

Création d'histoires de jeux pédagogiques



Auteur:

SADIK Hajar

ESSALHI Sara

KAISSI Houda

Encadré par: Profe.Lotfi EL AACHAK

Contents

0.1	Introduction	2
0.1.1	Contexte et Motivation	2
0.1.2	Objectif du Projet	2
0.1.3	Motivation	3
0.1.4	Revue de la Littérature	3
0.1.5	Portée et Limitations	3
0.2	Outils et Technologies	3
0.2.1	Backend	3
0.2.2	Frontend	6
0.2.3	MLOps	7
0.3	Fine Tuning LLama2	7
0.3.1	Prérequis	8
0.3.2	Étapes	8
0.4	RAG	14
0.4.1	Introduction	15
0.4.2	Configuration de l'Environnement	15
0.4.3	Chargement et Séparation des Documents	16
0.4.4	Embedding Text par Langchain	17
0.4.5	Création du Vector Store par Chroma DB	18
0.4.6	Persistance dans Chroma DB	18
0.4.7	Utiliser le modèle Mistral (LLM) avec Chroma DB	19
0.4.8	Création du Prompt Template	20
0.4.9	Extraction des Réponses des Documents	20
0.5	Déploiement	21
0.5.1	Fichier Docker	21
0.5.2	Explication du Fichier Docker	22
0.6	Interfaces de l'Application et Description	23
0.6.1	Description de l'Application Chatbot :	23
0.6.2	Fonctionnalités Principales	23
0.6.3	Interfaces de l'Application	24

0.7	Conclusion	27
0.8	Bibliographie	27

0.1 Introduction

0.1.1 Contexte et Motivation

L'essor des technologies de l'intelligence artificielle (IA) et du traitement automatique des langues (TAL) a ouvert de nouvelles perspectives dans divers domaines, y compris celui des jeux vidéo. La génération automatique de contenu, en particulier d'histoires narratives pour les jeux vidéo, représente un défi majeur et une opportunité innovante. Ce projet vise à développer un chatbot intelligent capable de générer des histoires de jeux vidéo en français, utilisant les technologies avancées de *Retrieval Augmented Generation* (RAG), LangChain, et les bases de données vectorielles.

La génération d'histoires de jeux vidéo n'est pas seulement une tâche technique, mais également un processus créatif qui nécessite une compréhension profonde des structures narratives et des nuances linguistiques. En combinant les capacités de RAG, LangChain, et les bases de données vectorielles, ce projet cherche à créer un outil puissant pour les développeurs de jeux, leur permettant de générer des récits immersifs et interactifs de manière efficace.

0.1.2 Objectif du Projet

L'objectif principal de ce projet est de concevoir et de mettre en œuvre un chatbot qui peut générer des histoires de jeux vidéo en français. Ce chatbot sera basé sur une combinaison de technologies de pointe :

- **Retrieval Augmented Generation (RAG)** : Cette technique combine les approches de récupération d'informations et de génération de texte pour produire des réponses plus pertinentes et contextuellement appropriées.
- **LangChain** : Une infrastructure permettant de chaîner différents modules de traitement de la langue naturelle pour créer des applications linguistiques complexes.
- **Bases de Données Vectorielles** : Utilisées pour stocker et récupérer efficacement des vecteurs de mots, facilitant ainsi l'intégration de données contextuelles et sémantiques dans le processus de génération.

0.1.3 Motivation

La motivation derrière ce projet repose sur plusieurs aspects :

- **Innovation dans la Génération de Contenu** : Les histoires de jeux vidéo sont cruciales pour l'expérience utilisateur. La possibilité de générer automatiquement des récits cohérents et captivants peut révolutionner le développement de jeux.
- **Technologies de Pointe** : L'intégration de RAG, LangChain, et des bases de données vectorielles représente une convergence des technologies les plus avancées en matière de TAL, offrant des performances et une flexibilité accrues.
- **Adaptation Linguistique et Culturelle** : En se concentrant sur le français, ce projet met en avant l'importance des spécificités linguistiques et culturelles dans le développement de technologies IA, souvent dominées par des modèles en anglais.

0.1.4 Revue de la Littérature

Une brève revue de la littérature montre que les chatbots actuels, bien que performants dans certains contextes, présentent des limitations lorsqu'il s'agit de générer des contenus narratifs complexes, surtout en français. Les travaux existants se concentrent principalement sur des applications de conversation générale ou de support client, laissant un espace important pour l'innovation dans la génération d'histoires de jeux vidéo.

