



















Chatbot Cinématographique basé sur Neo4j en utilisant Python

CineBot  

Présentation 	2
Vue d'ensemble 	3
I. La base de données 	4
A. Conception de la base 	4
B. Mise en place 	5
C. Exploitation 	5
II. L'Intelligence Artificielle 	6
A. Choix du modèle LLM 	6
B. Interaction 	6
III. Le chatbot 	7
A. Fonctionnalités 	7
1. Cypher QA 	7
2. Vector Search Index 	8
B. L'agent 	9
C. L'interface 	10
Conclusion 	11
Liens 	12
Certifications 	13

Présentation

Dans le cadre de ce projet, nous avons pour but de réaliser un chatbot de recommandation de films à l'aide d'une base de données de graphe Neo4j, programmé en Python avec le modèle GPT 3.5 Turbo d'OpenAI.

L'interface est simple et semblable à un chat gpt, l'utilisateur y est capable de taper des questions en rapport avec les films et il recevra des réponses intelligibles et compréhensibles pour un humain moyen.

Ainsi, notre chatbot sera capable de répondre à des questions formulées en anglais en rapport avec les films et plus particulièrement avec la base de données que nous lui fournissons.

Vue d'ensemble🔍

Ce projet comprend donc différents composants que nous allons rapidement énumérer.

D'abord il repose sur la base de données de graphe **Neo4j**, cette base comprend des films, acteurs, directeurs et spectateurs liés par des relations que nous détaillerons d'autant plus dans la première partie. Cette base a pour unique but de fournir les données qui servent au chatbot de répondre aux requêtes utilisateurs.

Ensuite, un autre élément important dans ce projet est l'utilisation d'un **Large Language Model (LLM)**, indispensable pour comprendre le langage de l'utilisateur et le transformer en requête vers notre base de données et retranscrire le résultat dans un format lisible pour un utilisateur moyen. Dans notre cas, nous utilisons le modèle GPT 3.5 Turbo d'OpenAI mais n'importe quel autre modèle est utilisable.

Le projet est écrit en **Python**, c'est le langage de programmation le plus performant dans le domaine de l'IA et il est aussi pratique pour le développement rapide avec une conception épurée. Nous utilisons principalement 2 bibliothèques:

- **Streamlit** qui nous permet de générer l'interface web de type chat gpt à partir des données de notre chatbot (requêtes utilisateur, réponses du bot)
- **Langchain** qui nous permet d'interagir avec notre LLM et Neo4j à travers différents outils, fournissant une base solide pour le développement d'applications d'IA et est livré avec des intégrations Neo4j pour Cypher et Vector Indexes.

