L'Ecole Nationale d'Electronique et des Télécommunications de Sfax (ENET'Com)

IA Generative

Système RAG Agentique : De la Recherche Syntaxique à l'IA Conversationnelle

---

Élèves :
Mareim M'Bareck
Baba EL Kori

Enseignant :
Nessrine Trabelsi

9 décembre 2025

Table des matières

Résumé Exécutif

Ce projet présente le développement complet d'un système RAG (Retrieval-Augmented Generation) multi-documents avec architecture agentique. Le système permet d'interroger intelligemment un corpus de 224 documents PDF (Journaux Officiels de 2017 à 2025) en langage naturel.

Évolution : Le projet a évolué à travers 4 versions majeures, passant d'une recherche syntaxique simple (v1.0) à un système agentique sophistiqué avec mémoire conversationnelle (v4.2).

Architecture finale : 4 agents spécialisés (Orchestrator, Search, Web Fallback, Synthesis) travaillant en coordination pour fournir des réponses contextuelles précises.

Performance : 40,021 chunks vectorisés, temps de réponse < 10 secondes, confiance moyenne 55-75%, interface Streamlit intuitive.

Accès au projet : Le dataset et le notebook associés à ce projet sont disponibles sur le GitHub suivant : `https://github.com/Baba103/Projet-IA-generative.git`

1 Introduction

1.1 Contexte du Projet

La gestion et l'interrogation efficace de grandes bases documentaires représentent un défi majeur dans de nombreux domaines professionnels. Dans le cadre de ce projet, nous avons travaillé avec un corpus de 224 documents PDF de Journaux Officiels couvrant la période 2017-2025, représentant environ 10,000 pages de contenu juridique.

Les approches traditionnelles de recherche documentaire (Ctrl+F, recherche par mots-clés) présentent des limitations majeures :

- Incapacité à comprendre le sens et le contexte
- Absence de synonymie ("président" ne trouve pas "chef d'État")
- Pas de mémoire entre les requêtes successives
- Interface peu intuitive pour des recherches complexes

1.2 Objectifs

L'objectif principal de ce projet est de développer un système RAG (Retrieval-Augmented Generation) capable de :

1. **Recherche sémantique** : Comprendre le sens des questions, pas seulement les mots-clés
2. **Mémoire conversationnelle** : Maintenir le contexte sur plusieurs échanges
3. **Architecture modulaire** : Agents spécialisés pour différentes tâches
4. **Interface intuitive** : Application web Streamlit accessible
5. **Performance** : Réponses rapides (< 10 secondes) et pertinentes

2 État de l'Art : RAG et Intelligence Artificielle

2.1 Définition du RAG

Le RAG (Retrieval-Augmented Generation) est une architecture d'IA qui combine deux approches complémentaires :

— **Retrieval** : Recherche d'informations pertinentes dans une base de connaissances

— **Generation** : Utilisation d'un LLM pour générer des réponses contextuelles

Cette combinaison permet de surmonter les limitations des chatbots classiques (hallucinations, connaissances limitées) en ancrant les réponses dans des documents réels.

2.2 Technologies Clés Utilisées

2.2.1 Embeddings : Sentence Transformers

Nous utilisons le modèle `all-MiniLM-L6-v2` de Sentence Transformers pour transformer le texte en vecteurs de 384 dimensions. Ce modèle offre un bon compromis entre performance et vitesse :

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer("all-MiniLM-L6-v2")
embedding = model.encode("Votre texte")  # → [384 dim]
```

2.2.2 Base Vectorielle : ChromaDB

ChromaDB est utilisée pour stocker et interroger efficacement les 40,021 vecteurs d'embeddings. Cette base permet des recherches par similarité cosinus en quelques millisecondes.

2.2.3 Chunking Intelligent : Chonkie

Le découpage des documents en chunks de 512 tokens est réalisé avec Chonkie, qui préserve le contexte sémantique contrairement à un découpage fixe.

2.2.4 LLM : Groq (Llama-3.3-70B)

Pour la génération de réponses, nous utilisons l'API Groq avec le modèle Llama-3.3-70B-Versatile, offrant des réponses de qualité avec une latence très faible (500ms).

2.3 Architectures RAG

Trois types d'architectures RAG existent :

1. **RAG Simple** : Recherche → Génération (notre v1.0-v2.0)
2. **RAG Multi-étapes** : Recherche → Re-ranking → Génération (v3.0)
3. **RAG Agentique** : Orchestration multi-agents (v4.2) ← Notre approche finale

3 Développement Itératif : Évolution du Système

3.1 Version 1.0 : RAG Syntaxique (Baseline)

Description : Version initiale avec recherche par mots-clés exacte.
Technologies : Python, `Sentence-Transformers`, `ChromaDB`, `pdfplumber`, `RecursiveCharacterTe`
Limitations :
— Pas de compréhension sémantique
— Sensible aux variations lexicales
— Les résultats de similarité sont souvent faibles, avec les mêmes chunks retournés plusieurs fois
— Incapacité à différencier les segments pertinents de manière contextuelle
Exemple d'échec : Requête "chef d'État" ne trouve pas "président".

3.2 Version 2.0 : RAG Avancé Chunking Sémantique

Innovation majeure : Introduction du modèle Chonkie pour un découpage intelligent du texte, garantissant des chunks sémantiquement cohérents.
Technologies : `unstructured.io`, `Sentence-Transformers`, `Mistral-7B`.
Code clé :

