Rapport du projet PolyChat

Développeurs (MAIN5 2024-2025) :

Samuel AVAH BELOBO-OHANA, Tristan CHENAILLE,
Amalia SEYDOU, Yannick ZHANG

Encadrant (Responsable MAIN5) :

Xavier TANNIER

Table des matières

1 Introduction

PolyChat s'inscrit dans le cadre de notre projet industriel de 3ème année du cycle ingénieur en Mathématiques Appliquées et Informatique (MAIN) à Polytech-Sorbonne. L'objectif était d'implémenter un chatbot intégrable au site web de l'école, permettant de répondre de manière conversationnelle aux questions des étudiants, futurs étudiants ou parents d'élèves, avec des informations précises et accessibles sur l'institution, ses programmes, et ses services.

Le principe de chatbot repose sur l'utilisation de grands modèles de langage (LLM). Plusieurs logiques et modèles ont été testés dans le cadre de ce projet, notamment `GPT-4o`, `GPT-4o-mini` et `Hermes-3-8B`, qui ont été intégrés dans notre maquette finale et sur lesquels le chatbot peut actuellement fonctionner. D'autres approches moins fructueuses ont également été faites avec d'autres versions de `GPT` et de `LLaMA`, mais ne sont pas intégrées dans le rendu définitif.

Nous avons choisi le langage de programmation Python pour son haut niveau d'abstraction ainsi que pour ses bibliothèques, qui représentent actuellement l'état de l'art en intelligence artificielle.

Pour structurer l'interaction entre l'utilisateur et le chatbot, nous avons opté pour la librairie `LangChain` afin de mettre en place un système de Retrieval-Augmented Generation (RAG) et d'intégrer une mémoire conversationnelle conservant le contexte des échanges précédents.

La mise en place d'une interface utilisateur pour échanger avec le bot a été faite via le framework `Streamlit`. Nous l'avons reforgée graphiquement grâce à des paramétrages en `CSS`.

Le déploiement web de notre maquette a été fait sur un clone du site de l'école, grâce à un hébergeur web et un hébergeur de serveurs privés virtuels pour faire tourner le code et les LLM.

Ce rapport présente une explication détaillée de l'implémentation du chatbot, en décrivant et documentant nos choix techniques. Il est vivement conseillé de consulter - même brièvement - le repository Github du projet de manière à en avoir une première vue globale, puisque le présent rapport se veut technique. En effet, le lecteur est supposé avoir des bases dans le domaine.

Lien vers le repository Github contenant le code

Les implications écologiques, éthiques et sociales de ce projet et de son potentiel futur sont évoquées dans le chapitre ≪*Développement durable et responsabilité sociétale*≫ des *Annexes*.

Toujours dans les *Annexes*, un glossaire est disponible pour les lecteurs étrangers aux LLM.

2 Structure logique du système

Un rapide schéma peut aider à se figurer la manière dont le système de notre maquette finale fonctionne.
La logique de déploiement peut être modifiée très aisément selon les besoins et moyens à un instant t. Celle que nous utilisons ici était (site maquette et code/LLM sur des VPS) le plus ergonomique pour le développement, car elle nous permettait de nous projeter facilement (grâce au site web maquette), et de ne pas avoir à faire tourner de LLM gourmand localement (grâce aux VPS).

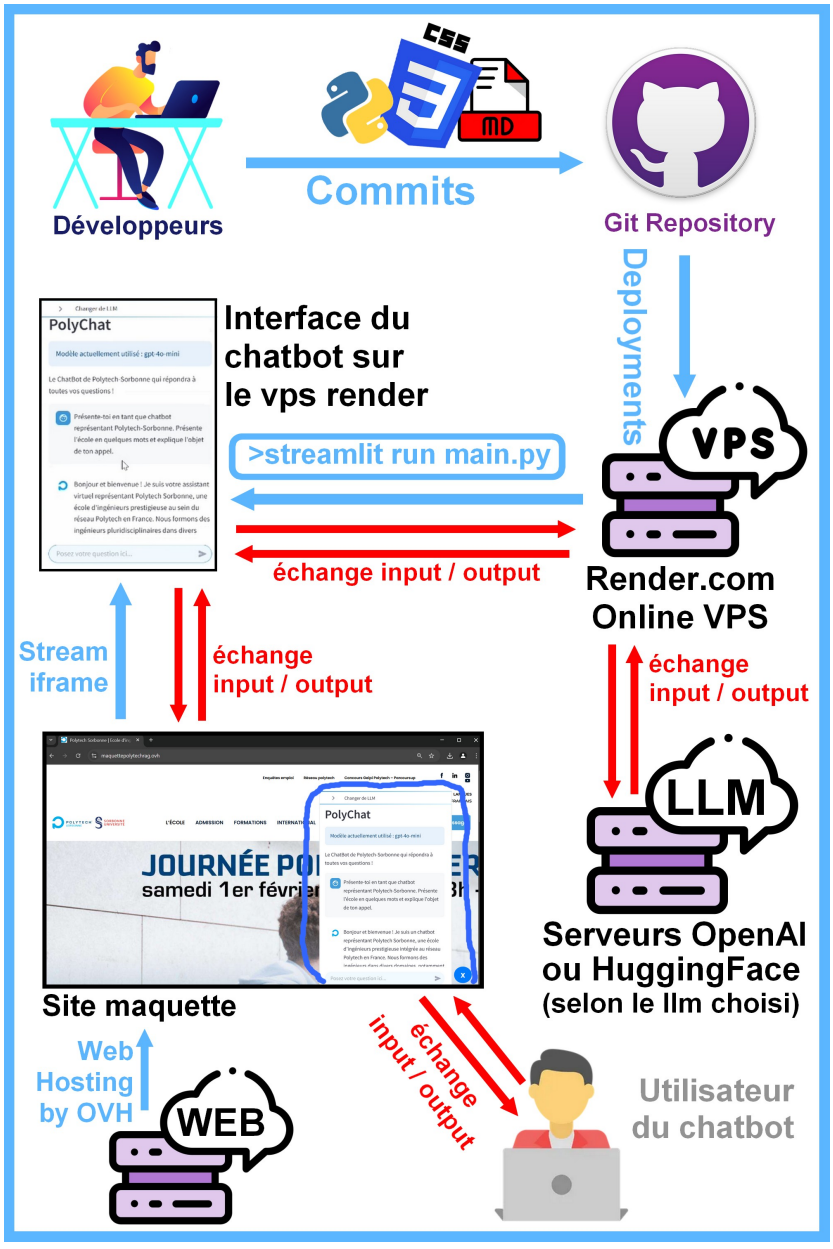


Figure 1: Workflow de la maquette

3 Structure logique de l'arborescence du code

De la même manière qu'en page précédente, un schéma de l'arborescence du code peut aider plutôt que de naviguer à l'aveuglette dans les dossiers et sous-dossiers.

Des dossiers et fichiers peu importants ont ici été omis pour préserver la clarté du résumé :

