Cours : Construire une Application RAG Locale de A à Z

Introduction

Bienvenue dans ce cours intensif de 3 jours, conçu spécifiquement pour des élèves ingénieurs curieux et désireux de maîtriser les technologies d'intelligence artificielle de pointe. Nous allons adopter une approche résolument pratique, *learning by doing*, pour démystifier l'un des concepts les plus puissants de l'IA générative : le RAG (Retrieval-Augmented Generation). L'objectif final est clair et ambitieux : à la fin de ces trois jours, chacun d'entre vous aura construit de ses propres mains une application complète de Question-Réponse, capable de discuter intelligemment du contenu de vos propres documents. Le tout, en fonctionnant entièrement sur votre machine, garantissant une confidentialité totale de vos données. Cette compétence est aujourd'hui fondamentale, car elle représente la clé pour transformer le potentiel des grands modèles de langage (LLMs) en solutions d'entreprise concrètes, sécurisées et à forte valeur ajoutée.

Jour 1 : Les Fondations du RAG – Le "Pourquoi" et le "Comment"

Module 1.1 : Contexte et Enjeux Stratégiques du RAG

Avant de plonger dans la moindre ligne de code, il est crucial de comprendre pourquoi nous le faisons. Le RAG n'est pas simplement une technique amusante ; c'est une réponse directe et élégante aux limitations fondamentales des LLMs. Nous allons positionner le RAG comme une solution stratégique, permettant aux entreprises d'exploiter la puissance des LLMs sur leurs données internes de manière pertinente, vérifiable et surtout, sécurisée.

1. Le Problème : Les Limites des LLMs Standards

Les grands modèles de langage pré-entraînés présentent deux obstacles majeurs :

- **Manque de connaissances spécifiques** : un modèle comme `gemma3:12b` ignore tout de votre entreprise. Il ne peut donc pas répondre à des questions comme : "Quel était le chiffre d'affaires du produit X au dernier trimestre ?".
- **Risque d'hallucination** : lorsqu'un LLM ne connaît pas la réponse, il invente une information plausible mais fausse. Dans un contexte professionnel (médical, financier, juridique), c'est inacceptable.

2. La Solution : Le RAG

Le RAG (Retrieval-Augmented Generation) consiste à donner un "cerveau externe" au LLM. On cherche d'abord l'information pertinente dans une base privée, puis on l'injecte dans la question posée au LLM. L'analogie : un expert humain consulte ses notes avant de répondre.

3. L'Enjeu de la Sécurité : RAG Local vs API Cloud

Deux approches existent, avec des implications majeures en termes de sécurité :

RAG Local (ce que vous construisez)	**API Cloud (ex : OpenAI, Google)**
Confidentialité : les données restent dans l'entreprise	Risque de fuite : les données sont envoyées à un tiers, donc moins de contrôle sur leur usage
Contrôle total : choix du modèle, version, infrastructure	Dépendance : disponibilité, évolutions, prix décidés par le fournisseur
Modèles open source possibles	Modèles propriétaires uniquement
Pas de coûts par requête : investissement initial dans le matériel	Coûts récurrents : facturation par requête ou token
Performance : potentiellement très rapide si l'infrastructure est optimisée localement, mais dépend du matériel disponible	Performance : généralement stable et optimisée, mais dépend de la latence réseau
Scalabilité : limitée par les ressources physiques ; montée en charge plus coûteuse et complexe	Scalabilité : quasi-illimitée ; le fournisseur gère la montée en charge automatiquement

TABLE 1 – Comparaison entre RAG Local et API Cloud

Astuce :
Pour un usage professionnel, souvent on choisit un mix hybride :
- **RAG Local** pour les données sensibles ou critiques.
- **API Cloud** pour des traitements moins sensibles ou pour bénéficier de modèles très puissants sans gros investissement matériel.

Module 1.2 : L'Anatomie d'un Système RAG

Ce module dissèque l'architecture complète d'un système RAG, du chargement des documents jusqu'à la génération d'une réponse.

1. Présentation de la Stack Technique

Notre application repose sur quatre technologies open-source majeures :

— **Ollama** : moteur local pour exécuter modèles de langage et d'embedding sans Internet.
— **LangChain** : framework orchestrateur du pipeline RAG.

— **ChromaDB** : base de données vectorielle pour stocker et rechercher nos documents.
— **Streamlit** : interface utilisateur web interactive en Python.

