* les découpe en fragments,
* génère leurs embeddings (all-mpnet-base-v2),
* et les stocke dans une base FAISS unique.

Le retriever ainsi obtenu utilise cette base commune, avec un paramètre k = 2 pour retourner uniquement les deux documents les plus pertinents.

Figure  : Modèle de récupération des posts et guides dans le pipeline RAG

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

### Étape 2 - Définition du LLM local via Ollama

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.Pour la génération, j’ai choisi Ollama, qui permet de faire tourner des LLMs open-source localement (comme Mistral, Llama3, etc.) sans dépendre d’API externes ni payer de frais. J’ai utilisé la dépendance ollama-python pour connecter le notebook au daemon Ollama installé sur ma machine.

Figure  : Classe LangChain dérivée pour utiliser un LLM via Ollama

(naive\_rag.ipynb)

### Étape 3 - Construction du prompt (version initiale)

Le prompt est construit à l’aide de PromptTemplate et contient deux variables :

* context : les documents sélectionnés,
* question : la question utilisateur.

Figure  : Prompt du LLM dans la chaîne RAG

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect. (naive\_rag.ipynb)

Ce prompt initial n’impose pas explicitement au LLM de se limiter aux documents fournis. Il l’incite à les utiliser, mais ne lui interdit pas de s’appuyer sur d'autres connaissances.

### Étape 4 - Chaînage des composants avec create\_rag\_chain

La fonction create\_rag\_chain assemble l’ensemble des composants avec le pipe operator |.

Voici la chaîne complète :

* retriever retourne une liste de Document,
* format\_documents() crée une chaîne de texte à partir des documents récupérés,
* RunnablePassthrough() passe directement la question,
* prompt\_template construit le prompt complet,
* llm appelle le modèle Ollama,
* StrOutputParser() extrait le texte généré en sortie.

Figure  : Formatage des documents retourné par « retriever » et fonction création pipeline RAG

Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

### Étape 5 - Exécution du pipeline avec .invoke()

L’exécution se fait simplement avec :

* « rag\_chain.invoke(‘Repair iPhone 12 screen’) »

Le retriever retourne ici deux guides iFixit. La fonction format\_documents les convertit en texte lisible. Les deux documents concernent la réparation d’écran, mais un seul est réellement pertinent pour l’iPhone 12. Le LLM utilise principalement le guide « iPhone 12 Screen Refurbishment » pour proposer une réponse structurée.

Cependant, comme les étapes précises du guide ne sont pas indexées, la réponse reste vague. Le modèle ne dispose que du résumé, du titre et des métadonnées.

Figure  : Exécution pipeline RAG

Une image contenant texte, capture d’écran, document

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

## Modules spécifiques : intégration dynamique des étapes iFixit

Pour améliorer la qualité des réponses générées, j’ai exploré une solution permettant d’être plus précis après la récupération initiale des documents.

En effet, si l’attribut guideid a été intégré dans les métadonnées des guides iFixit, c’est pour permettre la récupération dynamique des étapes du guide, via l’endpoint de l’API :

* « /api/2.0/guides/{guideid} »

Cette approche est volontairement différée (post-retrieval), car ajouter les étapes dès l’indexation aurait surchargé le corpus. Chaque guide peut contenir 5 à 30 étapes, ce qui, multiplié par plusieurs centaines de documents, rendrait l’index lourd et lent à construire

### Implémentation d’une fonction spécifique

J’ai donc créé une fonction Python dédiée permettant de récupérer les étapes d’un guide à partir de son guideid. Cette fonction retourne une liste structurée d’étapes numérotées, prêtes à être injectées dans le texte :

Figure  : Fonction retournant les étapes d’un guide iFixit

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

### Modification de format\_documents()

La fonction format\_documents(), qui jusque-là concaténait uniquement le contenu brut et les métadonnées, a été modifiée. Si un document retourné par le retriever contient un guideid, cette fonction :

1. appelle l’API iFixit,
2. récupère les étapes sous forme de tableau numéroté,
3. ajoute cette chaîne complète au champ page\_content.

Figure  : Formatage des documents retourné par « retriever » avec étapes pour les guides iFixit

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

### Adaptation du prompt

Avec l’intégration des étapes dans le contexte, il devient pertinent de modifier le prompt pour indiquer explicitement au LLM de les utiliser dans sa réponse. Le nouveau prompt contient une instruction supplémentaire :

Figure  : Utilisation étapes guides iFixit dans prompt du LLM dans la chaîne RAG

(naive\_rag.ipynb)

### Résultat et analyse

J’ai relancé le pipeline RAG avec la même requête qu’auparavant :

* « Repair iPhone 12 screen »

Cette fois, le guide iFixit retourné inclut ses étapes détaillées, injectées dans le contexte. Cela permet au LLM de générer une réponse structurée, étape par étape.