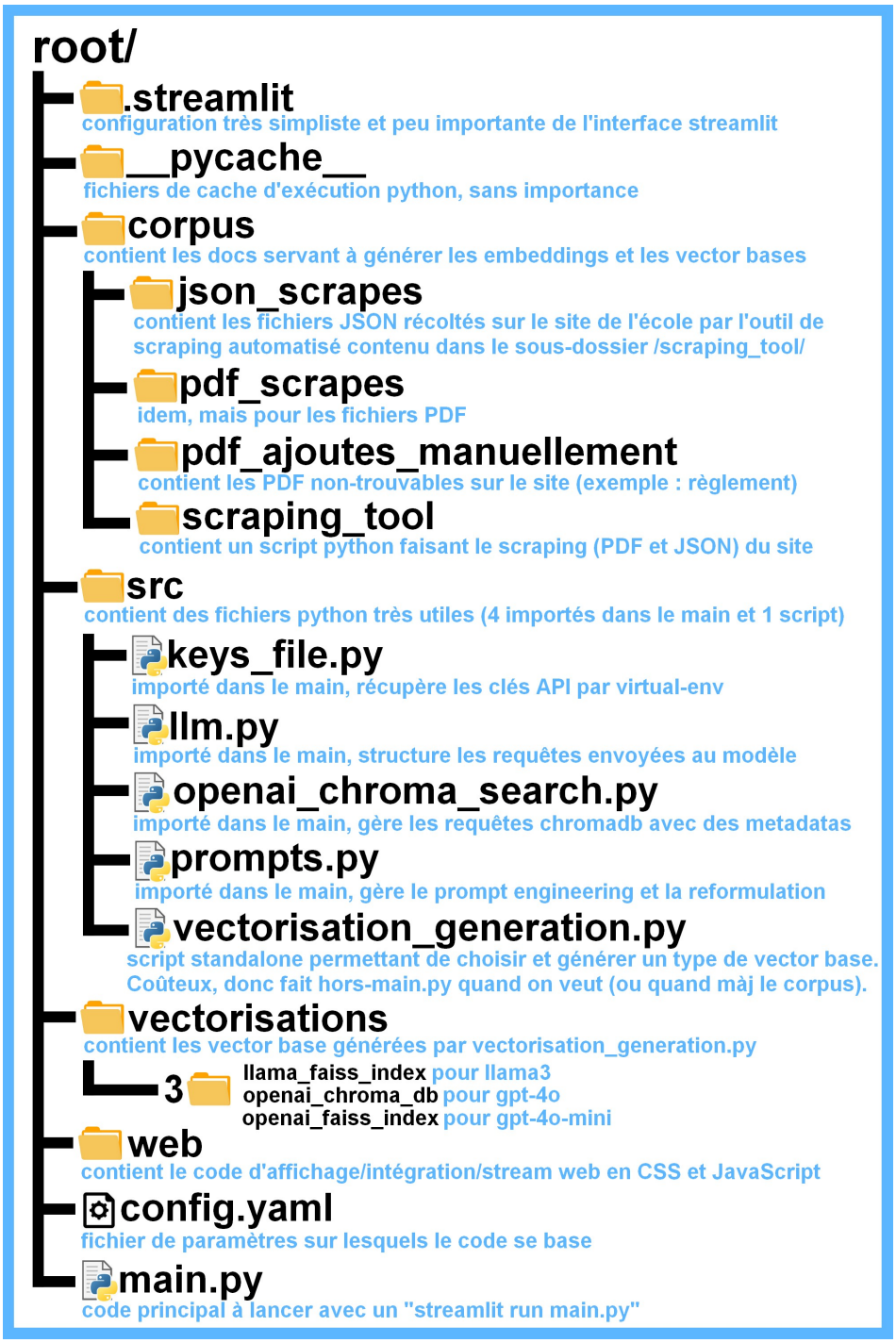


Figure 2: Arborescence du code

4 Web-scraping

`/corpus/scraping_tool/scraping_script.py` : Ce script sert à parcourir le site web de l'école pour y récolter tous les PDF. Il effectue ensuite un deuxième balayage du site, et transforme l'information textuelle de chaque page en fichier JSON qu'il télécharge. Tous les fichiers PDF sont enregistrés dans le dossier "pdf_scrapes" du corpus, et tous les fichiers JSON sont enregistrés dans le dossier "json_scrapes" du corpus. Un log est généré pour dire de quand date le dernier scraping, et combien de nouveaux fichiers il a téléchargé.

Ce script importe en réalité deux sous-modules ("`/corpus/scraping_tool/src/module_scrap_pdf.py`" et "`/corpus/scraping_tool/src/module_scrap_json.py`") définissant respectivement la logique de scraping des PDF et des JSON. Dans ces deux modules, la navigation web du scraping est faite avec la librairie `BeautifulSoup`.

Il importe également une configuration YAML (dont le chemin d'accès est "`/corpus/scraping_tool/config_scrap.yaml`") qui dicte les URL, strings et fragments d'URL à éviter. Sans ce dernier, le scrap n'est pas possible, car le site de l'école contient de très nombreuses références à des liens inexistants ou hors-sujet. Il définit aussi le texte du footer à bannir du content des JSON, et également la liste des spécialités à attribuer ou non en métadonnées avec les URLs de leurs pages respectives. Ce fichier de configuration est adaptable aux besoins de l'utilisateur à tout instant.

`Point important` : ce YAML définit actuellement la page d'actualités du site comme une URL à éviter. Cela est dû au fait que cette page compte beaucoup de sous-pages, donnant lieu à beaucoup de JSON contenant très peu d'information pertinente, et brouillant même parfois le RAG. Néanmoins, il est tout à fait possible d'inclure ces pages dans le scrap si on le souhaite en retirant la ligne correspondante dans le YAML. Il en est de même pour les sites tiers, que nous choisissons actuellement de ne pas scrapper car leurs seules pages étant référencées depuis le site de l'école sont inintéréssantes.

`Deuxième point important` : ce script implémente une logique de comparaison entre fichiers lui évitant de re-télécharger des fichiers déjà existants.

Pour les PDF, cela est fait via une requête à l'en-tête *length* du fichier distant pour comparer sa taille avec le fichier local de même nom. En cas de taille identique, le téléchargement n'est pas fait puisqu'il s'agit du même fichier. En cas de taille différente, le téléchargement est fait puisque le fichier a été mis à jour sur le serveur distant. En cas d'échec de la requête, le téléchargement est fait par précaution. Voici des exemples :

Fichier distant non-présent localement : on télécharge

```
[SCRAPING] Page : https://www.polytech.sorbonne-universite.fr/acces
[DOWNLOAD] Téléchargement de : https://www.polytech.sorbonne-universite.fr/sites/default/files/
2025-01/Plaquette-80120253-web.pdf
```

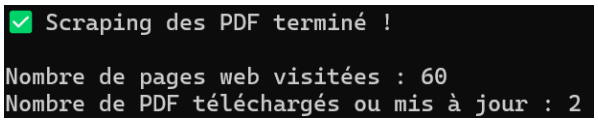
Fichier distant de même nom et de même taille : on passe à la page suivante plutôt que de télécharger

```
[SCRAPING] Page : https://www.polytech.sorbonne-universite.fr/lequipe-pedagogique
[INFO] Fichier identique déjà présent : ../pdf_scrapes\Plaquette-80120253-web_0.pdf
[SCRAPING] Page : https://www.polytech.sorbonne-universite.fr/polytech-sorbonne-et-son-reseau
```

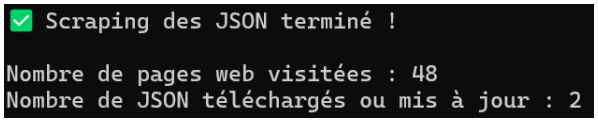
Fichier distant de même nom mais de taille différente : on télécharge

```
[INFO] Fichier existant diffère en taille (local: 2111914 vs distant: 660880). Téléchargement de la nouvelle version.
[DOWNLOAD] Téléchargement de : https://www.polytech-reseau.org/wp-content/uploads/reseauPolytech_GuideAdmissions.pdf
```

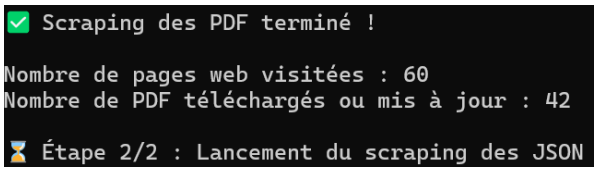
La même logique est appliquée aux fichiers JSON mais sans requête : on compare simplement l'objet JSON courant (avant son enregistrement) à celui potentiellement présent localement, en comparaison exacte. À la fin de chacune des deux étapes (PDF et JSON) du scrap, on a les informations du nombre de nouveaux téléchargements. Exemple pour la fin de la première étape (celle des PDF) :



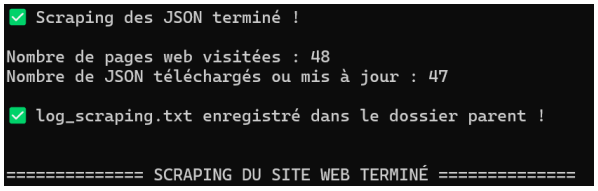
Et exemple pour la fin de la deuxième étape (celle des JSON) :



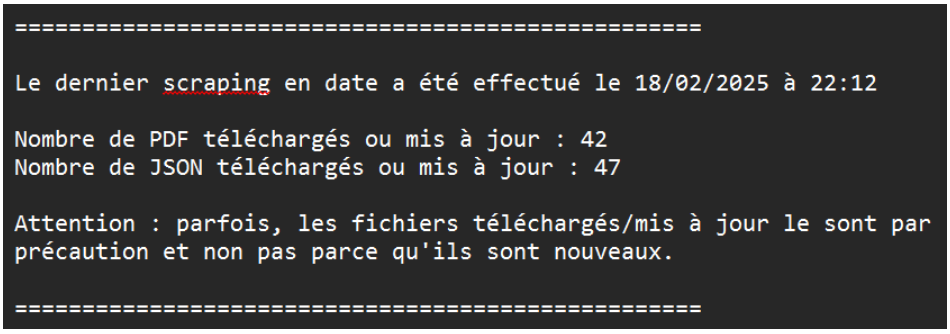
On obtient ces scores en raison de la pré-existence de documents dans le corpus. Si on vide le corpus de ses docs scrapés et que l'on exécute le script, on doit plutôt obtenir ce genre de score pour les PDF:



Et ce type de score pour les JSON :



Maintenant, regardons le log enregistré par le script :



Cette logique de scrap modulaire est très pratique car elle permet de gérer les exclusions indépendamment du code, et de vérifier ce qui se passe en temps réel, erreurs y compris.