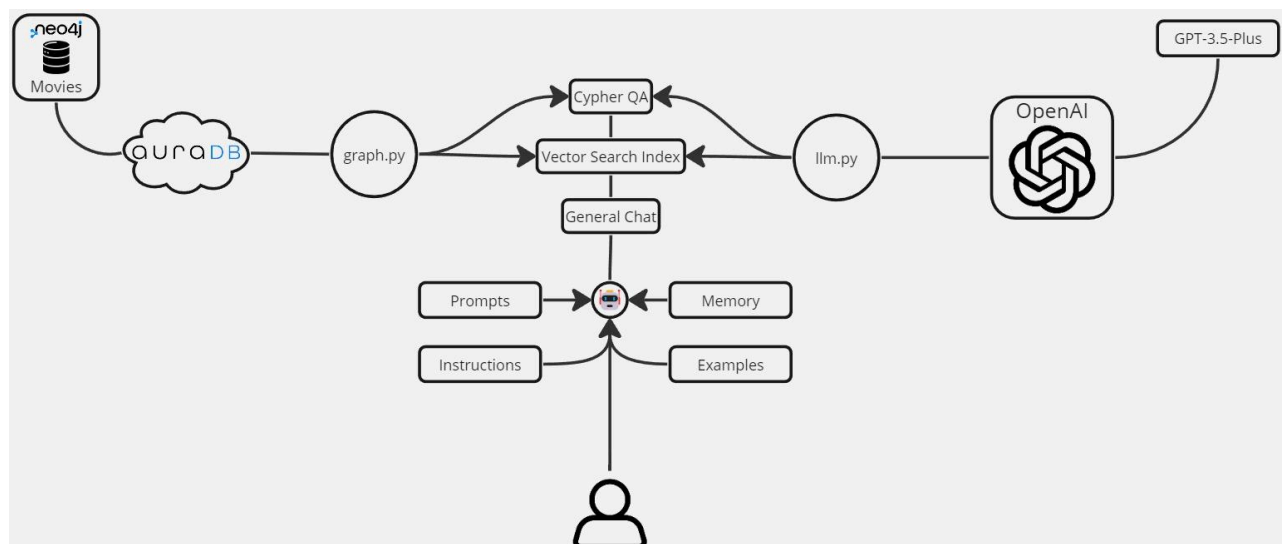


Schéma détaillant comment notre projet est structuré

I. La base de données 🗄️

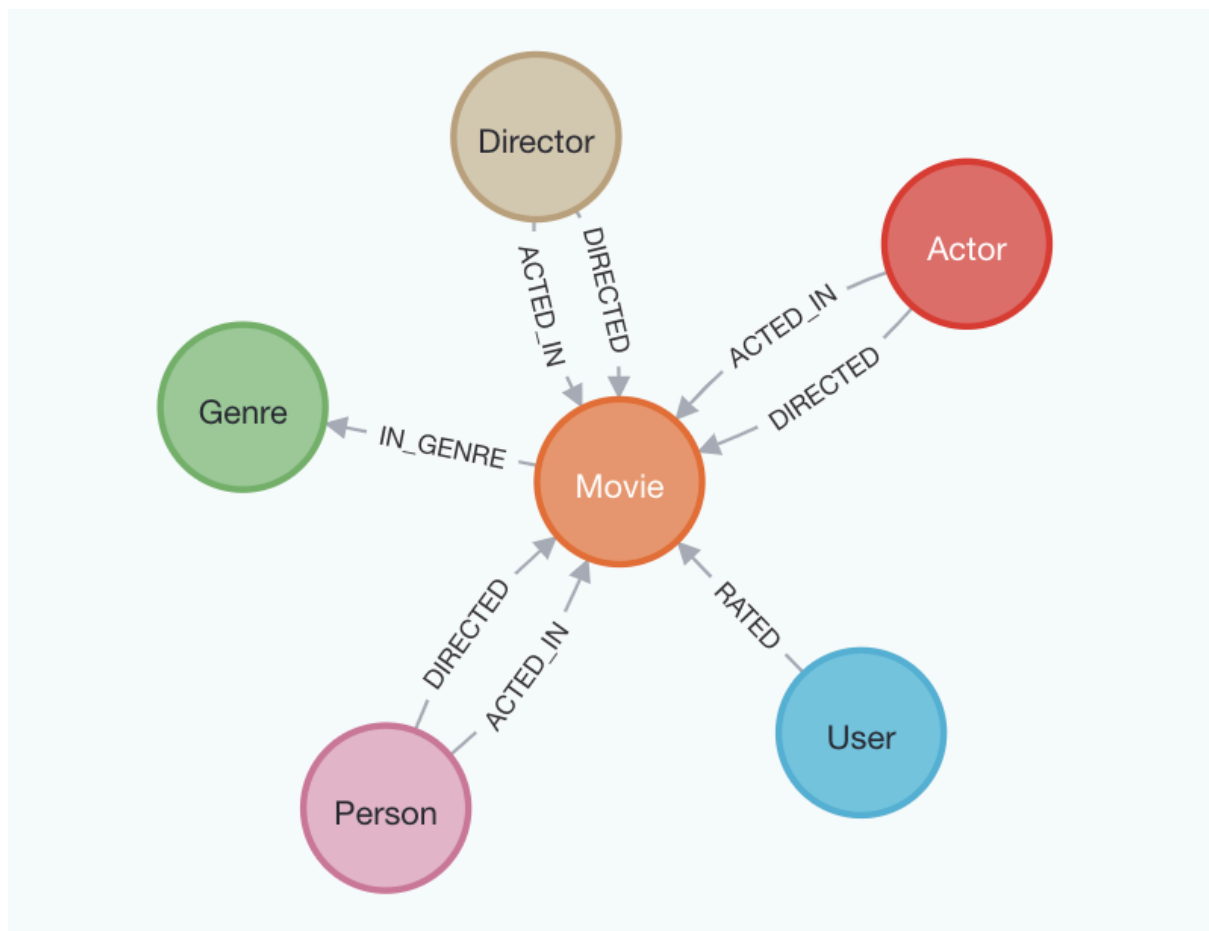
Dans un premier temps, détaillons notre base de données de graphe Neo4j, de sa conception à partir d'un dataset, à sa mise en place dans le web jusqu'à son exploitation dans notre chatbot.