0.1.5 Portée et Limitations

Le projet se concentre sur la génération d'histoires pour des jeux vidéo en langue française, en utilisant des corpus spécifiques pour l'entraînement et le fine-tuning des modèles de langue. Les limitations incluent la gestion des variations dialectales du français et la complexité de la création de récits véritablement immersifs qui nécessitent une compréhension approfondie des arcs narratifs et des personnages.

0.2 Outils et Technologies

Dans cette section, nous présentons les différents outils et technologies utilisés pour le développement du chatbot, en détaillant les aspects backend, frontend et MLOps.

0.2.1 Backend

Pour le développement du backend, nous avons utilisé une combinaison de technologies avancées pour assurer la performance, la scalabilité et l'efficacité du système.

LangChain



LangChain est une infrastructure permettant de chaîner différents modules de traitement de la langue naturelle pour créer des applications linguistiques complexes. LangChain facilite l'intégration de différents modèles et techniques de TAL pour une génération de texte plus cohérente et pertinente.

Chroma



Pour la gestion des données vectorielles, nous avons utilisé **Chroma**, une base de données vectorielle. Chroma permet de stocker et de récupérer efficacement des vecteurs de mots, ce qui est crucial pour l'intégration de données contextuelles et sémantiques dans le processus de génération de texte.

Mistral

Nous avons également travaillé avec **Mistral**, une technologie de génération de texte qui permet d'améliorer la qualité des réponses générées en utilisant des modèles de langage avancés. Mistral est utilisé pour affiner et enrichir les résultats fournis par le modèle principal.



LLaMA2



Pour le fine-tuning des modèles de langue, nous avons utilisé **LLaMA2**. LLaMA2 est un modèle de langage développé pour être finement ajusté sur des corpus spécifiques, ce qui permet d'adapter le chatbot aux nuances et aux particularités du français.

Google Colab et Hugging Face



HUGGING FACE



FastAPI



Nous avons choisi **FastAPI** comme framework principal pour le développement du backend. FastAPI est un framework web moderne et rapide, basé sur Python, qui permet de créer des APIs de manière simple et efficace. Ses principales caractéristiques incluent :

- **Performance** : Comparable à NodeJS et Go, grâce à l'utilisation de Starlette et Pydantic.
- **Facilité de Développement** : Documentation automatique avec Swagger et ReDoc.
- **Asynchronisme** : Supporte les opérations asynchrones, ce qui améliore la gestion des requêtes concurrentes.

Nous avons intégré des modèles de **OpenAI** et **Hugging Face** pour fournir des capacités avancées de traitement du langage naturel. OpenAI offre des modèles performants comme GPT-3, tandis que Hugging Face fournit une bibliothèque et une infrastructure pour le déploiement et le fine-tuning des modèles.

0.2.2 Frontend

Pour le développement du frontend, nous avons choisi **Angular**, un framework populaire pour les applications web.

Angular



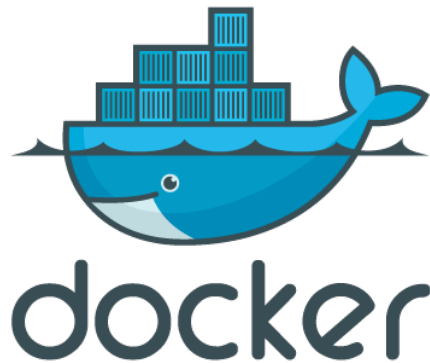
Angular est un framework utilisé pour créer des **Single Page Applications (SPA)**, offrant une expérience utilisateur fluide et réactive. Les principales raisons de notre choix incluent :

- **Modularité** : Architecture modulaire facilitant la maintenance et la scalabilité.
- **Performance** : Chargement rapide des pages et interactivité améliorée.
- **Outils de Développement** : Écosystème riche avec des outils comme Angular CLI pour la gestion des projets.

0.2.3 MLOps

Pour assurer une intégration continue et un déploiement efficace de nos modèles, nous avons utilisé des outils MLOps, notamment **Docker** et **Kubernetes**.

Docker



Docker est utilisé pour containeriser nos applications, assurant ainsi la portabilité et la consistance des environnements de développement, de test et de production. Les avantages de Docker incluent :

- **Isolation** : Chaque application tourne dans son propre conteneur, évitant les conflits de dépendances.
- **Portabilité** : Les conteneurs peuvent être déployés sur n'importe quel système supportant Docker.
- **Scalabilité** : Facilite le déploiement d'applications à grande échelle.