Les logs permettent quant à eux de s'assurer que le dernier scrap en date n'est pas trop ancien, dans quel cas on pourrait vouloir en relancer un afin de récolter d'éventuelles nouvelles données présentes sur le site.

Le compteur de nouveaux documents téléchargés permet justement de se rendre compte de la présence ou de l'absence de ces éventuelles nouvelles données.

5 Génération des vector bases

`/src/vectorisation_generation.py` : Ce script contient toutes les classes et fonctions nécessaires à la création des embeddings. Ces derniers sont créés à partir du corpus de documents PDF (+JSON dans le cas de gpt-4o) et, en fonction du LLM utilisé (GPT ou Llama), ils sont générés avec la classe appropriée via OpenAI ou Hugging Face. Nous stockons ensuite ces embeddings sous la forme de vector bases (FAISS dans le cas d'embeddings de PDF, et ChromaDB dans le cas d'embeddings de PDF+JSON) afin de faciliter la recherche d'informations dans le corpus de documents.

Voir la documentation langChain sur les FAISS

Voir la documentation langChain sur les ChromaDB

Attention ! Ce fichier est un script, pas un module. Il n'est pas importé dans le main. Il ne peut être exécute qu'individuellement avec un `"python vectorisation_generation.py"`.

En effet, il n'est pas nécessaire d'exécuter ce script à chaque exécution du main, puisque cela impliquerait de régénérer les vector bases à chaque fois, ce qui peut être très coûteux. De plus, cela serait parfaitement inutile puisque le corpus ne change la plupart du temps pas d'une exécution du main à l'autre. Exécuter ce script n'est utile que lorsque vous mettez le corpus à jour, afin que le RAG pioche dans une vector base également à jour qui contienne les représentations sémantiques des informations fraîchement ajoutées au corpus.

Voici les différentes classes et fonctions présentent dans le script :

- `HuggingFaceEmbeddings` : Cette classe permet d'obtenir des embeddings de textes en interrogeant une API Hugging Face. Les différentes fonctions associées à cette classe sont:
  - `__init__(hf_api_url, hf_api_token, batch_size)` : Initialise la classe avec l'URL de l'API, le token d'authentification et la taille des batchs pour optimiser les requêtes.
  - `_query_huggingface(texts)` : Envoie un batch de textes à l'API Hugging Face et récupère les embeddings correspondants.
  - `embed_documents(texts)` : Génère les embeddings pour une liste de documents.
  - `embed_query(text)` : Génère un embedding pour une seule requête.
- `load_config()` : Cette fonction charge le fichier `config.yaml` contenant les paramètres des modèles et des chemins d'accès.
- `load_all_pdfs_from_directory(directory_path, log_files)` : Charge tous les fichiers PDF situés dans un dossier pour en extraire le texte.
- `create_embeddings(embeddings_type, vectorisation_path, directories)` : C'est avec cette fonction que l'on génère les embeddings des documents et crée un index FAISS et les stock.
  1. La première étape consiste à charger tous les documents PDF (ou PDF+JSON) situés dans les dossiers fournis en paramètre.
  2. Ensuite, on segmente les documents en morceaux de 500 caractères avec un chevauchement de 50 (`RecursiveCharacterTextSplitter`).
  3. Puis, on sélectionne le type d'embedding à utiliser (`OpenAIEmbeddings` ou `HuggingFaceEmbeddings`).
  4. On crée le FAISS (ou ChromaDB) à partir des embeddings générés et on le sauvegarde dans le dossier indiqué.
  5. Enfin, on génère un fichier log disant à quelle date à été faite la dernière génération de vector base, et quels sont les documents qui ont été inclus 'dedans".

Ce script joue un rôle clé dans le pipeline de récupération d'informations en permettant d'effectuer des recherches sémantiques optimisées. La vector base (FAISS ou ChromaDB) ainsi créée est utilisée par le chatbot pour retrouver rapidement les informations pertinentes lors des requêtes des utilisateurs.

Voici un visuel de l'interface de ce script de génération :

```
PS C:\Users\lil\Documents\GitHub\Projet-Chatbot\src> python vectorisation_generation.py

================= VECTORISATION GENERATION SCRIPT =================

Pour quel type de modèle voulez-vous générer la vectorisation ?

Tapez 1 pour GPT-4o-Mini.
Tapez 2 pour GPT-4o (avec metadata).
Tapez 3 pour LLaMA3.

Tapez votre choix : 2

✅ Choix enregistré et config.yaml chargé avec succès.

⏳ Veuillez patienter, ChromaDB en cours de génération...

✅ Vectorisation ChromaDB créée et sauvegardée (avec log) dans : ../vectorisations/openai_chroma_db

============================================================
```

Comme indiqué sur cette capture d'écran, les générations de vector base se sauvegardent dans leur dossier attitré, mais y enregistrent également un log. Voyons le log en question :

```
==================================================

Cette vectorisation a été générée le 18/02/2025 à 19:21

Fichiers JSON utilisés :

- acces_polytech_sorbonne.json
- agroalimentaire.json
- bdac_polytech_sorbonne.json
- bde_polytech_sorbonne.json
- bds_polytech_sorbonne.json
- cellule_daccompagnement_individualise_des_eleves_polytech_sorbonne.json

Fichiers PDF utilisés :

- 2022_Enquete_emploi.pdf
- 2024-12-02__Organigramme_strucure_EPU.pdf
- Enquete_emploi_2021_BD.pdf
- Enquete_emploi_2023_101023_version_diffusable.pdf
- Enquete_emploi_2024_version_diffusable_v4_0.pdf
- Enquete_emploi__2022_-_AGRAL_0.pdf
- Enquete_emploi__2022_-_GM.pdf
- Maquette_ROB5.pdf
- Maquette_ST_2022-23_officielle_modif.pdf
- Plaquette-80120253-web.pdf
- plaquette_A4_0.pdf
- Plaquette_TAP_V4.pdf
- Poster_du_projet_-capgemini.pdf
- Poster_du_projet_-Lucky-Cart.pdf
- ReseauPolytech_CharteDDRS_Janv2021.pdf
- reseauPolytech_GuideAdmissions.pdf
- admissions_etudiants_etrangers.pdf
- charte_bons_comportements.pdf
- programme_detaille_filiere_MAIN.pdf
- reglement_des_etudes.pdf
- reglement_interieur.pdf

==================================================
```

Ces logs sont très utiles puisqu'ils permettent de savoir en un coup d'oeil à quelles données le RAG a accès au travers d'une vector base donnée, et s'il faut donc lancer le script pour la régénérer ou non. Par exemple, dans ce cas précis, on remarque que le RAG actuel tourne avec une vectorbase ayant été construite sans la maquette de MAIN5, ce qui est inacceptable puisqu'il ne pourra pas répondre aux questions sur le programme de MAIN5. On relance donc un scrap avec `scraping_tool.py` pour que cette maquette soit ajoutée au corpus. Si le scrap ne la trouve pas, on l'ajoute manuellement. Ensuite, on relance une génération de vectorbase avec le script `vectorisation_generation.py`. La vector base est maintenant à jour, on redéploie le système et il peut parler de la MAIN5.

6 Implémentation du RAG

Le coeur du fonctionnement du RAG se retrouve dans le dossier "**src**" : dans ce dossier, on trouve 4 modules qui regroupent des fonctions de manière structurée en fonction de leur utilisation dans l'implémentation du RAG.