A. Conception de la base 🛠️

Notre base de données est issue du dataset [Recommendations](#) accessible sur github, il comprend 28863 Noeuds (Nodes) et 166261 Relations (Relationships) parmi lesquels on comprend Person(19047), Actor(15443), Movie (9125), Director(4091), User(671), Genre(20).

Toutes les données sont générées par la base de données officielle Internet Movie Database (IMDb), ça veut dire que toutes les données sur les acteurs, directeurs et films sont exactes et très complètes (notation, synopsis, date de sortie, titre, genre etc...)

L'ensemble des relations sont **ACTED_IN**, **DIRECTED**, **RATED**, **IN_GENRE** sont définies comme ceci:

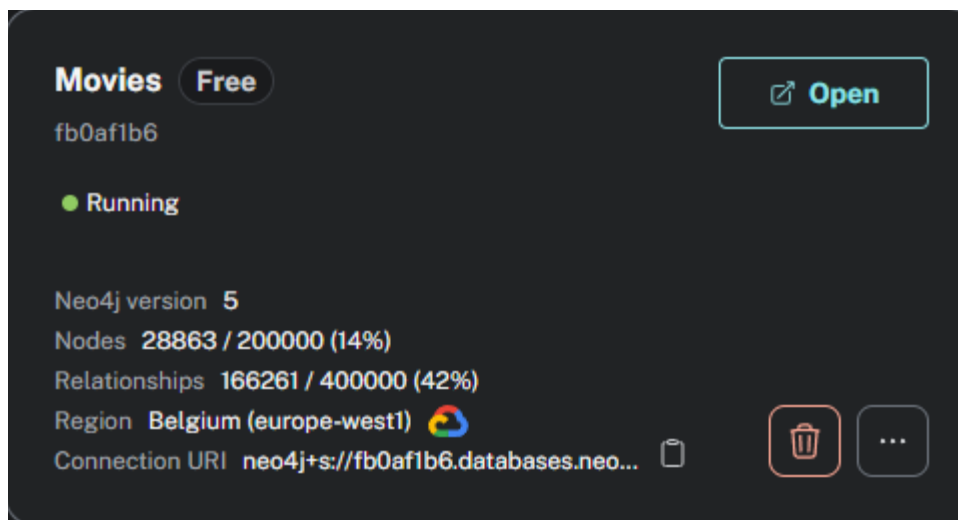


B. Mise en place 🏗️

Nous avons choisi d'utiliser le service cloud Neo4j AuraDB pour la création de notre base de données, ainsi elle est accessible facilement et rapidement de n'importe quelle machine à condition d'utiliser ses identifiants.

On a créé une instance et importé le fichier .dump de notre dataset après avoir sauvegardé les identifiants de connexion à notre base de données.