2. Le Flux de Données en Deux Phases

Phase 1 : Ingestion (Indexation)

1. Chargement du document via un loader (PyPDFLoader, CSVLoader, TextLoader).
2. Découpage en *chunks* avec `RecursiveCharacterTextSplitter`.
3. Création des embeddings via `OllamaEmbeddings`.
4. Stockage dans ChromaDB.

Phase 2 : Génération (Récupération et Réponse)

1. L'utilisateur saisit une question.
2. Création de l'embedding de la question.
3. Recherche de similarité dans ChromaDB pour trouver les chunks les plus proches.
4. Construction du prompt avec les chunks + la question.
5. Génération de la réponse par le LLM (`ChatOllama`).

Module 1.3 : Atelier Pratique – L'Ingestion de Documents

Passons à l'action ! L'objectif est de compléter la fonction `process_uploaded_file` pour charger et découper des documents.

Objectif de l'Atelier

Permettre à l'application de :

— charger un fichier PDF, TXT ou CSV ;
— le lire et le découper en chunks ;
— préparer ces chunks pour l'étape suivante.

Concepts Théoriques à Appliquer

— **Loaders** : `PyPDFLoader`, `CSVLoader`, `TextLoader`.
— **TextSplitter** : `RecursiveCharacterTextSplitter` avec `chunk_size=1000` et `chunk_overlap=20`

Instructions pour le Code

1. Instancier la classe **Loader** appropriée (`PyPDFLoader`, `CSVLoader`, `TextLoader`) en lui passant le chemin du fichier temporaire.
2. Appeler la méthode nécessaire sur le *loader* instancié pour charger les `documents`.
3. Créer une instance du `RecursiveCharacterTextSplitter` en appliquant les paramètres **1000** et **200** pour la taille et le chevauchement du texte.
4. Appeler la méthode de découpage sur le *splitter* pour obtenir la liste finale des `chunks`.

Vérification

Exécutez l'application Streamlit, chargez un fichier et cliquez sur **Traiter le document**. Un message de succès confirme l'ajout du document.

Félicitations ! Vous avez construit la porte d'entrée de votre application RAG. Demain, nous attaquerons le cœur du système : transformer ces chunks en une base de connaissances interrogeable.

Jour 2 : Le Cœur du Réacteur - Vectorisation et Génération

Module 2.1 : La Magie des Embeddings et des Bases Vectorielles

Ce module aborde le concept le plus fascinant et le plus fondamental du RAG : comment un ordinateur peut-il « comprendre » le sens du texte au-delà des simples mots-clés ? La réponse se trouve dans les embeddings. Nous allons les présenter comme une méthode de traduction du langage humain vers le langage universel des mathématiques. Ensuite, nous verrons comment les bases de données vectorielles, comme ChromaDB, agissent comme des bibliothèques hautement spécialisées pour organiser et rechercher dans cette connaissance mathématique.

Qu'est-ce qu'un Embedding ? Un embedding est une représentation numérique du sens d'un morceau de texte. Imaginez que chaque mot, phrase ou document est représenté par une coordonnée sur une carte géographique immense en plusieurs milliers de dimensions. Sur cette carte, les concepts sémantiquement similaires sont placés très près les uns des autres.

— Le mot « roi » serait très proche de « reine » et de « monarque ».
— La phrase « le chiffre d'affaires de l'entreprise a augmenté » serait proche de « les revenus ont connu une croissance ».

Techniquement, ces « coordonnées » sont un vecteur : une longue liste de nombres (par exemple, 768 ou 1024 chiffres à virgule flottante). C'est le rôle de modèles d'IA spécialisés, comme `nomic-embed-text` que nous utiliserons via Ollama, de lire un texte et de générer ce vecteur unique qui capture son essence sémantique.

Qu'est-ce qu'une Base de Données Vectorielle ? Une base de données traditionnelle est excellente pour trouver des correspondances exactes (par exemple : trouver l'utilisateur avec l'ID « 123 »). Une base de données vectorielle comme ChromaDB est, quant à elle, optimisée pour une tâche prodigieuse : étant donné un vecteur de requête, trouver très rapidement les vecteurs les plus proches dans sa collection.

C'est ce qu'on appelle la recherche de similarité ou recherche sémantique. Au lieu de chercher des mots-clés, on cherche le sens le plus proche.

— Exemple concret : Si l'utilisateur demande « quel est le salaire du PDG ? », notre application va créer l'embedding de cette question. ChromaDB ne cherchera pas les mots « salaire » ou « PDG ». Il cherchera le vecteur le plus proche et pourra

ainsi retrouver un chunk contenant « la rémunération annuelle du directeur général s'élève à... », même si aucun des mots de la question n'est présent.