6.1 Gestion des clés

`keys_file.py` : Le module `keys_file.py` est utilisé pour la gestion des clés privées d'authentification et des URLs permettant d'envoyer des requêtes pour récupérer et envoyer des données. Il centralise ces clés et assure leur sécurité dans l'implémentation en les séparant du code source. Il permet de récupérer les clés stockées dans un fichier `.env` situé en local. On récupère donc ces clés grâce à la fonction `os.getenv()`, qui permet d'extraire les valeurs des variables d'environnement chargées à l'aide de la bibliothèque `dotenv`.

Voici les différentes clés utilisées:

- `OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")` : Cette clé est utilisée pour authentifier les requêtes envoyées à l'API d'OpenAI. Elle permet d'effectuer les appels aux deux modèles GPT que nous utilisons : gpt4o et gpt4o-mini. Elle est aussi utilisée pour la création d'embeddings lorsque l'on fait appel à ces modèles. Documentation OpenAI
- `HF_API_URL = os.getenv("HF_API_URL")` et `HF_API_TOKEN = os.getenv("HF_API_TOKEN")` : Ces clés permettent l'authentification pour l'API de Hugging Face. On accède ainsi aux modèles hébergés sur la plateforme, dans notre cas un modèle Hermes-3-8B, une version optimisée de Llama-3.1-8B. Hermes est conçu pour offrir des capacités avancées en conversation, un meilleur raisonnement et une cohérence améliorée sur un long contexte : il est donc un peu plus intéressant en tant qu'agent conversationnel, ce qui est précisément l'intérêt dans notre contexte. Hermes 3 - Hugging Face
- `HF_API_URL_EMBEDDING = os.getenv("HF_API_URL_EMBEDDING")` : Cette URL est utilisée pour effectuer les requêtes liées aux embeddings. Elle permet d'obtenir des représentations vectorielles de textes en utilisant un modèle d'encodage disponible sur Hugging Face, facilitant ainsi des tâches comme la recherche sémantique.

6.2 Gestion des prompts

`prompts.py` : Ce module définit l'ensemble des prompts utilisés pour structurer les interactions entre l'utilisateur et le modèle de langage. Le **prompt engineering** joue un rôle crucial dans l'optimisation des performances du chatbot en fournissant des instructions claires au modèle et en guidant ses réponses en fonction du contexte. Ce module contient plusieurs types de prompts, chacun ayant une fonction spécifique.

- **Prompt système principal** (`system_prompt`) : Ce prompt définit les instructions générales du chatbot. Il spécifie son rôle, son ton et ses limitations. Dans notre cas, le chatbot est conçu pour fournir des informations sur **Polytech Sorbonne**. Il est explicitement guidé pour répondre uniquement aux questions liées à l'école et ignorer les sujets externes. Ce prompt évite donc la divagation, mais peut être assoupli si besoin.
- **Prompt question-réponse** (`qa_prompt`) : Ce prompt est utilisé pour structurer l'interaction entre l'utilisateur et le modèle dans un format conversationnel. Il utilise `ChatPromptTemplate` et `MessagesPlaceholder` de LangChain pour organiser la structure des messages échangés.
  - `ChatPromptTemplate` : Il s'agit d'une **classe** de LangChain permettant de définir un format structuré pour les interactions avec un LLM. Il sert à construire des prompts modulaires en combinant différents types de messages (système, utilisateur, historique) et en y intégrant dynamiquement des variables d'entrée.

- `MessagesPlaceholder` : C'est une **fonction** de LangChain utilisée pour insérer dynamiquement l'historique des conversations dans le prompt. Elle permet au modèle de conserver la mémoire de la conversation.

  Ce prompt prend en compte plusieurs éléments, notamment les instructions système définies dans `system_prompt`, qui guident le comportement du modèle et assurent la cohérence des réponses. L'historique de la conversation est géré par `MessagesPlaceholder`, permettant au modèle de conserver le contexte des échanges précédents. Enfin, il prend en entrée l'input, qui correspond au message envoyé par l'utilisateur.

- **Prompt pour la reformulation des questions** (`contextualize_q_prompt`) : Ce prompt permet de reformuler une question en tenant compte du contexte précédent. Il est utilisé pour générer une version autonome de la question d'un utilisateur qui pourrait être ambiguë si prise isolément. Par exemple, si un utilisateur demande "Et les frais d'inscription ?", le modèle reformulera automatiquement cette question en "Quels sont les frais d'inscription à Polytech Sorbonne ?" pour assurer une meilleure compréhension du modèle. Comme (`qa_prompt`), il utilise `ChatPromptTemplate` et `MessagesPlaceholder`.

Ces différents prompts sont utilisés à travers les fonctions du module `llm.py`.

6.3 Mise en place de la chaîne de RAG

`llm.py` : C'est dans ce module que l'on structure concrètement les requêtes envoyées au modèle. On y met en place la chaine de RAG en utilisant les outils de la librairie langchain.

Voici les différentes fonctions qui y sont utilisées :

- `query_rag(llm, retriever, question, chat_history)` : Cette fonction effectue une requête en utilisant un retriever actif pour un système de type RAG.

  - `create_history_aware_retriever(llm, retriever, contextualize_q_prompt)` : Crée un retriever capable de prendre en compte l'historique des interactions afin de reformuler la question en fonction du contexte.Cette fonction permet d'améliorer la pertinence des réponses en reformulant la question posée par l'utilisateur en fonction du contexte des échanges précédents. Concrètement, lorsqu'une requête est soumise, elle est d'abord transformée en fonction des instructions du `contextualize_q_prompt`, qui reformule la question pour inclure les éléments pertinents. Une fois cette reformulation effectuée, la nouvelle requête est transmise au `retriever`, qui explore la base de données pour identifier les documents les plus pertinents en fonction du contenu de la requête. Les documents récupérés par le `retriever` sont ensuite renvoyés au modèle de langage.
  - `create_stuff_documents_chain(llm, qa_prompt)` : Cette fonction construit une chaîne de génération de réponse à partir des documents récupérés par le retriever. Elle prend en entrée un modèle de langage et un prompt spécifique (`qa_prompt`) pour formater les interactions avant la génération de réponse.
  - `create_retrieval_chain(history_aware_retriever, question_answer_chain)` : Cette fonction assemble le `history_aware_retriever` et la `question_answer_chain` pour former un pipeline complet d'interaction avec le modèle. Elle assure la coordination entre la récupération des documents et la génération de réponse.

    Une fois la question reformulée par le `history_aware_retriever` et les documents correspondants récupérés, ces derniers sont injectés dans la `question_answer_chain`. Cette dernière structure les informations et transmet la requête finale au modèle de langage, qui génère une réponse adaptée et enrichie du contexte trouvé. Ainsi, cette fonction finalise le processus RAG en reliant l'étape de récupération d'informations à celle de génération de réponse.

- `query_norag(llm, retriever, question, chat_history)` : Cette fonction effectue une requête sans utiliser de retriever actif. Elle est actuellement utilisée uniquement pour le modèle LLaMA.

  - Contrairement à `query_rag`, cette fonction ne récupère pas d'informations externes avant d'envoyer la requête au modèle.

– `qa_prompt.format_messages(chat_history, input=question, context="")` : Formate la requête en prenant en compte l'historique des messages mais sans ajouter d'informations récupérées.

– La requête est ensuite envoyée directement au LLM, qui génère une réponse.

6.4 Recherches dans la base de données vectorielle Chroma

`openai_chroma_search.py` : Ce module est dédié à la gestion des recherches dans la base de données vectorielle Chroma. Il permet d'interroger ChromaDB à l'aide d'OpenAI Embeddings et d'optimiser la récupération des documents pertinents pour le chatbot.

Voici les principales fonctions de ce module :

- `get_document_by_id(collection, vector_id)` : Cette fonction récupère un document spécifique dans la base de données Chroma à partir de son identifiant unique (`vector_id`).
- `filter_detection(question)` : Analyse la question de l'utilisateur afin d'extraire des métadonnées pertinentes sous forme de filtres exploitables pour la recherche de document. Cette extraction repose sur un appel à un modèle OpenAI.
- `retrieval_single_field(query, _filter, vectordb)` : Effectue une recherche basée sur un seul critère de filtrage (ex. spécialité) et retourne les documents pertinents.
- `retrieval(query, filter, vectordb)` : Recherche les documents les plus pertinents dans Chroma en appliquant plusieurs filtres simultanément (spécialité, statut, etc.).