Ces outils et technologies sont essentiels pour la réalisation de notre projet, assurant une efficacité, une scalabilité et une robustesse accrues tout au long du cycle de développement et de déploiement.

0.3 Fine Tuning LLama2

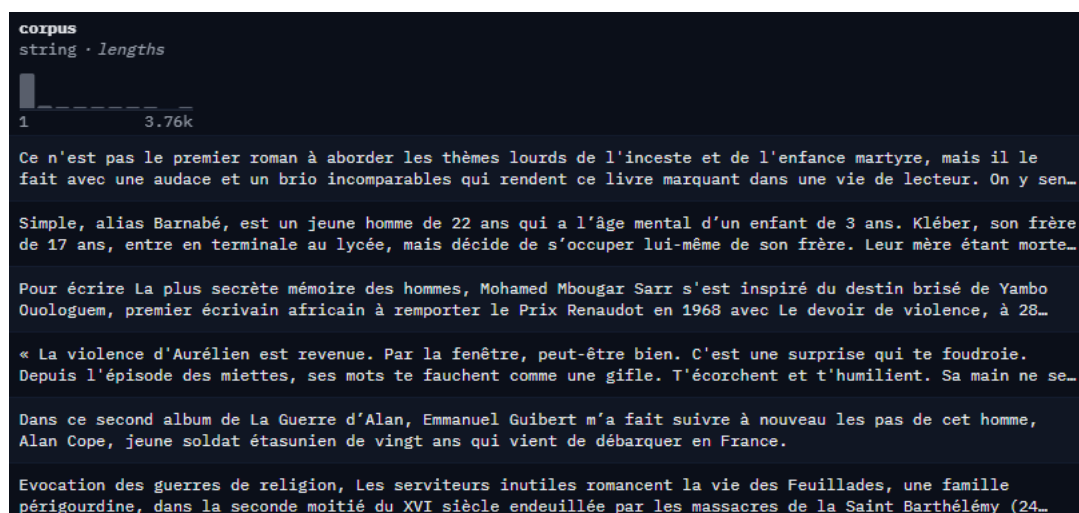
Pour commencer le finetuning de LLama2 sur un corpus français, nous utiliserons Google Colab en tirant parti de ses ressources de calcul. Voici un guide étape par étape pour ce processus.

0.3.1 Prérequis

- Un compte Google pour accéder à Google Colab
- Un Corpus Français
- HuggingFace Compte

0.3.2 Étapes

- **Configuration de l'Environnement** : Ouvrez Google Colab et créez un nouveau notebook. Ajouter HuggingFace API Token
- **Préparation de corpus** : Des données de haute qualité sont fondamentales pour produire un bon modèle. Plus la qualité des données est élevée, meilleur sera le modèle.



```
corpus
string · lengths
1 3.76k
```

Ce n'est pas le premier roman à aborder les thèmes lourds de l'inceste et de l'enfance martyre, mais il le fait avec une audace et un brio incomparables qui rendent ce livre marquant dans une vie de lecteur. On y sen...

Simple, alias Barnabé, est un jeune homme de 22 ans qui a l'âge mental d'un enfant de 3 ans. Kléber, son frère de 17 ans, entre en terminale au lycée, mais décide de s'occuper lui-même de son frère. Leur mère étant morte...

Pour écrire La plus secrète mémoire des hommes, Mohamed Mbougar Sarr s'est inspiré du destin brisé de Yambo Ouologuem, premier écrivain africain à remporter le Prix Renaudot en 1968 avec Le devoir de violence, à 28...

« La violence d'Aurélien est revenue. Par la fenêtre, peut-être bien. C'est une surprise qui te foudroie. Depuis l'épisode des miettes, ses mots te fauchent comme une giflle. T'écorchent et t'humilient. Sa main ne se...

Dans ce second album de La Guerre d'Alan, Emmanuel Guibert m'a fait suivre à nouveau les pas de cet homme, Alan Cope, jeune soldat étasunien de vingt ans qui vient de débarquer en France.

Evocation des guerres de religion, Les serviteurs inutiles romancent la vie des Feullades, une famille périgourdine, dans la seconde moitié du XVI siècle endeuillée par les massacres de la Saint Barthélémy (24...