Ensuite, afin de recommander des films similaires basée sur le contenu narratif de leur résumé on a converti les résumés de films en "embeddings" ou représentations vectorielles ce sont des représentations numériques de texte qui capturent la sémantique et le sens des phrases pour mesurer la similarité entre des documents textuels. Puis on a ajouté ces données à chaque film de la base cela nous servira pour les requêtes basées sur les résumés de films.



L'instance Movies hébergée sur AuraDB

C. Exploitation 🧳

La base de données est créée et prête à être exploitée, alors dans notre application on a créé une instance globale de Neo4jGraph responsable de la connexion à la base de données, de l'envoi des requêtes et la réception des données à l'aide des identifiants de connexions vers la base sur AuraDB.

```
graph = Neo4jGraph(  
    url=st.secrets["NEO4J_URI"],  
    username=st.secrets["NEO4J_USERNAME"],  
    password=st.secrets["NEO4J_PASSWORD"],  
)
```

II. L'Intelligence Artificielle

A présent, nous allons décrire notre utilisation de l'intelligence artificielle dans ce projet. On rappelle que son rôle est de répondre aux requêtes des utilisateurs concernant les films. Nous verrons comment nous avons choisi un modèle et la façon dont on interagit avec.

A. Choix du modèle LLM





Un Large Language Model (LLM) est une IA capable de comprendre et générer du langage naturel, pour permettre des interactions conversationnelles fluides et compréhensibles, dans notre cas en interprétant les requêtes des utilisateurs concernant les films et les convertir en requête Cypher et en fournissant des réponses contextuellement appropriées.

Nous avons choisi un modèle d'OpenAI pour leur simplicité d'usage, leur disponibilité dans le cloud et leur vitesse de traitement.

B. Interaction

Pour interagir avec un modèle d'OpenAI on vient d'abord générer une clé API qu'on utilise ensuite dans notre application, de la même manière que pour la base de données on utilise une instance globale de ChatOpenAI et un autre de OpenAIEmbeddings avec en paramètre notre clé api et le nom du modèle à utiliser (gpt-3.5-turbo). La première instance est utilisée pour les fonctionnalités générales de chat avec le modèle et la deuxième pour l'utilisation des embeddings générés précédemment.

```
llm = ChatOpenAI(  
    openai_api_key=st.secrets["OPENAI_API_KEY"],  
    model=st.secrets["OPENAI_MODEL"],  
)  
embeddings = OpenAIEmbeddings(  
    openai_api_key=st.secrets["OPENAI_API_KEY"]  
)
```

API keys						
Your secret API keys are listed below. Please note that we do not display your secret API keys again after you generate them.						
Do not share your API key with others, or expose it in the browser or other client-side code. In order to protect the security of your account, OpenAI may also automatically disable any API key that we've found has leaked publicly.						
Enable tracking to see usage per API key on the Usage page .						
NAME	SECRET KEY	TRACKING 	CREATED	LAST USED 	PERMISSIONS	
Neo4j Chatbot	sk-...8qyr	Enabled	4 mars 2024	7 mars 2024	All	 
+ Create new secret key						

Clé api pour l'utilisation d'un modèle OpenAI

III. Le chatbot

Nous avons passé en revue les éléments de base de notre projet, la base de données Neo4j et l'intelligence artificielle, il est temps de voir comment nous avons mis en œuvre le chatbot pour exploiter ces éléments et créer une expérience intéressante et interactive.

A. Fonctionnalités

Si nous reprenons le schéma de notre projet, on peut voir que notre agent utilise une chaîne d'outils (Tools) pour choisir la meilleure façon de répondre aux requêtes utilisateur. Chacun de ces outils est indispensable et a un rôle bien précis à jouer, **Cypher QA** pour traiter les requêtes en utilisant directement les données de la base de données avec des requêtes Cypher, **Vector Search Index** pour traiter les informations à de l'intrigue des film en utilisant leur représentations vectorielles et enfin **General Chat** pour toutes les requêtes n'étant pas couvertes par les autres outils qui se résume simplement appeler la fonction de base de l'instance llm sans contexte particulier.