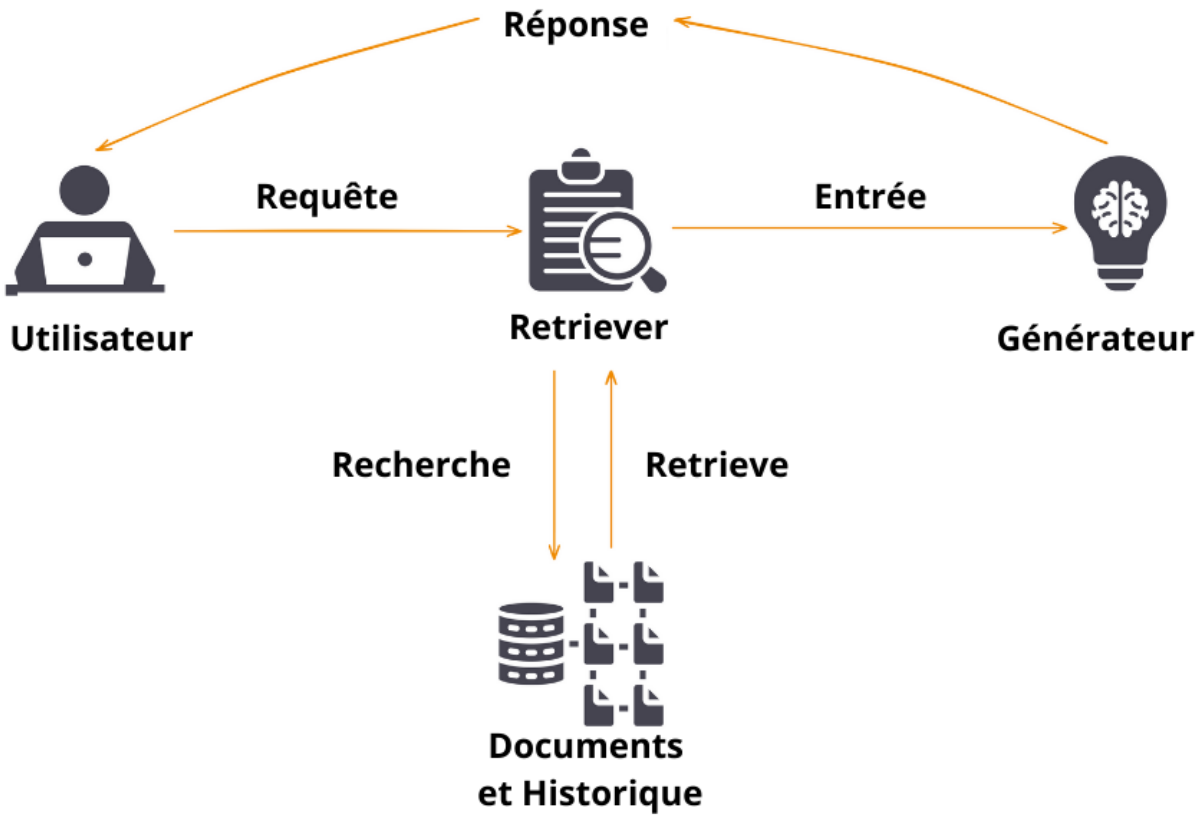


Figure 3: Processus RAG. Source : Superlinked

Les méta-données initialisés pour chaque vecteurs de la base de donnée sont :

- **ID** : un **Identifiant** au format **str** généré avec la bibliothèque uuid4 - unique_id = str(uuid4())
- **Source** : Le **Chemin vers le fichier** dont est issu le vecteur (Home/user/.../corpus/...), méta donnée plus utile pour la gestion par un humain permettant de contrôler certaines propriétées
- **Statut** : Le statut du fichier dont est issu le vecteur, il peut être **privé** ou **publique**, une telle information présente l'intérêt de pouvoir contrôler un peu mieux ce que le chatbot peut divulguer.
- **Speciality** : À quelle **Spécialité** de l'école Polytech Sorbonne le fichier source se rapportes (MAIN, MTX, EISE, GM...) il s'agit de la méta donnée principale car c'est la seule qu'on exploite actuellement pour faire des recherches filtrées
- **URL (si présent)** : Cette méta donnée est pour l'instant propre aux fichiers json qui dans notre travail étaient issus du scraping du site web de l'école, cette information permet de donner au chatbot de quoi rediriger vers des sources ou des pages précises, améliorant l'utilité de ses réponses et pouvant aussi faciliter la navigation du/des site(s) internet scrapé(s).

Ces choix ont été faits car ils étaient les plus pertinents et simples au moment du développement de ce projet ; mais d'autres méta données sont sans problème ajoutables si on le souhaite comme par exemple une date spécifique aux fichiers.

En ce qui concerne l'exploitation des meta données dans le RAG. C'est là que les fonctions de `openai_chroma_search.py` interviennent. Leur fonctionnement conjoint est simplement résumé dans la figure 3 ; en extrayant les filtres, on peut alors retrouver des vecteurs spécifiques et pertinents vis à vis de la question dans la partie de retrieval et ainsi construire une réponse.
Bien que celà aurait été très pertinent, nous n'avons malheureusement pas réussi faute de temps à implémenter une fonctionnalité qui, selon la question, empêche des appels ultérieurs dits "inutiles" à l'API, par exemple pour des messages tels que "Bonjour", "Merci", "Au revoir" ...

Des améliorations de ce côté sont donc nettement envisageables, avec par exemple une plus grande polyvalence des fichiers traitables pour la créatoin du store, l'ajout de plus de meta données, et biensûr une fonctionnalité pour éviter les appels superflus à l'API.

7 Gestion de l'historique et de la session utilisateur

La gestion de l'historique des conversations est un élément central du fonctionnement du chatbot. Un bon assistant conversationnel doit être capable de conserver le contexte des échanges pour fournir des réponses cohérentes et éviter les redondances. Cette section détaille les mécanismes de stockage, récupération et mise à jour de l'historique des interactions entre l'utilisateur et le chatbot.

7.1 Identification de l'utilisateur

Afin de garantir une gestion personnalisée de l'historique pour chaque utilisateur, un système d'identification unique est mis en place. L'identifiant de session de l'utilisateur (`user_id`) est récupéré à partir de l'URL lors de l'accès à l'interface du chatbot. Cette approche permet d'associer chaque utilisateur à son historique et d'assurer une continuité dans les échanges.
L'identification repose sur la fonction suivante :

- `get_user_id_from_url()` : Cette fonction extrait le `user_id` à partir des paramètres de l'URL via Streamlit. Si l'ID est fourni dans l'URL sous la forme `?user_id=X`, la fonction le récupère et le retourne. En l'absence de cet identifiant, elle renvoie une chaîne vide, ce qui empêche l'accès à l'historique.

7.2 Stockage et structure de l'historique

Comme dit précédemment chaque utilisateur est identifié par un **user_id**, qui sert de référence pour l'accès à son historique. Les conversations sont enregistrées sous forme de fichiers JSON, puis stockés dans un répertoire dédié **/var/data**. Dans cette structure, chaque utilisateur dispose d'un sous-dossier contenant son fichier `history.json`. Et chaque fichier JSON contient une liste d'échanges sous la forme d'un dictionnaire structuré comme suit :

```
[
  {
    "role": "user",
    "content": "texte de la question posé par l'utilisateur"
  },
  {
    "role": "assistant",
    "content": "texte de la réponse fourni par le modèle."
  }
]
```

Chaque message est ainsi enregistré avec :

- **role** : identifiant l'auteur du message (`user` ou `assistant`).
- **content** : le texte du message.

Cette structure garantit une récupération efficace des conversations et permet au chatbot de conserver le fil des échanges en intégrant les messages précédents dans ses réponses. Si ce stockage permanent pose un problème d'espace disque, on peut mettre en place une mécanique de suppression des historiques toutes les 24h, ce qui ne devrait pas poser de problème aux utilisateurs. L'historique peut être supprimé manuellement en utilisant la commande bash `rm -rf /var/data/*`.