Module 2.2 : Atelier Pratique - Création de la Base de Connaissances

Dans cet atelier, nous allons coder la fonction `update_vectorstore`. Son rôle est de prendre les chunks créés précédemment, de les transformer en vecteurs via un modèle d'embedding local, et de les stocker de manière organisée et permanente dans notre base ChromaDB.

Objectif de l'Atelier Compléter le code pour transformer les chunks de texte en embeddings et les indexer dans ChromaDB.

Concepts Théoriques

- **OllamaEmbeddings** : Classe de LangChain reliant aux modèles locaux (par ex. `nomic-embed-text`).
- **Logique d'upsert** : Ajouter des chunks si la base existe déjà, sinon la créer.
- **Chroma.from_documents** : Crée directement une base à partir des chunks, embeddings et répertoire de persistance.

Instructions de Code

1. Initialiser la classe `OllamaEmbeddings` en lui passant le modèle sélectionné (`embedding_model`) et l'URL de base (`OLLAMA_BASE_URL`).
2. Mettre en œuvre la **Logique d'upsert** : si la `vectorstore` existe déjà, utiliser la méthode appropriée (sur l'objet `st.session_state.vectorstore`) pour **ajouter** les nouveaux `chunks`.
3. Sinon, utiliser la méthode de classe `Chroma` pour créer la nouvelle base à partir des `chunks`, des `embeddings` et du répertoire de persistance.

Module 2.3 : La Génération Augmentée par la Récupération (RAG)

Ce module explique comment connecter tous les éléments : recherche sémantique, construction de prompt et génération de réponse par le LLM.

Le Retriever LangChain permet de transformer une vectorstore en retriever via :
`retriever = st.session_state.vectorstore.as_retriever(search_kwargs={"k":4})`.

Le Prompt Augmenté Structure recommandée :

```
prompt = f"Voici le contexte :\n{context}\n\nQuestion : {query}\nRéponse :"
```

Paramètres du LLM

- `temperature=0.3` : réponses factuelles et déterministes.
- `streaming=True` : génération mot par mot pour plus de fluidité.

Module 2.4 : Atelier Pratique - Le Pipeline de Réponse

Dans cet atelier, nous complétons la fonction `generate_response_stream` :

Objectif Gérer les deux cas : chatbot classique (pas de RAG) et chatbot augmenté par RAG.

Instructions de Code

1. Mode direct (pas de RAG) : si la `vectorstore` est absente, instancier `ChatOllama` (avec `temperature=0.3` et `streaming=True`) et renvoyer le résultat de la méthode de génération en `stream`.
2. Mode RAG :
   — Transformer le `vectorstore` en `retriever` en appliquant la méthode appropriée. Configurer la recherche pour récupérer les **4** documents les plus pertinents.
   — Appeler la méthode requise sur le `retriever` pour lancer la recherche de similarité basée sur la `query`.
   — Construire la variable `context` en joignant les contenus (`page_content`) des documents récupérés (`docs`) avec la chaîne `"\n"`.
   — Construire le prompt augmenté en utilisant une *f-string* pour injecter le `context` et la `query` dans le format de prompt recommandé (vu en Module 2.3).
   — Appeler la méthode de génération en *streaming* sur l'instance `ChatOllama` créée (avec les bons paramètres).

Bravo ! Le moteur de votre application est désormais complet. Demain, nous lui construirons une interface utilisateur élégante.

Jour 3 : L'Interface Utilisateur et les Bonnes Pratiques

Module 3.1 : Créer une Interface Intuitive avec Streamlit

Bienvenue en ce troisième et dernier jour. Notre moteur RAG est puissant, mais pour l'instant, il n'est accessible qu'à un développeur. Ce module présente **Streamlit**, un outil révolutionnaire qui permet de transformer un script Python en une application web interactive, sans écrire de HTML, CSS ou JavaScript.

1. Les Composants de Base de Streamlit

En parcourant notre code, vous verrez plusieurs fonctions préfixées par `st.`. Ce sont les "widgets" de Streamlit. Voici les principaux que nous utilisons :

— `st.sidebar` : crée une colonne latérale à gauche, parfaite pour les options de configuration.
— `st.title` et `st.subheader` : pour créer des titres et organiser le contenu.
— `st.selectbox` : génère un menu déroulant, idéal pour choisir nos modèles.
— `st.file_uploader` : affiche un widget pour charger des fichiers locaux.
— `st.button` : crée un bouton cliquable pour déclencher une action.
— `st.chat_input` et `st.chat_message` : composants spécialisés pour construire rapidement une interface de chat.