1. Cypher QA ?

Nous avons mis en place notre approche en utilisant la chaîne GraphCypherQACHain de langchain. Cette chaîne intègre l'utilisation de LLM et des bases de données Neo4j pour permettre leur interaction. En effet, GraphCypherQACHain nous permet d'utiliser des requêtes Cypher générées à partir de modèles qu'on peut lui fournir, on lui fournit un template. Ce template offre des directives claires au bot qui devient alors un expert développeur Neo4j, lui demandant de traduire les questions des utilisateurs en requêtes Cypher pour répondre aux questions sur les films et fournir des recommandations.

C'est ainsi que notre chatbot tire parti de cette approche et utilise GraphCypherQACHain pour interpréter les questions des utilisateurs, générer des requêtes Cypher correspondantes, les exécuter sur la base de données Neo4j et présenter des réponses contextualisées. Cela permet une interaction plus profonde et personnalisée avec les utilisateurs, car le chatbot peut comprendre les nuances des questions liées aux films et répondre de manière pertinente grâce à la puissance de Neo4j. Cette combinaison de technologies renforce l'expertise de notre chatbot dans le domaine cinématographique.

```
cypher_prompt = PromptTemplate.from_template(CYPHER_TEMPLATE)
cypher_qa = GraphCypherQACHain.from_llm(
    llm,
    graph=graph,
    verbose=True,
    cypher_prompt=cypher_prompt
)
```

2. Vector Search Index

Nous avons implémenté le Vector Search Index pour enrichir les informations sur les films en se basant sur leurs intrigues. Cette approche utilise Neo4jVector, une classe de langchain_community, pour intégrer des embeddings des intrigues générées précédemment dans notre base de données Neo4j.

Le processus commence par la création d'une instance de Neo4jVector à partir d'un index existant, spécifiant des détails tels que l'URL de la base de données Neo4j, le nom de l'index, la propriété du nœud texte et la propriété d'embedding, comme indiqué dans notre exemple. Ce Vector Search Index est ensuite utilisé comme un "retriever" dans la chaîne de traitement RetrievalQA. Cette chaîne, combinée avec notre modèle de langage large (LLM), permet au chatbot d'interpréter les questions des utilisateurs, d'utiliser le Vector Search Index pour récupérer des informations contextuelles sur les films basées sur leurs intrigues, et de présenter des réponses complètes. Cette approche augmente significativement la richesse des informations disponibles, offrant ainsi une expérience utilisateur plus approfondie et engageante dans le domaine cinématographique.

```
neo4jvector = Neo4jVector.from_existing_index(  
    embeddings,  
    url=st.secrets["NEO4J_URI"],  
    username=st.secrets["NEO4J_USERNAME"],  
    password=st.secrets["NEO4J_PASSWORD"],  
    index_name="moviePlots",  
    node_label="Movie",  
    text_node_property="plot",  
    embedding_node_property="plotEmbedding",  
    retrieval_query=RETRIEVAL_QUERY  
)  
retriever = neo4jvector.as_retriever()  
kg_qa = RetrievalQA.from_chain_type(  
    llm,  
    chain_type="stuff",  
    retriever=retriever,  
)
```