- **Tokenisation** : nous utilisons la bibliothèque Transformers de Hugging Face pour effectuer cette tâche. En utilisant la méthode AutoTokenizer, nous chargeons un tokenizer pré-entraîné spécifique, en l'occurrence **NousResearch/Llama-2-7b-hf**, qui est adapté à notre tâche de traitement du langage naturel. Une fois le tokenizer chargé, chaque exemple de notre jeu de données est tokenisé à l'aide de la méthode tokenizer.tokenize. Cette étape convertit chaque exemple en une séquence de tokens, permettant ainsi au modèle de les traiter efficacement. .

```
def deduplicate_dataset(dataset: Dataset, model: str, threshold: float):
    sentence_model = SentenceTransformer(model)
    outputs = dataset['train']['corpus'] # Accessing the 'corpus' feature within the 'train' split

    print("Converting text to embeddings...")
    embeddings = sentence_model.encode(outputs, show_progress_bar=True)
    dimension = embeddings.shape[1]
    index = faiss.IndexFlatIP(dimension)
    normalized_embeddings = embeddings / np.linalg.norm(embeddings, axis=1, keepdims=True)
    index.add(normalized_embeddings)

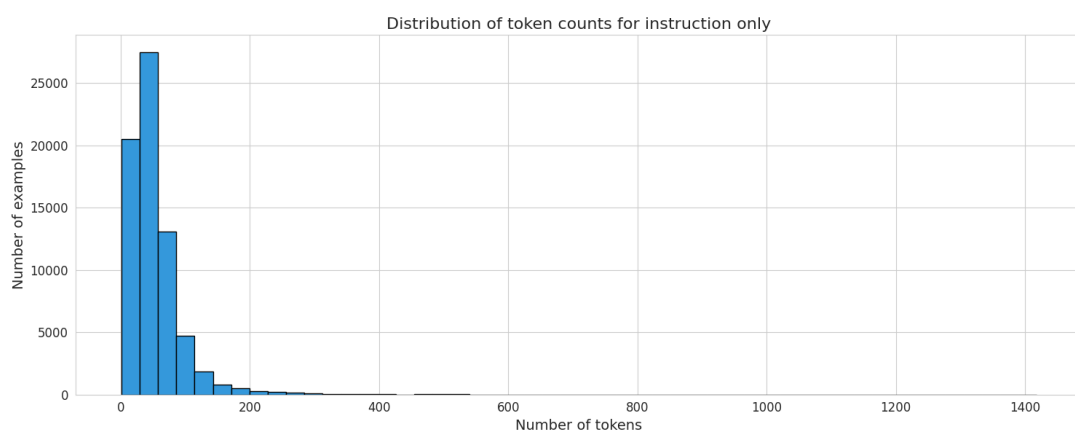
    print("Filtering out near-duplicates...")
    D, I = index.search(normalized_embeddings, k=2)
    to_keep = []

    for i in tqdm(range(len(embeddings)), desc="Filtering"):
        # If the second closest vector (D[i, 1]) has cosine similarity above the threshold
        if D[i, 1] >= threshold:
            # Check if either the current item or its nearest neighbor is already in the to_keep list
            nearest_neighbor = I[i, 1]
            if i not in to_keep and nearest_neighbor not in to_keep:
                # If not, add the current item to the list
                to_keep.append(i)
        else:
            # If the similarity is below the threshold, always keep the current item
            to_keep.append(i)

    deduplicated_dataset = dataset['train'].select(to_keep) # Selecting examples from the 'train' split
    return DatasetDict({"train": deduplicated_dataset})

# Assuming deduped_dataset is already defined
deduped_dataset = deduplicate_dataset(dataset, "thenlper/gte-large", 0.95)
```

Pour évaluer la distribution des comptes de tokens dans un jeu de données d'instructions, nous devons analyser la longueur des instructions. Cette analyse permet de mieux comprendre la structure de notre jeu de données et d'ajuster les hyperparamètres du modèle en conséquence



- **Fine-tune Llama 2 dans Google Colab** : Il y'a trois options pour l'affinage supervisé, l'affinage complet LoRA, et QLoRA. Dans cette section, nous allons affiner un modèle Llama2 avec **7 milliards de paramètres** sur un GPU T4 avec une grande RAM en utilisant Google Colab . De plus, nous devons prendre en

compte les surcoûts dus aux états de l'optimiseur, aux gradients et aux activations avant.