Figure  : Contenu guide iFixit retourné par « retriever » avec étapes données au contexte

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect. (naive\_rag.ipynb)

Figure  : Sortie pipeline RAG avec étapes guides iFixit

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

# Expérimentation et évaluation

## Constat initial

Une fois le pipeline RAG mis en place et les documents bien injectés dans le contexte, j’ai constaté que la qualité des réponses générées dépendait fortement du modèle LLM utilisé. En effet, même avec un contexte pertinent (incluant les étapes des guides iFixit), les réponses retournées par le modèle llama3:8b manquaient souvent de clarté ou de cohérence. Il arrivait que :

* le modèle réponde en anglais, malgré une consigne explicite de répondre en français,
* il ignore des étapes pourtant présentes dans le contexte,
* ou pire, hallucine une source qui n’existe pas réellement

Une hypothèse pour expliquer ces problèmes est la langue majoritaire du contexte (souvent en anglais) qui pourrait perturber un modèle qui doit répondre en français. Une autre hypothèse concerne la taille du modèle, la version utilisée étant la 8b (8 milliards de paramètres), ce qui limite ses capacités de compréhension et de génération.

## Comparaison entre plusieurs LLM

Pour améliorer la qualité des réponses, j’ai décidé de tester différents modèles génératifs, open source et propriétaires, tout en conservant exactement le même pipeline RAG et la même requête :

* « Repair iPhone 12 screen »

### Modèle 1 - llama3.1:8b

Ce modèle a été exécuté localement via Ollama. Il s’agit d’une version allégée (8b) qui demande peu de ressources, ce qui le rend compatible avec ma machine. Cependant, les résultats sont mitigés :

* la réponse manque de précision,
* les étapes du guide ne sont pas exploitées correctement,
* une source inexacte est parfois citée,
* la génération en français est souvent approximative.

Figure  : Sortie pipeline RAG avec modèle « llama3.1:8b »

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

Figure  : Temps exécution pipeline RAG avec modèle « llama3.1:8b »

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.

(naive\_rag.ipynb)

### Modèle 2 - mistral:7b

Le modèle mistral:7b, aussi exécuté via Ollama, offre une légère amélioration :

* il utilise les documents fournis dans le contexte,
* il intègre les étapes iFixit dans sa réponse.

Cependant, il n’élimine pas les documents inappropriés, ce qui affecte la cohérence de sa réponse. Par exemple, il a pu combiner les étapes de deux guides, dont un qui ne concernait pas l’iPhone 12.

Figure  : Sortie pipeline RAG avec modèle « mistral:7b »

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

Figure  : Temps exécution pipeline RAG avec modèle « mistral:7b »

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.

(naive\_rag.ipynb)

### Modèle 3 - deepseek-r1:32b

Avec deepseek-r1:32b (32,8B de paramètres), la qualité de réponse a franchi un cap :

* la réponse est exhaustive et correctement structurée,
* le modèle sélectionne uniquement le guide pertinent,
* il exploite les étapes du guide de manière appropriée,
* il cite la bonne source (pas de contenu halluciné).

Figure  : Sortie pipeline RAG avec modèle « deepseek-r1:32b »

Une image contenant texte, capture d’écran, Police, document

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

Figure  : Réflexion modèle « deepseek-r1:32b » pipeline RAG

Une image contenant texte, capture d’écran, Police, document

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

Figure  : Temps exécution pipeline RAG avec modèle « deepseek-r1:32b »

Une image contenant texte, Police, capture d’écran, nombre

Le contenu généré par l’IA peut être incorrect.

(naive\_rag.ipynb)

### Modèle 4 - gpt-4.1 (API OpenAI)

Enfin, j’ai testé GPT-4.1 via l’API OpenAI. Contrairement aux autres modèles, il est hébergé sur une machine distante, ce qui permet :

* une exécution rapide,
* un chargement contextuel important,
* des réponses très précises et bien formulées.

La réponse générée est équivalente, voire légèrement supérieure, à celle de deepseek-r1:32b. Elle est bien rédigée en français, exploite les bonnes sources, utilise les bonnes étapes.

Figure  : Sortie pipeline RAG avec modèle « gpt-4.1 »

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

Figure  : Temps exécution pipeline RAG avec modèle « gpt-4.1 »

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.