B. L'agent

Notre fichier `agent.py` incarne l'architecture de notre chatbot, exploitant tous les outils précédemment présentés. L'agent utilise deux chaînes principales pour répondre aux questions des utilisateurs : CypherQA pour fournir des informations spécifiques sur les films en utilisant des requêtes Cypher dans Neo4j, et Vector Search Index pour enrichir les réponses en se basant sur les embeddings des parcelles des films.

```
tools = [
    Tool.from_function(
        name="General Chat",
        description="For general chat not covered by other tools",
        func=llm.invoke,
        return_direct=False,
        handle_parsing_errors=True
    ),
    Tool.from_function(
        name="Cypher QA",
        description="Provide information about movies questions using Cypher",
        func=cypher_qa,
        return_direct=False,
        handle_parsing_errors=True
    ),
    Tool.from_function(
        name="Vector Search Index",
        description="Provides information about movie plots using Vector Search",
        func=kg_qa,
        return_direct=False,
        handle_parsing_errors=True
    )
]
```

Les outils, définis dans le fichier, encapsulent les fonctionnalités spécifiques de ces chaînes, et l'agent est configuré pour réagir de manière réactive à l'input utilisateur, en utilisant une fenêtre de mémoire pour conserver un historique de la conversation.

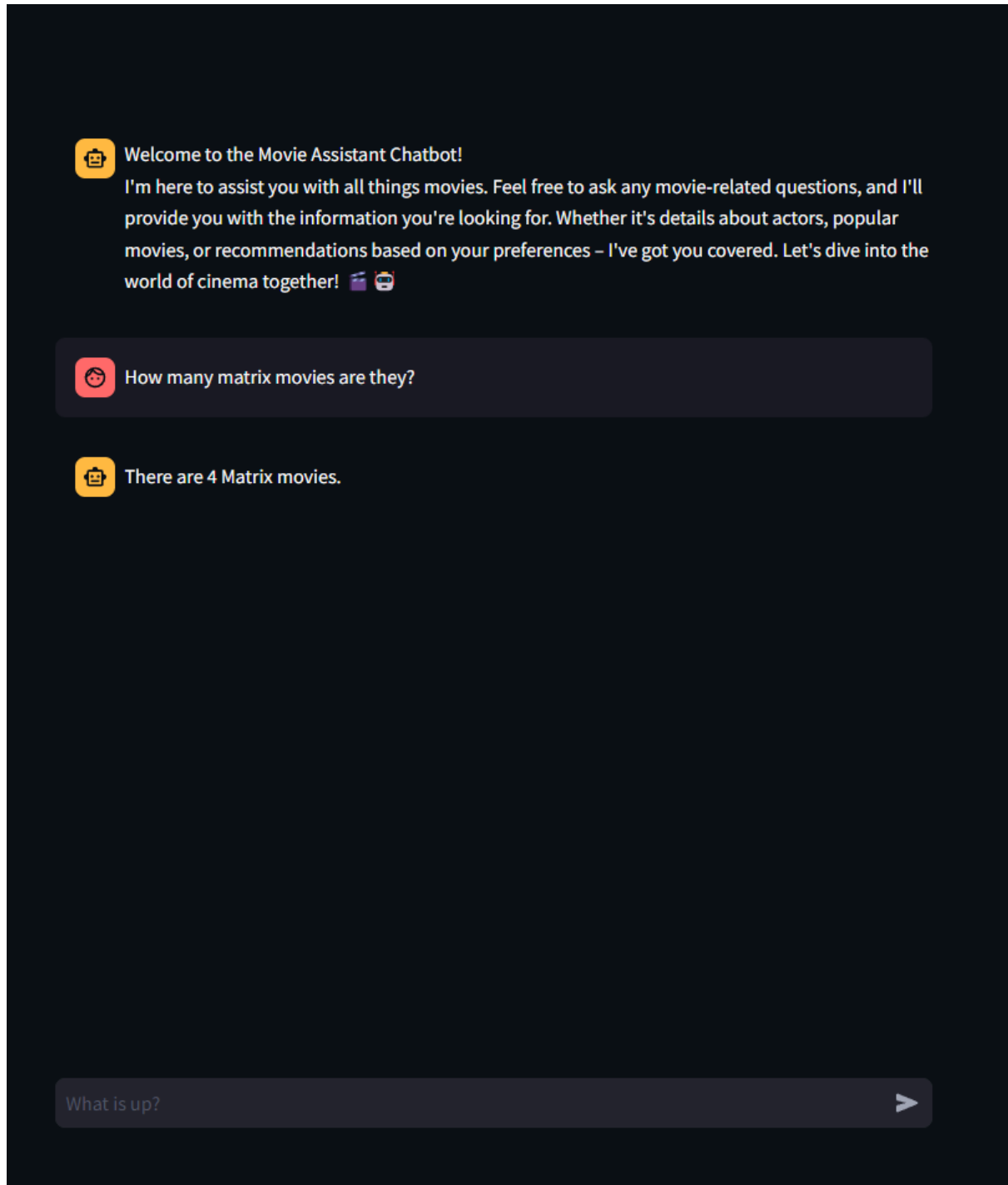
```
memory = ConversationBufferWindowMemory(
    memory_key='chat_history',
    k=5,
    return_messages=True,
)
```