7.3 Chargement et enregistrement des historiques

L'historique des conversations est chargé au début de toute consultation du chatbot, qui constitue déjà une interaction. Voici le code de chargement de l'historique dans le `show_ui()` :

```
# On obtient le user_id
user_id = get_user_id_from_url()
    if not user_id:
        st.error("Aucun user_id fourni dans l'URL (ex. ?user_id=ABC123).")
        st.stop()

# On recupere l'historique associé
if "chat_history" not in st.session_state:
    st.session_state["chat_history"] = load_history(user_id)
```

Lorsque l'utilisateur n'a pas d'historique connu, nous avons fait le choix de lui faire envoyer ("contre son gré" en quelque sorte) un message d'initialisation, afin d'engager la conversation avec le chatbot. Elle est ensuite immédiatement enregistrée, et l'utilisateur sans historique se retrouve avec un historique. Voilà cette gestion au niveau du code :

```
if len(st.session_state["chat_history"]) == 0:
        init_user_msg = (
            "Présente-toi en tant que chatbot représentant Polytech-Sorbonne. "
            "Présente l'école en quelques mots"
        )
        # if model_choice == "llama3" :
        #     response = query_norag(llm, retriever, init_user_msg, [])
        # else :
        response = query_rag(llm, retriever, init_user_msg, [])
        st.session_state["chat_history"].extend([
            {"role": "user", "content": init_user_msg},
            {"role": "assistant", "content": response if isinstance(response, str)
            else response["answer"]},
        ])
        save_history(user_id, st.session_state["chat_history"])
```

Par ailleurs, de manière générale, à chaque fois que l'utilisateur envoie un message, l'historique correspondant est récupéré, mis à jour avec la nouvelle interaction et sauvegardé dans le fichier JSON associé. Ces sauvegardes sont donc dynamiques.
Cette gestion repose sur plusieurs fonctions essentielles :

- `get_user_history_file(user_id)` : Cette fonction construit le chemin du fichier d'historique correspondant à l'utilisateur à partir de son `user_id`. Elle crée également le répertoire utilisateur s'il n'existe pas encore.
- `load_history(user_id)` : Permet de charger l'historique des conversations d'un utilisateur en lisant le fichier `history.json`.
- `save_history(user_id, history)` : Sauvegarde l'historique mis à jour d'un utilisateur en écrivant la liste des interactions dans son fichier `history.json`.

Grâce à ce mécanisme, le chatbot peut récupérer les échanges passés et assurer une continuité dans la conversation, en offrant des réponses contextuelles basées sur l'historique.

Pour exécuter le chatbot en local, il est nécessaire de s'assurer que l'environnement de stockage des historiques est bien configuré. Avant toute chose, il convient de vérifier que le dossier `/var/data/` existe. Si ce n'est pas le cas, il peut être créé à l'aide de la commande suivante :

```
sudo mkdir -p /var/data
sudo chmod 777 /var/data
```

Une fois cette étape réalisée, l'application peut être lancée en exécutant les commandes suivantes :

```
git clone https://github.com/Owkly/Projet-Chatbot
cd Projet-Chatbot
pip install -r requirements.txt
streamlit run main.py
```

Streamlit démarre alors un serveur local, rendant l'interface du chatbot accessible via une URL par défaut. Pour interagir avec le chatbot, il est indispensable de spécifier un identifiant utilisateur dans l'URL. Ainsi, après avoir lancé Streamlit, il suffit d'ouvrir un navigateur web et de se rendre à l'adresse `http://localhost:8501/?user_id=x`, en remplaçant `x` par un identifiant unique. Ce mécanisme permet de différencier les sessions utilisateur et d'assurer la gestion correcte des historiques de conversation.

Grâce à cette approche, le chatbot peut être testé localement sans générer d'erreur liée à l'absence d'un `user_id`. Elle permet également d'évaluer la gestion des différentes sessions utilisateur et d'observer comment l'historique est sauvegardé et récupéré au fil des interactions.

8 Paramétrage de l'interface graphique

Le framework Streamlit n'est pas des plus flexibles. Nous avons donc pris un certain temps à trouver un moyen efficace de reparamétrer son interface graphique. Tout d'abord, voyons le fichier `/.streamlit/config.toml` :

```
[theme]
primaryColor="#429DDA"
backgroundColor="#FFFFFF"
secondaryBackgroundColor="#f1faff"
```

Ces trois premières lignes servent à définir les couleurs du thèmes de l'app streamlit. Par défaut, c'est du noir/orange ; ici, on a choisi un blanc/bleu pour rappeler les couleurs de Polytech et de Sorbonne-Université. Ensuite :

```
[client]
toolbarMode="minimal"
[ui]
hideTopBar=true
```

Cette partie s'assure que la toolbar et la topbar de streamlit, qui sont parfois envahissantes visuellement, soient forcée à être altérées. La toolbar est passée en mode minimal et la topbar est cachée. Sans cela, outre l'encombrement visuel, on peut avoir des bugs d'affichage sur téléphone/tablette rendant l'application totalement inutilisable. Cela dégage aussi la place pour introduire nos propres éléments. Au-delà de cette configuration basique, nous avons du faire des ajustements forcés en CSS dans le `main.py`, au début de la fonction `show_ui()` :

```
st.markdown(
        """
        <style>
        [data-testid="stChatMessageAvatarUser"][aria-label="assistant"]
        svg { background-color: #FFFFFF !important; }
        [data-testid="stChatMessageAvatarUser"][aria-label="user"]
        { background-color: #429DDA !important; }
        ... ETC ETC
        [data-testid="stSidebarUserContent"] { padding-bottom:0px!important; }
        [data-testid="stSidebarCollapsedControl"]::after
        { content: "Changer de LLM"; margin-left: 15px; font-size: 14px; color: inherit; }
    </style>
        """,
        unsafe_allow_html=True
    )
```

Nous n'avons pas copié-collé le bloc-entier ici car il est assez long. Ici, ce sont tous les éléments de l'interface streamlit qui sont identifiés puis recalibrés pour correspondre aux besoins graphiques. Ce bout de code est assez précieux car trouver les sélecteurs des éléments visuels de streamlit un par un prend beaucoup de temps, et la documentation n'aide pas beaucoup car elle n'est pas encore au point. En cas de modification, a fortiori d'élagage, il faudrait garder ce code commenté pour ne pas le perdre.

Toujours dans le `show_ui()` du `main.py`, juste après ce tronçon, il y a un autre bout de code très important :

```
st.header("PolyChat", divider=False)
with st.sidebar:
    model_choice = st.selectbox("Choisissez un modèle :", ALL_MODELS, index=0)
st.info(f"Modèle actuellement utilisé : {model_choice}")
```

Ce code permet d'introduire la sidebar définissant le choix dynamique du LLM. Les restrictions de streamlit font qu'il n'est pas possible de l'inclure dans la topbar ni sous les messages, alors cette option est la meilleure. Néanmoins, il convient de surveiller les mises à jour du framework, car la possibilité d'implémenter ce sélecteur dans la topbar devrait tout de même finir par voir le jour.

9 Déploiement sur un site web

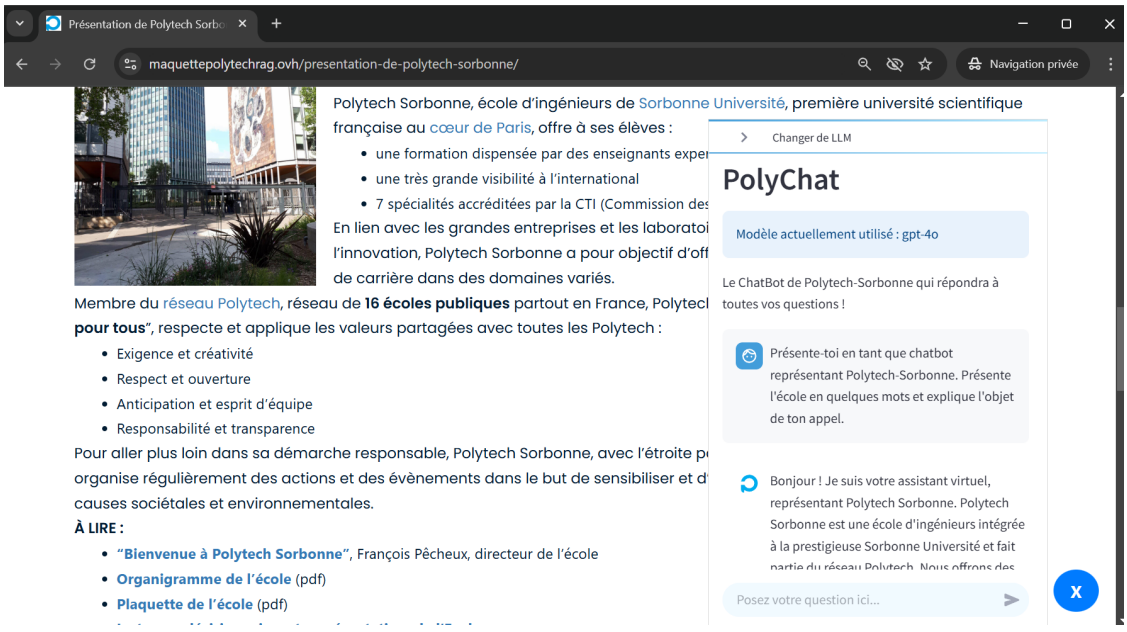
Le déploiement sur un site web peut être fait très simplement sur n'importe quel site web tant que l'injection JS y est possible. En effet, le fichier `/web/code_snippet_chat` contient un script JS (JavaScript) injectable qui streame l'app streamlit (sous forme de frame) sur la page web courante.