(naive\_rag.ipynb)

Figure  : Modèle « gpt-4.1 » instancié via « ChatOpenAI »

Une image contenant texte, capture d’écran, Police, ligne

Le contenu généré par l’IA peut être incorrect.(naive\_rag.ipynb)

## Tableau comparatif

Tableau - Comparatif des modèles génératifs testés

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Modèle** | **Taille (paramètres)** | **Temps (s)** | **Pertinence** | **Langue** | **Coût** | **Remarques clés** |
| llama3.1:8b | 8B (4.9 Go) | 25.1 | Faible | Faible | Gratuit | Réponse peu exploitable |
| mistral:7b | 7B (4.1 Go) | 27.7 | Moyenne | Moyenne | Gratuit | Mauvais tri des sources |
| deepseek-r1:32b | 32.8B (20 Go) | 716 (11 min) | Élevée | Bonne | Gratuit | Réponse pertinente mais lente |
| gpt-4.1 | ? (API) | 19.5 | Très élevée | Excellente | Payant | Très fluide, pas de charge machine |

## Analyse générale

* Langue : Les modèles avec peu de paramètres (ex. mistral, llama3.1:8b) peinent à générer du contenu fluide en français, surtout quand le contexte est en anglais.
* Pertinence : Seuls deepseek-r1:32b et gpt-4.1 ont su sélectionner le bon guide, sans confusion avec d’autres documents.
* Hallucination : Les modèles les plus petits hallucinent facilement des sources. Ce phénomène a disparu avec les modèles plus lourds.
* Coût/Ressources : Les modèles locaux demandent beaucoup de RAM et de VRAM. GPT-4.1, en API, contourne ces limites mais implique un coût d’usage.

# Amélioration du pipeline RAG

À la suite des résultats observés lors de la phase d’expérimentation, plusieurs limites du pipeline RAG initial ont été identifiées. Celles-ci ont motivé une série d’ajustements qui transforment le pipeline initial, encore naïf, en un pipeline de type RAG avancé, tel que défini par Gao et al. (2024, p. 4). Ces améliorations concernent à la fois la récupération des documents, la construction du prompt, et la gestion du contexte.

## Post-récupération : un premier pas vers le RAG avancé

Comme présenté précédemment (section 5.6), l’intégration dynamique des étapes iFixit via le champ guideid constitue déjà une forme de post-récupération. Cette stratégie permettait d’alléger l’indexation tout en enrichissant le contexte seulement après avoir identifié un document pertinent.

Cette méthode rejoint la logique du RAG avancé, qui selon Gao et al. introduit des stratégies spécifiques après récupération pour améliorer la qualité de la génération : reclassement, compression ou enrichissement ciblé du contexte.

## Refonte du prompt pour limiter les hallucinations

Le prompt initial ne contraignait pas explicitement le modèle à s’en tenir au contexte fourni. Cela laissait la porte ouverte à des réponses contenant des informations inventées.

Dans le pipeline révisé, j’ai introduit un nouveau prompt formulé de manière explicite :

Figure  : Prompt du LLM dans la chaîne RAG avancé

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect.(advanced\_rag.ipynb)

Cette formulation permet de lier directement la qualité de la réponse à la qualité des documents récupérés, et donc d’évaluer plus finement les performances du système.

## Changement de modèle d’embeddings

Un autre ajustement a concerné le modèle d’embeddings. Initialement basé sur all-mpnet-base-v2, le pipeline utilise désormais all-MiniLM-L6-v2[[19]](#footnote-19). Ce modèle, plus léger, offre des résultats très proches en termes de pertinence, tout en réduisant considérablement les temps d’indexation.

Cela s’aligne avec la logique du RAG avancé, où l’optimisation de la structure d’index est un des leviers identifiés par Gao et al. dans la phase de pre-retrieval.

## Implémentation complète de la Query Expansion

### Objectif

Une des faiblesses du pipeline initial était sa dépendance à la formulation exacte de la requête utilisateur. Gao et al. soulignent que l’efficacité d’un RAG naïf chute rapidement quand la question est mal formulée ou imprécise. Pour y remédier, j’ai intégré un module de Query Expansion, qui constitue un pilier du RAG avancé.

« Expanding a single query into multiple queries enriches the content of the query, providing further context to address any lack of specific nuances… » (Gao et al. 2024, p. 8)

### Étape 1 - Génération des variantes de la question

J’ai implémenté une chaîne generate\_queries, qui utilise un LLM et un prompt spécifique pour générer cinq reformulations en anglais de la question d’origine. Cela permet d’augmenter les chances de trouver des documents pertinents, en particulier lorsque ceux-ci sont majoritairement rédigés en anglais.