Pour réduire considérablement l'utilisation de la **VRAM**, nous devons affiner le modèle en précision 4 bits, c'est pourquoi nous utiliserons QLoRA ici.

Configure les imports nécessaires pour l'affinage d'un modèle de langage en utilisant diverses techniques et bibliothèques avancées. Cela inclut des modules pour charger des jeux de données, configurer et entraîner des modèles, optimiser l'utilisation de la mémoire, et suivre les expériences.

```
[ ] import os
import torch
from datasets import load_dataset
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
    AutoTokenizer,
    TrainingArguments,
    pipeline,
)
from peft import LoraConfig, PeftModel, prepare_model_for_kbit_training
from trl import SFTTrainer
```

Dans cette section, nous allons affiner un modèle Llama 2 avec 7 milliards de paramètres sur un GPU T4 avec une grande RAM en utilisant Google Colab (2,21 crédits/heure). Notez qu'un T4 dispose seulement de 16 Go de VRAM, ce qui est à peine suffisant pour stocker les poids de Llama 2-7b ($7 \text{ milliards} \times 2 \text{ octets} = 14 \text{ Go}$ en FP16). De plus, nous devons prendre en compte les surcoûts dus aux états de l'optimiseur, aux gradients et aux activations avant.

Pour réduire considérablement l'utilisation de la VRAM, nous devons affiner le modèle en précision 4 bits, c'est pourquoi nous utiliserons QLoRA ici.

```
[ ] # Model
    base_model = "meta-llama/Llama-2-7b-chat-hf"
    new_model = "llama-2-7b-miniplatypus"

    # Dataset
    dataset = load_dataset("EssalhiSara/french.corpus", split="train")

    # Tokenizer
    tokenizer = AutoTokenizer.from_pretrained(base_model, use_fast=True)
    tokenizer.pad_token = tokenizer.unk_token
    tokenizer.padding_side = "right"
```

1. Configuration de la quantification : Configure le modèle pour utiliser une quantification efficace en 4 bits afin de réduire l'utilisation de la mémoire.
2. Configuration LoRA : Prépare le modèle pour un affinage efficient des paramètres en utilisant l'adaptation à faible rang (Low-Rank Adaptation).
3. Chargement du modèle de base : Charge le modèle de base pré-entraîné avec la configuration de quantification spécifiée et le mappe sur le dispositif approprié.
4. Préparation du modèle pour l'entraînement : S'assure que certaines couches restent en haute précision (fp32) pour la stabilité et l'efficacité de l'entraînement, particulièrement lors de l'utilisation de la quantification.