Examinons le bout de code important de ce JavaScript :

```
(function() {
    // Récupération de l'ID dans le localStorage
    let visitorId = localStorage.getItem("visitorId");

    // Génération d'un ID pseudo-aléatoire s'il n'existe pas
    if (!visitorId) {
      visitorId = Math.random().toString(36).substring(2) + Date.now().toString(36);
      localStorage.setItem("visitorId", visitorId);
    }

    // Construction de l'URL pour l'iframe
    const chatbotURL = "https://chat-jg2t.onrender.com/?user_id=" + visitorId;

    // Affectation de l'URL à l'iframe
    document.getElementById("chatbot-frame").src = chatbotURL;
  })();
```

La logique d'id de visiteur est ici simple à comprendre, et on peut voir comment elle utilisée pour construire une URL personnalisée. Ici, la base de l'URL est `https://chat-jg2t.onrender.com/` pour la simple et bonne raison qu'il s'agit de l'URL d'accès à notre VPS Streamlit. Dans un autre contexte de déploiement, il faudrait remplacer cette URL par l'URL permettant d'accéder à l'interface streamlit brute, mais la logique resterait identique (tant que l'on ne passe pas en local).

Le reste du `/web/code_snippet_chat` contient des fonctions d'affectation et de comportement qu'il n'est pas utile de détailler dans la présente documentation. Le fichier`/web/style.css` contient quant à lui un clonage du style CSS du site de l'école dans sa version de février 2025. Il a servi a réaliser un clonage du site original sur notre site web de maquettage, afin de voir comment rendrait le Chatbot s'il était web-intégré. Voici un aperçu en HD :



10 Limitations et faisabilité

Bien que le chatbot PolyChat soit tout à fait fonctionnel, certaines limitations subsistent et méritent d'être analysées afin d'évaluer la faisabilité d'un déploiement définitif, et pour orienter les futures améliorations.

L'une des principales contraintes concerne la consommation de ressources pour les modèles locaux de type LLaMA comme HERMES3-8B, qui nécessitent - même dans leur version légères - un serveur relativement puissant avec une infrastructure adaptée pour garantir des performances optimales. En effet, des GPU puissants sont nécessaires pour paralléliser massivement l'inférence des réseaux neuronaux de ces modèles, sans quoi le temps d'exécution devient incompatible avec un agent conversationnel censé répondre en quelques secondes tout au plus. Cette exigence en termes de calcul peut entraîner des coûts élevés et des difficultés de maintenance, que ce soit sur des clusters de l'école/université ou sur des serveurs virtuels privés (VPS) de cloud computing.

De plus, quand bien même l'université et l'école possèderaient les infrastructures théoriquement aptes à faire tourner de tels modèles localement, il se pose la question de l'utilisation même de ces clusters. En effet, il faudrait établir des pipes de transmissions de requêtes sécurisées aux nodes concernés, ce qui n'est pas évident du tout. De la même manière, il se poserait la question logistique de l'ordonnancement entre ces nodes, ce qui peut vite devenir un casse-tête.

Enfin, même si tout cela pouvait être mis en place la facture d'électricité associée aux calculs d'inférence serait probablement assez élevée pour être dissuasive.

De l'autre côté, les modèles GPT (qui tournent systématiquement de manière distante sur les clusters de l'entreprise OpenAI) présentent des performances similaires voire meilleures, pour un prix bien moindre : durant le développement du projet, nous avons consommé seulement 15$ en requêtes aux clusters d'OpenAI, contre plus de 70$ en requêtes aux clusters de VPS faisant tourner LLaMA sur leurs GPU (alors que nous avons effectué bien plus de requêtes à GPT qu'à LLaMA).

In fine, PolyChat peut fonctionner avec des modèles OpenAI pour un budget raisonnable, mais pas LLaMA. Ce dernier est donc probablement à écarter.

Pour ce qui est de l'hébergement du serveur faisant tourner le code python/streamlit, il peut être effectué par l'école ou même à très bas coût en ligne puisque 2CPU, 2GB de RAM et 2GB d'espace disque suffisent amplement à son bon fonctionnement, même sollicité par plusieurs utilisateurs. Pas besoin de GPU puissant ni de quoi que ce soit de plus.

Toutefois, il faudrait veiller à ce que le système soit entretenu (qu'il s'agisse de maintenance ou de mise à jour du corpus), ce qui implique forcément un coût humain, fût-il ponctuel.

Un autre point négatif à considérer est l'ergonomie de l'interface utilisateur en ce qui concerne l'affichage responsive sur mobile. L'application Streamlit, utilisée pour l'interface principale, ne s'adapte pas toujours parfaitement aux écrans de petite taille, ce qui peut nuire à l'expérience utilisateur. Une optimisation de l'affichage responsive permettrait d'améliorer l'accessibilité du chatbot sur différentes plateformes et appareils, ce qui serait assez important puisque le public cible de PolyChat est jeune et utilise donc forcément beaucoup ce type d'appareil. Mettre en place une gestion responsive adaptée demanderait probablement plusieurs semaines de travail à un développeur web, ce qui n'est pas négligeable : le coût financier associé pourrait être dissuasif. Idem pour un système d'authentification permettant de conserver un historique, par exemple.

Une perspective intéressante serait de confier la suite de ce projet à un autre groupe d'étudiants, afin de le perfectionner, d'en mener le déploiement effectif et la maintenance. Cela réduirait les coûts financiers et permettrait d'avoir un suivi sur un laps de temps conséquent, pour mieux cerner les problèmes sur le moyen, puis long terme. On pourrait alors également imaginer de la data-analyse sur les requêtes envoyées par les utilisateurs du chatbot.

11 Conclusion

Le développement de PolyChat a permis de concevoir une **solution efficace et fonctionnelle** pour offrir une assistance conversationnelle sur le site de Polytech Sorbonne. Ce projet a couvert **plusieurs aspects techniques majeurs**, chacun contribuant à la robustesse et à la pertinence du système mis en place.

L'acquisition et la structuration des données nécessaires à l'entraînement et au bon fonctionnement du chatbot. **Un scraping web avancé** a été mis en place afin d'extraire des informations pertinentes sous forme de fichiers **PDF et JSON**. Ces données ont ensuite été normalisées et structurées pour en faciliter l'exploitation dans le système RAG.

La récupération efficace de l'information, un processus de génération des embeddings a été implémenté, optimisé pour fonctionner avec des modèles **OpenAI Embeddings et Hugging Face**. Les documents ont été indexés dans une vector base, en utilisant **FAISS et ChromaDB** pour retrouver rapidement les documents pertinents, facilitant ainsi la recherche sémantique et l'optimisation des requêtes utilisateurs.

L'implémentation d'un système de Retrieval-Augmented Generation. Pour cela, un module dédié à **l'authentification et à la gestion sécurisée des clés API** a été conçu. L'optimisation du chatbot a également reposé sur une **définition rigoureuse des prompts**, grâce à l'utilisation de la librairie **LangChain**. Enfin, une gestion avancée de **l'historique conversationnel** a été développée, garantissant une continuité dans les échanges avec les utilisateurs.

Une interface utilisateur interactive a été conçue avec **Streamlit**, enrichie par une personnalisation graphique avancée grâce à des ajustements CSS et aux fonctionnalités natives de Streamlit. Le chatbot a été intégré sur le site web via un script JavaScript, facilitant ainsi son déploiement et son accessibilité.
Le système a été testé et déployé sur une maquette du site de Polytech Sorbonne. L'hébergement a été réalisé sur un VPS, garantissant une accessibilité continue au chatbot, tout en optimisant les coûts. **Les tests ont démontré que les modèles OpenAI (GPT-4o) constituaient une solution plus économique et performante que l'exécution locale de modèles comme LLaMA**, qui requiert des ressources importantes en calcul GPU. La gestion de la maintenance et des mises à jour a également été optimisée afin de permettre un rechargement rapide du corpus et une régénération des embeddings lorsque nécessaire.