Figure  : Prompt LLM « Query Expansion »

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.(advanced\_rag.ipynb)

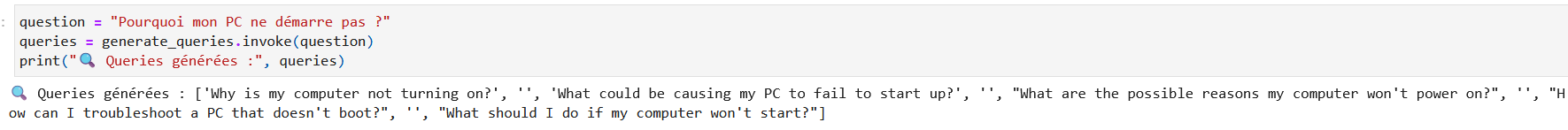
Figure  : Chaîne de génération des questions variantes

(advanced\_rag.ipynb)

À l'exécution, la chaîne retourne un tableau de chaînes, incluant la version traduite de la question originale et plusieurs reformulations :

* « Why is my computer not turning on? »
* « What are the possible reasons my computer won't power on? »
* …

Figure : Générations des questions variantes

(advanced\_rag.ipynb)

### Étape 2 - Récupération multi-requêtes

Chaque requête générée est ensuite envoyée individuellement au retriever, via une nouvelle chaîne appelée retrieval\_chain.

Avec .map(), on applique le retriever à chaque question, ce qui génère une liste de listes de documents. Ces documents sont ensuite fusionnés et dédupliqués avec la fonction get\_unique\_union.

Figure  : Chaîne de récupération

(advanced\_rag.ipynb)

Figure  : Fonction retournant une liste de documents uniques à partir de listes imbriquées

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.(advanced\_rag.ipynb)

Cette étape garantit que seuls les documents uniques, réellement pertinents, sont conservés, même si plusieurs questions convergent vers les mêmes sources.

### Étape 3 - Filtrage par score de similarité

Pour limiter la pollution du contexte, j’ai ajouté une contrainte sur le score de similarité minimal dans le retriever :

Figure  : Configuration du score de similarité minimal pour filtrer les documents retournés par le « retriever »

Une image contenant texte, Police, capture d’écran, ligne

Le contenu généré par l’IA peut être incorrect.(advanced\_rag.ipynb)

Cela permet de filtrer les résultats retournés pour chaque requête et de garantir un certain niveau de qualité dans les documents injectés.

### Résultat final

La combinaison de :

* la traduction et reformulation de la question,
* la multi-récupération + déduplication,
* le filtrage par similarité,
* et le prompt contraint,

forme un pipeline complet et avancé, capable de mieux comprendre la requête, d’enrichir son contexte de manière ciblée, et de produire des réponses plus fiables et structurées.

Figure  : Exécution pipeline RAG avancé

Une image contenant texte, Police, capture d’écran, ligne

Le contenu généré par l’IA peut être incorrect.

(advanced\_rag.ipynb)

Figure  : Sortie pipeline RAG avancé (1)

Une image contenant texte, capture d’écran, document, Police

Le contenu généré par l’IA peut être incorrect.(advanced\_rag.ipynb)

Figure  : Sortie pipeline RAG avancé (2)

Une image contenant texte, capture d’écran, Police, document

Le contenu généré par l’IA peut être incorrect.(advanced\_rag.ipynb)

## Synthèse

Avec ces optimisations, mon pipeline dépasse les limites d’un RAG naïf. Il intègre désormais plusieurs stratégies pré- et post-récupération, tout en s’appuyant sur un modèle de génération (GPT-4.1) capable de tirer parti de ce contexte enrichi.

Il ne s’agit pas d’un pipeline modulaire, où chaque étape peut être activée indépendamment, mais bien d’un RAG avancé unifié, structuré autour d’une logique d’amélioration continue de la qualité de récupération et de génération.