Cette configuration est cruciale pour affiner de grands modèles sur des ressources matérielles limitées, en équilibrant efficacité et performance.

```
[ ] # Quantization configuration
    bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_compute_dtype=torch.float16,
        bnb_4bit_use_double_quant=True,
    )

    # LoRA configuration
    peft_config = LoraConfig(
        r=16,
        lora_alpha=32,
        lora_dropout=0.05,
        bias="none",
        task_type="CAUSAL_LM",
        target_modules=['up_proj', 'down_proj', 'gate_proj', 'k_proj', 'q_proj', 'v_proj', 'o_proj']
    )

    # Load base model
    model = AutoModelForCausalLM.from_pretrained(
        base_model,
        quantization_config=bnb_config,
        device_map={"": 0}
    )

    # Cast the layernorm in fp32, make output embedding layer require grads, add the upcasting of the lmhead to fp32
    model = prepare_model_for_kbit_training(model)
```

```

▶ # Set training arguments
training_arguments = TrainingArguments(
    output_dir="./results",
    num_train_epochs=1,
    per_device_train_batch_size=10,
    gradient_accumulation_steps=1,
    evaluation_strategy="steps",
    eval_steps=1000,
    logging_steps=1,
    optim="paged_adamw_8bit",
    learning_rate=2e-4,
    lr_scheduler_type="linear",
    warmup_steps=10,
    report_to="wandb",
    max_steps=2, # Remove this line for a real fine-tuning
)

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    eval_dataset=dataset,
    peft_config=peft_config,
    dataset_text_field="instruction",
    max_seq_length=512,
    tokenizer=tokenizer,
    args=training_arguments,
)

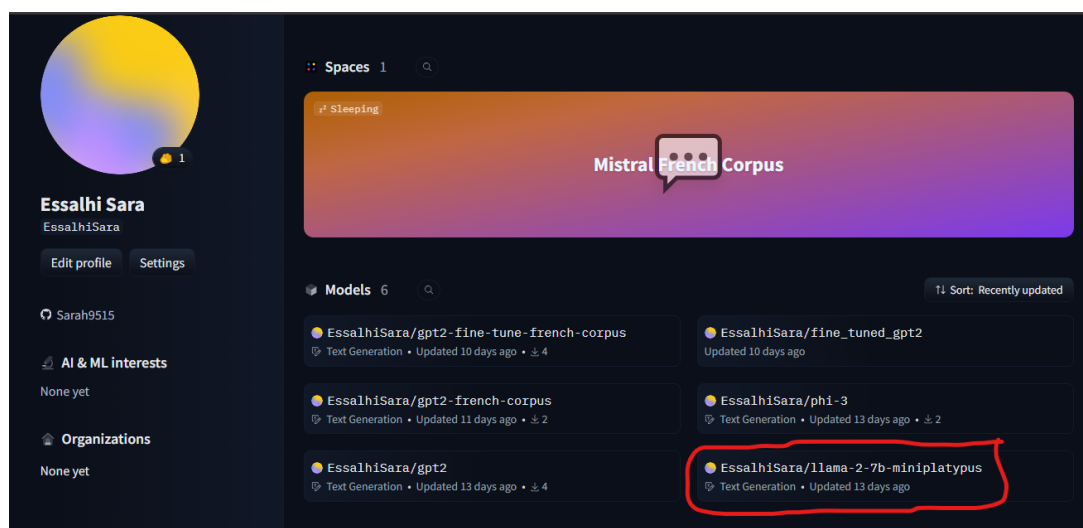
# Train model
trainer.train()

# Save trained model
trainer.model.save_pretrained(new_model)

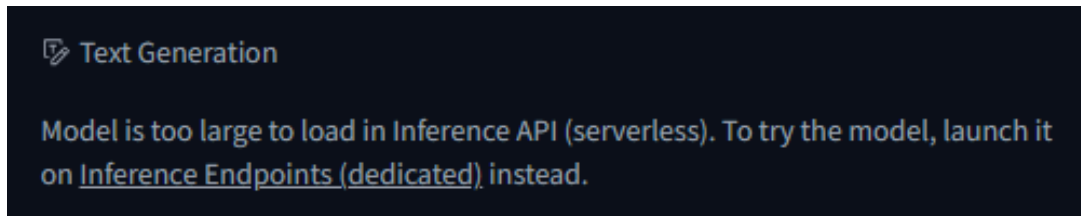
```

- **Déploiement de Llama 2 sur Hugging Face** : Une fois le code exécuté avec succès, notre modèle finement ajusté et le **tokenizer** correspondant seront disponibles dans notre dépôt de modèles sur **Hugging Face**, et pourront être utilisés par d'autres ou intégrés dans des applications via l'**API Hugging Face**. .

```
[ ] model.push_to_hub(new_model, use_temp_dir=False, token=hf_token)
    tokenizer.push_to_hub(new_model, use_temp_dir=False, token=hf_token)
```



Le message de Hugging Face indique que notre modèle est trop volumineux pour être chargé via l'API d'inférence en mode serverless.



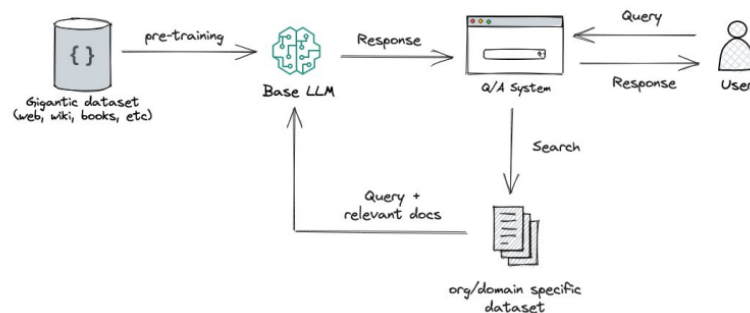
- **Solution** : Je vais travailler exclusivement avec un modèle préexistant sur Hugging Face en utilisant son API. Cela me permettra de bénéficier de la robustesse et de l'efficacité des modèles déjà optimisés et disponibles sur la plateforme. En exploitant les capacités de l'API de Hugging Face, je peux intégrer facilement ces modèles dans mes applications, sans avoir à me soucier des contraintes liées au déploiement et à la gestion des ressources pour des modèles de grande taille. Cette approche garantit une utilisation efficace des ressources et une simplification du flux de travail.