Bien que le projet soit fonctionnel, certaines améliorations peuvent être envisagées. Une optimisation de l'interface utilisateur permettrait d'améliorer la compatibilité avec les appareils mobiles. De même, une meilleure gestion des appels API pourrait réduire les requêtes inutiles et optimiser la consommation des ressources. L'ajout de nouvelles méta-données enrichirait le filtrage et la personnalisation des réponses, tandis que l'automatisation avancée du scraping et de la mise à jour du corpus renforcerait l'efficacité du système.

En conclusion l'évolution future de PolyChat pourrait inclure une intelligence contextuelle améliorée, une personnalisation accrue et un déploiement définitif sur le site.

En suivant ce lien vous pourrez visualiser une vidéo du premier prototype de PolyChat Vidéo maquette

12 Annexes

12.1 Glossaire :

- **LLM** : Un grand modèle linguistique (Large Language Model, LLM) est un type de programme d'intelligence artificielle (IA) capable, entre autres tâches, de reconnaître et de générer du texte. Les LLM sont entraînés sur de vastes ensembles de données, d'où l'emploi du terme "large" (grand) dans la dénomination anglaise. Ils s'appuient sur l'apprentissage automatique (Machine Learning, ML) et plus spécifiquement sur un type de réseau neuronal dénommé « modèle transformateur ». En voir davantage ici

- **GPT** : Acronyme de *Generative Pre-trained Transformer*, il s'agit du LLM le plus utilisé au monde, développé par la société *OpenAI*. Il a plusieurs déclinaisons plus ou moins rapides, puissantes et récentes : `GPT-3, GPT-3.5, GPT-4O, GPT-4O-MINI, GPT-O1`, etc. Il ne peut être éxécuté que sur les clusters privés d'*OpenAI* et n'est pas open-source.

- **LLaMA** : Acronyme de *Large Language model Meta Ai*, il s'agit d'un LLM également très utilisé, développé par la société *Meta* (ex-*Facebook*). Il a lui aussi plusieurs déclinaisons : `LLaMA2-70B, LLaMA3-8B, LLaMA3.1-70B, LLaMA3.2-90B`, etc. Contrairement à `GPT`, il peut être éxécuté localement et frôle l'*open-source*.

- **Hermes-3-8B** : LLM basé sur `LLaMA3.1-8B` mais customisé par une équipe de chercheurs (*NOUS RESEARCH*) pour présenter de meilleures capacités conversationnelles en tant qu'agent personnalisé, ce qui est utile dans le cadre du présent projet. En voir davantage ici

- **Embeddings** : Représentation numérique d'un texte sous forme d'une multitude de vecteurs en capturant la signification sémantique. Il existe naturellement plusieurs procédés mathématiques pour créer des embeddings à partir de documents, chacun ayant ses avantages et ses inconvénients. On peut par exemple citer les embeddings de type `OpenAIEmbeddings` qui sont optimisés pour les modèles `GPT` de la société *OpenAI*. En voir davantage ici

- **Vector Bases** : Systèmes de stockage et de recherche optimisés pour gérer des embeddings. Ces bases de données permettent d'effectuer des recherches rapides par similarité selon plusieurs métriques (cela peut être le cosinus, la distance euclidienne, ...) afin de trouver les documents les plus en adéquation avec la requête de l'utilisateur. Il en existe différents types : `FAISS`, `ChromaDB`, `Pinecone`, En voir davantage ici

- **Corpus** : Ensemble des documents qui seront transformés en *embeddings* puis stockés dans la *vector base*. Il contient les informations personnalisées que nous voulons apporter au système.

- **Retriever** : Mécanisme permettant de récupérer des documents pertinents dans une *vector base* ayant elle-même été générée à partir d'un *corpus*.

- **RAG (Retrieval-Augmented Generation)** : Il s'agit d'une approche qui combine la récupération d'informations (*retrieval*) et la génération de texte (*generation*) à l'aide d'un modèle de langage. Le système RAG utilise un retriever pour récupérer des documents pertinents et les intégrer dans la requête avant d'effectuer la génération de réponse, améliorant ainsi la précision et la cohérence du modèle. En voir davantage ici

12.2 Développement durable et responsabilité sociétale

Aspect écologique

Le fonctionnement de notre chatbot repose sur des requêtes API envoyées à des modèles de langage. L'utilisation de ces modèles n'est pas sans impact sur l'environnement. En effet, l'entraînement des modèles de langue nécessite d'importantes ressources de calcul, mobilisant des serveurs hautement énergivores, et dont l'énergie n'est pas toujours verte. De plus, chaque requête envoyée aux LLM consomme également de l'énergie, contribuant ainsi à l'empreinte énergétique des data centers.

Pour limiter ces impacts, nous avons inclus la possibilité d'utiliser des modèles optimisés, comme GPT-4o-mini ou Hermes-3-8B, qui réduisent grandement la consommation énergétique puisqu'ils se basent sur des milliers de fois moins de neurones que les versions classiques des LLM.

De plus, il est bon de noter que notre utilisation de bases de données vectorielles locales bien optimisées réduit le besoin de requêtes répétées vers des modèles externes. Notre système de cache et de pré-calcul des embeddings permet de réduire le nombre de requêtes effectuées vers des acteurs externes.

En sus, le code étant pensé pour être flexible et web-déployable, il est tout à fait possible de choisir des hébergeurs web/VPS à empreinte carbone nulle par exemple, même si leur coût est généralement plus élevé que celui des hébergeurs classiques. En combinant cela à un modèle local comme Hermes-3-8B, on peut avoir un contrôle total sur l'empreinte carbone du système.

Toutefois, ces considérations et optimisations techniques ne suffisent pas pleinement à réduire l'empreinte écologique du chatbot. Il est également essentiel de sensibiliser les utilisateurs à une utilisation responsable de l'outil. Chaque requête envoyée mobilise des ressources informatiques, et une utilisation excessive ou non justifiée peut entraîner une surconsommation inutile d'énergie, que cette dernière soit verte et durable ou non.

Aspect social et éthique

Ce chatbot permettrait aux étudiants, futurs étudiants et parents d'obtenir des réponses rapides et précises sur Polytech Sorbonne, sans nécessiter de longues recherches ni d'interactions directes avec un service administratif. En centralisant et structurant l'information, il améliorerait l'expérience utilisateur en rendant l'accès aux données plus fluide et accessible à tous.

Cela pourrait être particulièrement utile à bon nombre de personnes. On pourrait d'emblée penser à celles parlant mal le français, qui auraient du mal à lire le site ou à communiquer avec le secrétariat par téléphone. Elles ne seraient plus défavorisées puisqu'elles pourraient communiquer avec le LLM dans un grand nombre de langues différentes. Idem pour les personnes malentendantes à la recherche d'une information ne figurant pas explicitement sur le site : elles pourraient passer par ce Chatbot, qui a un accès 'manuel" à certaines données n'y figurant pas.

Il va sans dire que PolyChat répond à toute personne de la même manière : plus aucun risque de traitement discriminatoire ni inapproprié, puisque les LLM ne font aucune différence entre les utilisateurs et qu'ils ont été entraînés pour être aussi respectueux et serviables que possible.

Outre cela, il déchargerait l'administration de l'école d'un bon nombre de tâches puisqu'elle recevrait moins d'appels et de courriels leur demandant des informations. À terme, des centaines d'heures de travail humain pourraient être économisées à ce niveau-là.

Cependant, pour garantir un service fiable et éthique, il faut s'assurer que le modèle fournit des réponses exactes et cohérentes. En effet, les LLM sont susceptibles de produire des *hallucinations*, c'est-à-dire des informations inventées et erronées. Pour éviter cela, nous avons mis en place plusieurs garde-fous, notamment en les limitant aux informations leur ayant été fournies, et en leur imposant d'indiquer explicitement à l'utilisateur ses méconnaissances. Malgré ces précautions, il faudrait évaluer la possibilité d'informer l'utilisateur que le chatbot peut parfois se tromper. Cette transparence permettrait d'encourager une utilisation critique et réfléchie de l'outil.