# Discussion

À ce stade du projet, il est possible de revenir point par point sur les sous-questions posées dans l’introduction, à la lumière des résultats concrets obtenus et des améliorations successives apportées au pipeline. Cette discussion permet d’évaluer dans quelle mesure l’hypothèse centrale du mémoire est confirmée, à savoir :

« Il est possible de produire des réponses pertinentes à des questions techniques, sans entraîner un modèle, en construisant un pipeline RAG avancé qui exploite correctement des documents externes. »

## Récupération - Le retriever suffit-il à isoler les bons documents ?

Dans la majorité des cas, le système de récupération basé sur FAISS + embeddings permet effectivement de retourner des documents pertinents. Cela est particulièrement vrai lorsque :

* le modèle d’embedding est bien choisi (all-mpnet-base-v2 ou MiniLM-L6-v2),
* la question est bien formulée,
* et la méthode de query expansion est utilisée pour élargir la recherche.

Cependant, certains tests ont révélé que la récupération brute n’était pas toujours suffisante : deux documents peuvent contenir des termes proches, mais un seul est réellement pertinent pour la réponse. Cela a conduit à l’ajout d’un filtrage par score de similarité, puis à la logique de post-récupération (guideid), ce qui marque un passage vers un RAG avancé plus sélectif et enrichi.

## Modèle génératif - Tous les LLMs peuvent-ils exploiter le contexte ?

Les tests ont montré que tous les modèles ne sont pas capables d’utiliser efficacement les documents fournis.

Les modèles open source de taille modérée (comme mistral:7b ou llama3.1:8b) produisent souvent des réponses vagues, incomplètes, voire erronées, même avec un bon contexte injecté.

En revanche, les modèles plus puissants comme deepseek-r1:32b et surtout GPT-4.1 exploitent correctement les documents fournis, les synthétisent, et construisent une réponse cohérente et crédible.

Cela montre que la qualité du modèle génératif reste un facteur critique, et qu’un bon retriever ne suffit pas à garantir la qualité de la réponse finale.

## Formulation de la requête - Impact sur la récupération et la génération

L’impact de la formulation de la requête s’est avéré considérable. Une question floue ou mal formulée aboutit souvent à la récupération de documents hors sujet.

C’est ce constat qui a motivé l’intégration d’un module de query expansion, permettant au système de générer plusieurs reformulations automatiques en anglais. Cela permet d’augmenter la couverture sémantique et d’améliorer la qualité de la recherche documentaire, sans dépendre de la formulation exacte de l’utilisateur.

Grâce à cette stratégie, même des questions mal formulées peuvent aboutir à des résultats pertinents.

## Structuration du prompt - Peut-on vraiment éviter les hallucinations ?

Le prompt initial n’imposait pas explicitement au modèle de s’appuyer uniquement sur les documents injectés. Cela entraînait, notamment avec les petits modèles, des hallucinations : sources fictives, étapes inventées, ou conseils génériques.

L’ajout d’un prompt contraint, formulé de manière stricte, a permis de réduire considérablement ces hallucinations, en particulier avec GPT-4.1.

Le prompt reformulé précise désormais que le modèle ne doit pas utiliser de connaissances extérieures et doit répondre uniquement en se basant sur les documents fournis.

Ainsi, un prompt bien structuré est une condition nécessaire, mais pas toujours suffisante, selon le modèle utilisé.

## Performance globale - Un bon compromis sans fine-tuning ?

L’objectif final était de construire un système performant sans fine-tuning, capable de fonctionner localement ou à distance, selon les besoins.

Les expérimentations ont montré que cela est possible, mais qu’il faut trouver un équilibre :

* En local, les modèles open source fonctionnent mais leurs performances restent limitées.
* Des modèles plus puissants comme deepseek-r1:32b offrent d’excellents résultats, mais au prix de temps d’exécution très longs.
* Le meilleur compromis observé reste GPT-4.1 via API, qui offre une génération rapide, précise et fiable, même avec un contexte complexe.

L’approche RAG avancée permet donc de contourner le fine-tuning, tout en maintenant un haut niveau de qualité, à condition de combiner correctement :

* un bon retriever,
* un enrichissement de la requête,
* un prompt bien conçu,
* et un modèle génératif adapté.

# Présentation du prototype final

À l’issue des différentes étapes du projet – collecte de données, expérimentation des modèles, amélioration du pipeline – un prototype complet a été développé. Il intègre l’ensemble des modules RAG construits précédemment, dans une interface fonctionnelle, accessible depuis un navigateur web.

Ce prototype est disponible sur GitHub, tout comme les fichiers Jupyter utilisés lors des tests présentés dans les sections 5, 6 et 7. Une documentation technique dédiée accompagne l’ensemble du projet. Il est accessible ici : <https://github.com/CoCoRoRooo/TB>

## Vue d’ensemble

Le système est un chatbot technique fonctionnant selon l’architecture RAG (Retrieval-Augmented Generation).