```
# Extraction du texte avec unstructured.io
text = unstructured.io.extract(file_path)

# Chunking sémantique avec Chonkie
chunks = chonkie.chunk(text)

# Embedding des chunks
embeddings = model.encode(chunks)

# Recherche par similarité cosinus
results = db.query(query_emb, top_k=10)
```

Amélioration : Meilleure cohérence contextuelle des chunks (+40% de précision) grâce à l'analyse sémantique du texte.
Limitation : Toujours limité à un seul document pour chaque extraction.

3.3 Version 3.0 : Généralisation (Passage à l'échelle)

Après avoir validé notre pipeline sur un seul document (V1 et V2), nous l'appliquons maintenant à l'intégralité du dataset en utilisant Mistral 7B.
Nouveautés majeures :
— **Pipeline de Batch Processing** : Traitement de masse, itération sur tous les PDFs téléchargés.
— **Chunking intelligent (Chonkie)** : Applique le modèle Chonkie pour découper le texte de manière sémantique.
— **Indexation complète** : Tous les documents sont indexés dans la base de données de connaissances.

Conclusion de la Version 3 : Malgré les avancées techniques (traitement de masse, chunking intelligent), notre RAG reste limité dans sa capacité à répondre à des questions complexes, telles que celle des 'Typologies des établissements humains'. Le système récupère bien des fragments de texte, mais échoue à construire des réponses structurées nécessitant une compréhension profonde et la combinaison de plusieurs sources.

Les temps de traitement augmentent linéairement avec la taille du corpus, ce qui indique qu'un ajustement majeur est nécessaire pour améliorer l'efficacité.

Nécessité d'une évolution majeure : Les limitations identifiées justifient l'introduction de méthodes beaucoup plus avancées dans la prochaine version :

- Utilisation de frameworks d'orchestration cognitive comme `LangChain` ou `LangGraph`.
- Intégration de LLM plus puissants (ex : `Llama-3`) via des API rapides comme `Groq` pour une inférence instantanée.
- Mise en place d'agents autonomes capables de planifier leur recherche pour répondre plus efficacement aux requêtes complexes.

C'est ce que nous explorerons dans la prochaine itération du projet.

3.4 Version 4.0-4.2 : Architecture Agentique

3.4.1 v4.0 : L'Accélération (Groq & Llama-3)

Introduction & Objectif : Dans les versions précédentes, l'utilisation d'un modèle local (Mistral-7B) entraînait une latence très élevée (des minutes par réponse), rendant le système inutilisable en production. L'objectif de la V4.0 est de résoudre ce problème critique en intégrant l'API `Groq`, qui utilise des puces LPU (Language Processing Unit) pour exécuter le modèle `Llama-3 70B` à une vitesse fulgurante.

3.4.2 v4.1 : Mémoire & Contextualisation

Problème : Le chatbot oubliait de quoi on parlait (ex : Q1 "Qui est le ministre ?", Q2 "Et son adjoint ?" → Erreur).

Solution : Implémentation d'une classe `ConversationMemory` (File) pour l'historique et `EntityTracker` (Regex) pour suivre les sujets (Article X, Direction Y). Cette brique est essentielle pour passer d'un simple moteur de recherche à un véritable Agent Conversationnel.

3.4.3 v4.2 : Architecture Multi-Agents

Pour rendre le système robuste et modulaire, nous avons adopté une architecture multi-agents où chaque tâche est gérée par une entité spécialisée.

Les 5 Agents du Système :

- **Orchestrator Agent** : Reçoit la question et coordonne les autres agents.
- **Entity Tracker Agent** : Repère les numéros d'articles et noms de services pour résoudre les "il/elle/le".
- **Search Agent** : Interroge la base vectorielle `ChromaDB`.
- **Web Fallback Agent** : Va sur Google/DuckDuckGo si la base interne ne suffit pas.
- **Synthesis Agent** : Rédige la réponse finale propre.

Avantage de l'architecture : Cette approche permet de diviser le travail en tâches spécialisées, augmentant ainsi la modularité et la robustesse du système, tout en améliorant la gestion du contexte à travers la conversation.

4 Architecture Finale : Système Agentique v4.2

4.1 Vue d'Ensemble

Le système final repose sur une architecture à 5 agents spécialisés travaillant en pipeline :