0.4 RAG



0.4.1 Introduction

Dans le domaine des applications nativement basées sur l'IA, Chroma DB et Langchain ont fait des avancées significatives. Chroma DB est une base de données d'embedding (vecteur) open source, conçue pour fournir des moyens efficaces, évolutifs et flexibles de stocker et de rechercher des embeddings. Langchain, quant à lui, est un cadre complet pour le développement d'applications alimentées par des modèles de langage.



0.4.2 Configuration de l'Environnement

Pour commencer, configurons notre environnement. Pour cet exercice, nous aurons besoin des bibliothèques suivantes :

```
1 langchain
2 unstructured
3 fastapi
4 uvicorn
5 pydantic
6 langchain-community
7 langchain-core
8 sentence-transformers
9 chromadb==0.5.0
```


- **langchain** : Un framework complet pour le développement d'applications basées sur des modèles de langage.
- **Unstructured** : Une bibliothèque offrant des outils pour travailler avec des données non structurées, souvent utilisée pour le traitement de texte et d'autres types de données non tabulaires.
- **FstApi** : Un framework web rapide (FastAPI) pour créer des APIs basées sur Python.
- **Uvicorn** : Un serveur ASGI (Asynchronous Server Gateway Interface) basé sur ASGI pour Python 3.6+.
- **Pydantic** : Une bibliothèque pour la validation de données et les conversions de types de données en Python.
- **Langchain-Community** : Une communauté associée à Langchain, fournissant des extensions, des plugins et un support supplémentaire pour le framework.
- **Langchain-Core** : Le cœur (core) du framework Langchain, qui fournit les fonctionnalités de base pour le développement d'applications basées sur des modèles de langage.
- **Sentence Transformers** : Une bibliothèque pour encoder des phrases en vecteurs d'embeddings, souvent utilisée pour le traitement du langage naturel.
- **ChromaDB==0.5.0** : Une base de données d'embedding open source conçue pour stocker et rechercher efficacement des embeddings, dans sa version 0.5.0.

0.4.3 Chargement et Séparation des Documents

Maintenant que nous avons configuré notre environnement, commençons par charger et diviser les documents en utilisant les utilitaires de Langchain.

```

from langchain.document_loaders import DirectoryLoader

directory = 'data/books'

def load_docs(directory):
    loader = DirectoryLoader(directory)
    documents = loader.load()
    return documents

documents = load_docs(directory)
len(documents)

```

Une fois les documents chargés, nous les divisons en utilisant le RecursiveCharacterTextSplitter de Langchain.

```

from langchain.text_splitter import RecursiveCharacterTextSplitter

def split_docs(documents, chunk_size=300, chunk_overlap=100):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        length_function=len,
        add_start_index=True
    )
    docs = text_splitter.split_documents(documents)
    return docs

docs = split_docs(documents)
print(f"Split into {len(docs)} chunks")

```

Ce séparateur de texte, idéalement recommandé pour du texte général, fonctionne sur la base d'une liste spécifique de caractères. Il tente de diviser le texte sur ces caractères dans l'ordre séquentiel jusqu'à ce que les morceaux résultants soient suffisamment petits. Par défaut, il divise le texte en utilisant cette liste de caractères : `["", " ", " ", " "]`.

L'objectif est de maintenir les paragraphes, et par la suite, les phrases et les mots, ensemble aussi longtemps que possible, étant donné qu'ils forment généralement les unités sémantiques les plus puissantes à l'intérieur d'un texte.

0.4.4 Embedding Text par Langchain

Après avoir divisé les documents, l'étape suivante consiste à incorporer (Embedding) le texte en utilisant Langchain. Allons-y et utilisons l'embedding **SentenceTransformerEmbeddings** de Langchain.

```

from langchain.embeddings import SentenceTransformerEmbeddings
embeddings = SentenceTransformerEmbeddings(model_name="all-MiniLM-L6-v2")

```

0.4.5 Création du Vector Store par Chroma DB

Les magasins de vecteurs constituent une méthode courante pour manipuler et rechercher des données non structurées. Le processus standard implique la création d'embeddings à partir des données non structurées, la sauvegarde de ces vecteurs générés, puis, lors d'une requête, l'embedding de la requête non structurée pour récupérer les vecteurs "les plus similaires" à cette requête embarquée. Le rôle d'un magasin de vecteurs est principalement de faciliter cette stockage de données embarquées et d'exécuter la recherche de similarité.