Il permet à un utilisateur de poser une question en langage naturel (ex. : « Pourquoi mon ordinateur ne démarre plus ? ») et de recevoir une réponse construite à partir :

* de guides techniques (issus du site iFixit),
* de discussions communautaires (issues de Reddit),
* et d’un modèle de langage GPT-4.1.
  + Au cours des essais menés avec le modèle GPT-4.1 (via l’API d’OpenAI), j’ai relevé une contrainte susceptible de peser sur l’avenir du prototype : le modèle impose un plafond de 30 000 tokens par minute. Ce seuil reste acceptable pour des requêtes brèves, mais devient restrictif dès que l’on :
    - enchaîne plusieurs appels successifs incorporant un contexte riche ;
    - injecte un prompt volumineux (via le nombre potentiellement important de documents récupérés après l’enrichissement issu de la Query Expansion).

Afin de prévenir d’éventuels goulots d’étranglement – il est possible de migrer vers GPT-4.1-mini. Selon la documentation d’OpenAI, cette déclinaison élève le plafond à 200 000 tokens par minute tout en conservant une compatibilité quasi totale avec l’interface actuelle. Sur le plan technique, la transition se limite à :

1. remplacer l’identifiant du modèle dans l’appel à ChatOpenAI ;
2. ajuster, au besoin, les paramètres de suivi de quota.

Une telle évolution permettrait de maintenir la fluidité du service lorsque le volume textuel traité excède les capacités initiales, sans remettre en cause l’architecture du pipeline ni la qualité attendue des réponses.

## Architecture générale

L’architecture logicielle du système repose sur plusieurs modules interconnectés, chacun ayant un rôle bien défini :

* app.py : Point d’entrée principal (serveur Flask[[20]](#footnote-20)), qui expose une API REST et rend l’interface HTML.
* rag\_chain.py : Gère les chaînes LangChain, notamment la génération de requêtes, la récupération, et la génération finale de réponse.
* retriever.py : Gère le chargement des données, leur transformation en vecteurs (embeddings), et leur indexation dans FAISS.
* api\_client.py : Appelle l’API iFixit pour enrichir les guides avec leurs étapes détaillées.
* utils.py : Fournit des fonctions utiles, comme format\_documents ou get\_unique\_union, utilisées dans le pipeline RAG.
* templates/index.html + static/ : Ensemble des fichiers frontend, qui construisent l’interface utilisateur moderne.

Figure  : Schéma de l’architecture du prototype

Une image contenant texte, capture d’écran, Police, Page web

Le contenu généré par l’IA peut être incorrect.(documentation\_technique, p.3)

## Fonctionnement du pipeline intégré

Le cœur du système repose sur le pipeline RAG avancé, présenté en détail dans les chapitres précédents, et ici intégré dans un serveur Flask.

Le flux de traitement suit ces étapes :

1. L’utilisateur pose une question dans l’interface ou via l’API.
2. Le système génère cinq reformulations alternatives (query expansion).
3. Chaque requête est utilisée pour interroger la base vectorielle (FAISS).
4. Les documents sont fusionnés, dédupliqués et enrichis si nécessaire (via guideid).
5. Le modèle GPT-4.1 génère une réponse structurée et justifiée.
6. Le frontend affiche :
   * la réponse,
   * les documents utilisés,
   * les requêtes générées,
   * et le temps de traitement.

## Interface utilisateur

Le prototype intègre une interface web moderne, accessible localement via localhost:5000[[21]](#footnote-21). Elle a été pensée pour être à la fois :

* claire pour l’utilisateur final (affichage des réponses),
* transparente pour les développeurs (affichage des documents, requêtes et temps d’exécution).

Fonctionnalités principales :

* Chat en direct avec effet de frappe simulé.
* Panneaux pliables pour afficher les requêtes et les documents utilisés.
* Temps de traitement affiché après chaque interaction.
* Rendu Markdown des réponses (avec titres, listes, blocs).
* Mode responsive : l’interface s’adapte aux mobiles.

Figure  : Capture d’écran de l’interface du prototype en fonctionnement

Une image contenant texte, capture d’écran, logiciel, Page web

Le contenu généré par l’IA peut être incorrect.(Prototype, index.html)

## Déploiement et configuration

L’installation du prototype est décrite dans son readme, une carte graphique compatible CUDA est recommandée pour accélérer les calculs d’embeddings.