— `st.dataframe` : affiche des données tabulaires, comme notre liste de documents.
— `st.expander` : crée une section repliable, parfaite pour afficher des informations supplémentaires (par ex. les sources d'une réponse).

2. La Gestion de l'État avec `st.session_state`

À chaque interaction de l'utilisateur, le script Python est ré-exécuté de haut en bas. Pour conserver l'historique, nous utilisons `st.session_state`, un objet persistant fonctionnant comme un dictionnaire Python. Nous y stockons notamment :

— `chat_history`
— `document_list`
— `vectorstore`

Module 3.2 : Atelier Pratique – Assemblage de l'Interface

Le moteur RAG est prêt, mais il a besoin d'une interface utilisateur. Dans cet atelier, vous allez connecter le backend aux widgets Streamlit.

Objectif

Connecter les fonctions logiques de l'application aux widgets pour créer une application web complète, fonctionnelle et intuitive.

Instructions pour le Code

Partie 1 : La Sidebar dans `main`

1. Compléter les appels à `st.selectbox` pour les modèles (conversation et embeddings), en s'assurant de définir la valeur par défaut.
2. Créer un widget d'upload avec `st.file_uploader` (pour les types PDF, TXT, CSV) et stocker le résultat dans `uploaded_file`.
3. Ajouter une logique de bouton : si `uploaded_file` est non nul et que le bouton est cliqué, obtenir les `chunks`, puis mettre à jour la base vectorielle.

Partie 2 : La page de Chat dans `render_chat_page`

1. Bouton "Nouvelle Conversation" : vider l'historique du chat (`st.session_state.chat_history`) et forcer le rafraîchissement de l'application via la méthode `st.rerun()`.
2. Afficher l'historique via une boucle sur `st.session_state.chat_history` (déjà fait dans le squelette).
3. Utiliser le widget `st.chat_input` pour capturer la nouvelle question de l'utilisateur dans la variable `prompt`.
4. Générer la réponse dans un bloc `with st.chat_message("assistant"):` en utilisant l'appel Streamlit approprié pour afficher le flux de la réponse produite à partir du `prompt`.
5. Afficher les sources (stockées dans `st.session_state.used_chunks`) dans un widget `st.expander`.

Partie 3 : La page Documents dans `render_documents_page`

1. Construire un DataFrame à partir de `doc_data` (déjà fait dans le squelette).
2. Utiliser le widget Streamlit approprié (`st.dataframe`) pour afficher ce DataFrame, en utilisant toute la largeur du conteneur.
3. Implémenter le bouton "Supprimer" pour réinitialiser les variables d'état `document_list` et `vectorstore`, puis forcer le rafraîchissement de l'application.

Module 3.3 : Finalisation, Sécurité et Perspectives

Récapitulatif Sécurité

— **Isolation totale** : tout est exécuté en local grâce à Ollama.
— **Aucun transfert de données** : documents et questions restent sur la machine.
— **Contrôle d'accès physique** : la sécurité dépend de la machine hôte.

Conclusion et Prochaines Étapes

Synthèse des Acquis En trois jours, vous avez construit une application RAG :

— Compréhension du pourquoi et du comment.
— Maîtrise d'une stack moderne (Ollama, LangChain, ChromaDB, Streamlit).
— Sensibilisation aux enjeux de sécurité et d'explicabilité.

Axes d'Amélioration

— Confiance et explicabilité (affichage des sources).
— Optimisation du Retriever : Implémentation de la **Recherche Hybride** combinant l'approche lexicale (comme **BM25**) et la recherche sémantique (dense retrieval utilisant les embeddings) .
— Reranking :Ajout d'une étape de **Reranking** pour améliorer la qualité finale des documents proposés au LLM.
— Amélioration du découpage : Ajustement de la taille et du recouvrement des `chunks` pour chaque cas d'usage .
— Enrichissement des données : Intégration de **métadonnées** pour permettre le filtrage contextuel dans le retriever .
— Tester différents modèles d'embeddings (e.g., MiniLM, MPNet) pour évaluer l'impact sur la pertinence des résultats .
— Évaluer la performance du retriever à l'aide de métriques comme **MRR** .
— Implémentez les stratégies d'optimisation discutées (hybride, reranking, ajustement du chunking) .

Vous détenez désormais une compétence clé : construire des solutions d'IA privées, sécurisées et transparentes.