```
Utilisateur
    ↓
[1. Orchestrator Agent] ← Enrichir contexte + Coréférences
    ↓
[2. Entity Tracker Agent] ← Repère les numéros d'articlese ..etc
    ↓
[3. Search Agent] ← Recherche sémantique (40,021 vecteurs)
    ↓
[3. Web Fallback Agent] ← SI confiance < seil (50% par exemple)%
    ↓
[4. Synthesis Agent] ← Groq LLM (Llama-3.3-70B)
    ↓
Réponse Finale (structurée, citée)
```

4.2 Généralisation - Phase Finale

Après avoir validé notre LLM sur un seul PDF, nous avons passé à la généralisation. Cette fois, nous avons importé toutes les données :

Nous avons monté notre Google Drive et vérifié que le dossier contenant tous les PDFs était accessible. Ensuite, nous avons procédé à l'indexation incrémentale du dataset, traitant 10 PDFs à la fois pour éviter toute surcharge mémoire.

RAG Indexation Incrémentale par Batches - Version 4.2 : Vu le volume important du dataset, nous avons opté pour une **INDEXATION INCRÉMENTALE PAR BATCHES**. Ce processus permet de traiter efficacement les documents tout en assurant une sauvegarde automatique et une reprise après coupure.

Résultat d'indexation : L'indexation a été réussie pour 224 PDFs avec un temps d'exécution bien maîtrisé, et nous avons enregistré des checkpoints à chaque étape pour permettre la reprise de l'indexation en cas d'interruption. La procédure est désormais plus résiliente grâce à cette approche.

RAG Query - Interroger la Base Indexée : Une fois l'indexation terminée, nous avons implémenté un module d'interrogation pour rechercher de manière sémantique à travers tous les documents indexés. Ce module inclut des fonctionnalités avancées comme l'application de filtres (année et source), l'utilisation de la mémoire conversationnelle et la résolution des coréférences dans les réponses.

Mémorisation Conversationnelle : Le chatbot a été amélioré pour mémoriser l'historique des échanges, permettant de maintenir un contexte de conversation entre plusieurs

questions. Cela permet de gérer des scénarios où des informations doivent être rappelées (par exemple, suivre les entités mentionnées comme les articles ou les directions).

Architecture Multi-Agents : Nous avons adopté une architecture multi-agents, avec des agents spécialisés pour chaque tâche :

- **Orchestrator Agent** : Gère l'enrichissement des questions et coordonne les autres agents.
- **Entity Tracker Agent** : Suivi des entités (numéros d'articles, services).
- **Search Agent** : Recherche dans la base vectorielle ChromaDB.
- **Web Fallback Agent** : Recherche sur le web si la base interne ne suffit pas.
- **Synthesis Agent** : Génère la réponse finale.

Problème rencontré : Le processus d'indexation complète sur Colab a rencontré un problème de timeout. Pour le résoudre, nous avons mis en place un système de **checkpoints** permettant de sauvegarder la progression après chaque batch, assurant ainsi une reprise automatique en cas de coupure.

Avantage de la Solution : Grâce à l'indexation incrémentale et aux checkpoints, l'indexation devient résiliente et peut être répartie sur plusieurs sessions courtes (24×25 minutes), ce qui contourne les limitations de Colab.

5 Mise en œuvre de l'Interface et Déploiement

5.1 Justification du Choix Technologique : Streamlit

Le choix de **Streamlit** pour l'interface utilisateur s'est imposé pour sa capacité à transformer rapidement des scripts de Data Science en applications web interactives, entièrement en Python. Contrairement aux frameworks web traditionnels nécessitant des compétences frontend complexes (HTML/CSS/JS), Streamlit permet de se concentrer sur la logique métier du RAG. Sa gestion native de l'état de session (`session_state`) est particulièrement adaptée à notre besoin pour :

- Maintenir l'historique conversationnel (mémoire) entre les interactions.
- Gérer le suivi des entités et du contexte de manière persistante pendant la session.
- Offrir une expérience de chat fluide et réactive indispensable pour un assistant juridique.

5.2 Transition du Notebook vers l'Application

L'application `app_rag_agentique.py` ne fonctionne pas de manière isolée ; elle est l'**aboutissement de production** du travail de recherche effectué dans le notebook.

1. **Développement (Le Notebook) :** Le fichier `Notebook_RAG_Final_Academique.ipynb` sert d'environnement d'expérimentation pour construire, valider les briques logiques (indexation, agents) et surtout générer l'index vectoriel lourd.
2. **Production (L'Application) :** Le script `app_rag_agentique.py` intègre les classes validées (*Orchestrator, SearchAgent, SynthesisAgent*) dans une structure optimisée pour le web.