## Documentation associée

Le dépôt GitHub contient une documentation technique complète (fichier PDF) expliquant en détail :

* la structure des fichiers et des modules,
* les scripts de collecte des données (get\_posts.py, guides.py),
* les prompts utilisés pour le LLM,
* les modules spécifiques du système,
* les possibilités d’extension (ajout de sources, changement de modèle, personnalisation de l’UI).

Cette documentation est conçue pour permettre à un développeur tiers de comprendre, reproduire ou améliorer le prototype, sans avoir à relire tout le mémoire.

# Conclusion

## Bilan et apports techniques

Ce projet m’a permis d’explorer en profondeur la conception d’un système intelligent capable de générer des diagnostics techniques à partir d’une simple question utilisateur.

En partant d’une problématique concrète, comprendre et expliquer des pannes électroniques de manière automatisée, j’ai progressivement construit un pipeline RAG avancé, sans recourir au fine-tuning d’un modèle.

Parmi les principaux apports techniques, on peut retenir :

* la mise en place d’un pipeline RAG complet avec LangChain,
* l’intégration d’un retriever vectoriel performant (FAISS),
* l’enrichissement dynamique via guideid (post-récupération),
* l’ajout d’une query expansion automatique avec génération multi-requêtes,
* et l’utilisation d’un modèle de génération de haut niveau (GPT-4.1).

Le prototype final, disponible sur GitHub, fonctionne localement avec une interface claire et un backend entièrement documenté. Il offre des réponses pertinentes, contextuelles, et explicables.

## Retours personnels

Ce projet m’a permis de mettre en pratique plusieurs compétences, en traitement du langage naturel (NLP), en manipulation de modèles de langage, en architecture logicielle, et en gestion de projet.

J’ai aussi appris à jongler entre différentes contraintes :

* techniques (temps d’exécution, ressources machines),
* linguistiques (contexte en anglais, réponse attendue en français),
* et conceptuelles (comprendre le fonctionnement réel d’un pipeline RAG).

L’approche RAG m’a particulièrement intéressé car elle offre une alternative élégante au fine-tuning, en exploitant au maximum les documents disponibles. Cela m’a également permis de mieux comprendre les limites réelles des modèles de langage, et les leviers pour en améliorer l’usage dans des contextes spécialisés.

## Ouvertures possibles

Le système développé peut être étendu ou amélioré de plusieurs manières :

### Ajout de sources supplémentaires

Le pipeline pourrait intégrer d’autres types de documents, comme des PDF de manuels, des bases internes à une entreprise, ou des forums spécialisés.

### Déploiement cloud + CI/CD

Actuellement déployé localement, le prototype pourrait être intégré à une architecture cloud avec CI/CD (par exemple via Docker).

### Multilingue

Rendre le système plus universel, avec des réponses dans différentes langues selon l’utilisateur.

### Interaction avec l’utilisateur

Enfin, il serait possible d’ajouter une logique de dialogue plus avancée (par exemple une mémoire de session), pour que le système puisse poser des questions de clarification ou ajuster sa réponse en fonction des échanges précédents.

Bibliographie

AÏCHA, 2024. Comment développer un chatbot avec des LLM : le guide. [en ligne]. 3 mars 2024. Disponible à l’adresse : https://www.innovatiana.com/post/chatbot-with-llm [consulté le 9 mars 2025].

BARKOVA, Anastasia, 2023. Chatbots modernes : comment les LLM transforment les méthodes de développement et l’expérience utilisateur. *Blog Devoteam Rebirth* [en ligne]. 7 décembre 2023. Disponible à l’adresse : https://rebirth.devoteam.com/2023/12/07/chatbots-llm/ [consulté le 9 mars 2025].

BOMMASANI, Rishi et al., 2022. *On the Opportunities and Risks of Foundation Models*. arXiv:2108.07258. arXiv. arXiv:2108.07258. DOI 10.48550/arXiv.2108.07258. arXiv:2108.07258 [cs]

BROWN, Tom B. et al., 2020. *Language Models are Few-Shot Learners*. arXiv:2005.14165. arXiv. arXiv:2005.14165. DOI 10.48550/arXiv.2005.14165. arXiv:2005.14165 [cs]

DAM, Sumit Kumar et al., 2024. *A Complete Survey on LLM-based AI Chatbots*. arXiv:2406.16937. arXiv. arXiv:2406.16937. DOI 10.48550/arXiv.2406.16937. arXiv:2406.16937 [cs]

GAO, Yunfan et al., 2024. *Retrieval-Augmented Generation for Large Language Models: A Survey*. arXiv:2312.10997. arXiv. arXiv:2312.10997. DOI 10.48550/arXiv.2312.10997. arXiv:2312.10997 [cs]

STEPKURNIAWAN, 2024. Comparing RAG Part 2: Vector Stores; FAISS vs Chroma. *Medium* [en ligne]. 1 janvier 2024. Disponible à l’adresse : https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb [consulté le 17 mars 2025].