Le prompt structuré fournit à l'agent guide son comportement en définissant les formats attendus pour les interactions avec les outils et les réponses finales.

```
agent_prompt = PromptTemplate.from_template(AGENT_PROMPT_TEMPLATE)
```

C. L'interface

En ce qui concerne l'interface, elle est développée avec Streamlit, offrant une expérience utilisateur conviviale et interactive. Les messages de la conversation sont gérés dans un état de session, et l'utilisateur peut interagir avec le chatbot en soumettant des messages. La réponse générée par le chatbot est affichée en temps réel dans l'interface utilisateur, créant ainsi une expérience de conversation fluide et immersive.



Interface de notre CineBot

Conclusion

En conclusion, notre chatbot basé sur Neo4j et utilisant le modèle GPT 3.5 Turbo d'OpenAI offre une solution complète pour répondre aux requêtes des utilisateurs concernant les films. En combinant une base de données de graphe Neo4j riche en données cinématographiques avec l'intelligence artificielle du modèle de langage large, notre chatbot offre une expérience utilisateur immersive et interactive.

La conception de la base de données, alimentée par le dataset Recommendations, fournit des informations précises et complètes sur les films, acteurs, réalisateurs et genres. La mise en place de la base de données sur Neo4j AuraDB offre une accessibilité facile et rapide, et l'utilisation d'embeddings pour les résumés de films enrichit encore les capacités de recherche du chatbot.

Le choix du modèle GPT 3.5 Turbo d'OpenAI pour l'intelligence artificielle se révèle efficace, offrant une compréhension approfondie du langage naturel et la capacité de générer des réponses contextuelles. L'interaction entre le modèle, la base de données Neo4j et les outils spécialisés tels que Cypher QA et Vector Search Index permet au chatbot de fournir des réponses riches en contexte et adaptées aux besoins spécifiques des utilisateurs.

La mise en œuvre du chatbot à travers l'agent, avec ses outils spécifiques et son interface conviviale développée avec Streamlit, offre une expérience de conversation fluide et engageante. La modularité de l'architecture permet d'ajouter de nouvelles fonctionnalités facilement, garantissant l'évolutivité du chatbot pour répondre aux futurs besoins.

Liens

Vous trouverez le code source de notre projet sur notre adresse github:

<https://github.com/AntoninJuquel/neo4j-chatbot-esiee>

Vous pouvez tester notre chatbot en ligne à l'adresse suivante:

<https://neo4j-chatbot-esiee.streamlit.app>

Vous trouverez le lien vers notre vidéo de présentation du chatbot à l'adresse suivante:

<https://youtu.be/p1mVPilo9pY>

Certifications

Antonin Juquel:

<https://graphacademy.neo4j.com/c/333b4f4b-dd9d-47eb-b24a-d15ea73e7768/>



Raj Porus Hiruthayaraj:

<https://graphacademy.neo4j.com/c/2c444be6-1613-4cc2-ad89-2fc7f8f4b8a1/>