3. **Le Lien (Le Cache Partagé) :** La connexion technique entre les deux est le dossier `RAG_Cache_Incremental`. Ce dossier contient la base de données vectorielle (ChromaDB) et les métadonnées créées par le notebook. L'application se contente

de charger ce cache pour effectuer des inférences, évitant ainsi un temps de re-calcul prohibitif.

5.3 Prérequis Techniques et Installation

Pour déployer l'application localement, la présence du cache pré-calculé est obligatoire.

5.3.1 1. Gestion du Cache

Le dossier `RAG_Cache_Incremental` doit être impérativement **téléchargé** depuis l'environnement d'entraînement (ex : Google Colab) et placé dans le répertoire local du projet. Il assure la persistance des données vectorielles.

5.3.2 2. Configuration du Code

Avant l'exécution, le fichier `app_rag_agentique.py` nécessite une configuration minimale pour lier le cache et l'API de génération.

[language=Python, caption=Configuration des variables globales] 1. Chemin absolu vers le dossier de cache téléchargé CACHE$_PATH = "C : /Chemin/Vers/Votre/Dossier/RAG_Cache_In$

2. Clé API pour le service Groq (LLM) Remplacez 'xxx' par votre clé personnelle GROQ$_API_KEY = "gsk_xxx...$"

5.4 Exécution Locale

Une fois les dépendances installées et le cache positionné, l'application se lance via le terminal avec la commande suivante :

[language=bash, frame=single, caption=Commande de lancement Streamlit] streamlit run app$_rag_agentique.py$

Cette commande démarre un serveur web local (généralement sur le port 8501) et ouvre automatiquement l'interface de chat dans le navigateur par défaut.

6 Conclusion Générale et Perspectives

Ce projet a permis la mise en œuvre d'un système **RAG Agentique** (Retrieval-Augmented Generation) innovant, capable d'interroger efficacement un corpus juridique complexe grâce à une architecture multi-agents. En passant d'une simple recherche par mots-clés à une compréhension sémantique orchestrée par des agents spécialisés (Orchestrateur, Recherche, Synthèse), nous avons démontré qu'il est possible de réduire significativement les hallucinations des modèles de langage tout en garantissant la traçabilité des sources juridiques.

L'évaluation du prototype actuel, basé sur une sélection de Journaux Officiels, confirme la pertinence de l'approche hybride : le cache persistant assure la performance technique, tandis que la mémoire conversationnelle offre une fluidité d'interaction indispensable à l'usage naturel. Cependant, pour transformer ce prototype académique en un outil d'utilité publique nationale, plusieurs axes d'amélioration et de recommandations stratégiques sont identifiés.

6.1 Extension et Exhaustivité du Corpus

Pour garantir une couverture juridique totale, le système ne doit plus se limiter aux documents récents. Il est impératif de :

— **Intégrer la Constitution Mauritanienne :** Elle constitue la norme suprême et doit servir de référence prioritaire pour l'ancrage des réponses de l'IA.

— **Couverture Historique :** Élargir la base de données pour inclure l'intégralité des archives du Journal Officiel, les codes juridiques en vigueur et la jurisprudence, afin de couvrir tous les aspects du droit mauritanien.

6.2 Accessibilité et Adaptabilité par Profils

L'objectif est de démocratiser l'accès au droit. L'interface doit évoluer pour s'adapter dynamiquement au niveau d'expertise de l'utilisateur :

1. **Pour le Citoyen :** Offrir un mode "Vulgarisation" qui simplifie le jargon juridique, explique les démarches administratives clairement et répond aux questions du quotidien ("Quels sont mes droits si...").
2. **Pour les Juristes et Professionnels :** Proposer un mode "Expert" qui privilégie la citation brute des articles de loi, la précision terminologique et la référence immédiate aux sources (numéro de décret, date de promulgation).
3. **Pour les Étudiants et Chercheurs :** Développer un mode "Pédagogique" capable de faire des liens entre différents textes de loi, d'expliquer l'historique d'une réforme et de servir d'assistant de recherche documentaire avancée.

En somme, ce projet pose les fondations solides d'une LegalTech souveraine, capable de rapprocher le justiciable de ses droits par la puissance de l'Intelligence Artificielle.

6.3 Stack Technologique Complet

TABLE 1 – Technologies utilisées par composant

Composant	**Technologie**
Backend	Python 3.11
Vector DB	ChromaDB (PersistentClient)
Embeddings	sentence-transformers (all-MiniLM-L6-v2)
Chunking	Chonkie (SemanticChunker)
Extraction PDF	Unstructured (PDFPartitioner)
LLM API	Groq (llama-3.3-70b-versatile)
Frontend	Streamlit 1.51
Stockage Cache	Google Drive (500 MB)
Web Scraping	BeautifulSoup4 + Requests