VASWANI, Ashish et al., 2023. *Attention Is All You Need*. arXiv:1706.03762. arXiv. arXiv:1706.03762. DOI 10.48550/arXiv.1706.03762. arXiv:1706.03762 [cs]

ZEMČÍK, Tomáš, 2019. A Brief History of Chatbots. *DEStech Transactions on Computer Science and Engineering*. DOI 10.12783/dtcse/aicae2019/31439.

ZHAO, Wayne Xin et al., 2025. *A Survey of Large Language Models*. arXiv:2303.18223. arXiv. arXiv:2303.18223. DOI 10.48550/arXiv.2303.18223. arXiv:2303.18223 [cs]

Annexe 1 : Notebooks

Les notebooks utilisés tout au long du mémoire son disponible au format PDF et Python dans le dossier « Notebooks » lui-même dans le dossier « Annexes ».

Annexe 2 : Prototype

Le prototype mis en place est disponible dans le dossier « Prototype » lui-même dans le dossier « Annexes ».

Annexe 3 : Documentation technique du prototype

La documentation technique du prototype, « documentation\_technique », est accessible dans le dossier « Annexes ».

1. https://www.langchain.com/ [↑](#footnote-ref-1)
2. https://api.python.langchain.com/en/latest/prompts/langchain\_core.prompts.prompt.PromptTemplate.html [↑](#footnote-ref-2)
3. https://api.python.langchain.com/en/latest/runnables/langchain\_core.runnables.passthrough.RunnablePassthrough.html [↑](#footnote-ref-3)
4. https://python.langchain.com/api\_reference/core/output\_parsers/langchain\_core.output\_parsers.string.StrOutputParser.html [↑](#footnote-ref-4)
5. https://python.langchain.com/api\_reference/community/vectorstores/langchain\_community.vectorstores.faiss.FAISS.html [↑](#footnote-ref-5)
6. https://ollama.com/ [↑](#footnote-ref-6)
7. https://github.com/ollama/ollama-python [↑](#footnote-ref-7)
8. https://platform.openai.com/docs/overview [↑](#footnote-ref-8)
9. https://python.langchain.com/docs/integrations/chat/openai/ [↑](#footnote-ref-9)
10. https://python.langchain.com/api\_reference/huggingface/embeddings/langchain\_huggingface.embeddings.huggingface.HuggingFaceEmbeddings.html [↑](#footnote-ref-10)
11. https://praw.readthedocs.io/en/latest/ [↑](#footnote-ref-11)
12. https://www.ifixit.com/ [↑](#footnote-ref-12)
13. https://www.ifixit.com/api/2.0/doc [↑](#footnote-ref-13)
14. https://www.reddit.com/r/techsupport/ [↑](#footnote-ref-14)
15. https://api.python.langchain.com/en/v0.0.339/schema/langchain.schema.document.Document.html [↑](#footnote-ref-15)
16. https://python.langchain.com/api\_reference/text\_splitters/character/langchain\_text\_splitters.character.RecursiveCharacterTextSplitter.html [↑](#footnote-ref-16)
17. https://python.langchain.com/docs/integrations/retrievers/bm25/ [↑](#footnote-ref-17)
18. https://python.langchain.com/docs/how\_to/vectorstore\_retriever/ [↑](#footnote-ref-18)
19. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 [↑](#footnote-ref-19)
20. https://flask.palletsprojects.com/en/stable/ [↑](#footnote-ref-20)
21. Réponse générée par GPT-4, OpenAI. Prompt utilisé : « Génère une interface HTML complète et responsive pour un chatbot connecté à un backend Flask déjà développé. Ce backend expose une route /api/chat acceptant une question et retournant une réponse, une liste de requêtes générées, les documents utilisés, ainsi qu’un temps de traitement. L’interface doit inclure : une sidebar à gauche affichant dynamiquement ces trois éléments, et un panneau principal avec une zone de messages, un champ de saisie en bas, et un message de bienvenue. Utilise Tailwind CSS pour le design responsive, et crée un fichier style.css pour les animations (pulsation, chargement). Le fichier main.js doit gérer l'envoi de requêtes via fetch, afficher la réponse du bot lettre par lettre, et insérer les requêtes/documents récupérés dynamiquement. » [↑](#footnote-ref-21)