Il est important de noter que Langchain offre une prise en charge pour différents magasins de vecteurs, y compris Chroma, Pinecone, et d'autres. Cette flexibilité permet aux utilisateurs de choisir le magasin de vecteurs le plus adapté à leurs besoins et préférences spécifiques.

Créons maintenant un magasin de vecteurs en utilisant le Chroma DB à partir des documents que nous avons chargés et divisés. .

```
from langchain.vectorstores import Chroma
db = Chroma.from_documents(docs, embeddings)
```

```
PS C:\Users\dell\Downloads\botApi> python create_database.py
Split 96 documents into 10796 chunks.
Impressionné par l'intelligence et la bravoure du renard, le Grand Éléphant lui demanda quel souhait il désirait voir exaucé.
```

0.4.6 Persistance dans Chroma DB

La persistance est un aspect important de toute base de données. Dans cette étape, nous allons créer une instance persistante de Chroma DB.

Si vous souhaitez sauvegarder sur disque, il vous suffit d'initialiser le client Chroma et de spécifier le répertoire où vous souhaitez que les données soient sauvegardées.

```
persist_directory = "/chroma"

vectordb = Chroma.from_documents(
    documents=docs, embedding=embeddings, persist_directory=persist_directory
)

vectordb.persist()
```

```
▼ chroma
  ▼ 430832de-1f4c-4a79-8ead-6f2762a23ea6
    ≡ data_level0.bin
    ≡ header.bin
    ≡ index_metadata.pickle
    ≡ length.bin
    ≡ link_lists.bin
    ≡ chroma.sqlite3
```

0.4.7 Utiliser le modèle Mistral (LLM) avec Chroma DB

Ce code utilise le package Langchain pour instancier un modèle de langage basé sur Hugging Face. Il configure le modèle pour la génération de texte en utilisant un modèle spécifique ("*mistralai/Mixtral-8x7B-Instruct-v0.1*") avec des paramètres spécifiques tels que le nombre maximal de nouveaux tokens générés, le top-k sampling, la température et la pénalité de répétition. Enfin, il définit le token d'API Hugging Face Hub pour permettre l'accès au modèle.

```

from langchain import PromptTemplate, HuggingFaceHub, LLMChain
import os

os.environ['HUGGINGFACEHUB_API_TOKEN'] = 'Your_Huggingface_Api_Token'

model = HuggingFaceHub(
    repo_id="mistralai/Mixtral-8x7B-Instruct-v0.1",
    task="text-generation",
    model_kwargs={
        "max_new_tokens": 2048,
        "top_k": 30,
        "temperature": 0.1,
        "repetition_penalty": 1.03,
    },
)

```

0.4.8 Création du Prompt Template

Ce modèle de prompt est conçu pour encourager les utilisateurs à répondre à des questions en français en se basant uniquement sur un contexte donné. Le modèle fournit un contexte initial, suivi de la question à laquelle l'utilisateur est invité à répondre. L'objectif est d'inciter les utilisateurs à fournir des réponses pertinentes et informatives en utilisant les informations fournies dans le contexte.

```

PROMPT_TEMPLATE = """
Répondez à la question en français en vous basant uniquement sur le contexte suivant :

{context}

---

Répondez à la question en français en vous basant sur le contexte ci-dessus : {question}
"""

# Create the PromptTemplate object
prompt = PromptTemplate(template=PROMPT_TEMPLATE, input_variables=["context", "question"])

# Load the QA chain
chain = load_qa_chain(model, chain_type="stuff", prompt=prompt, verbose=False)

```

0.4.9 Extraction des Réponses des Documents

LangChain introduit une abstraction utile appelée **Chain** pour représenter des séquences d'appels à des composants. Ces composants peuvent inclure d'autres chaînes, ce qui permet de construire des séquences d'opérations complexes et imbriquées. Un type spécifique de chaîne est la chaîne de question-réponse **QA**.

La chaîne QA est spécifiquement conçue pour répondre aux questions basées sur un ensemble de documents fournis. Elle effectue une recherche de **similarité** pour la question

d'entrée par rapport aux documents intégrés, puis utilise un modèle pour générer une réponse basée sur les documents les plus pertinents.

En utilisant la chaîne de question-réponse fournie par LangChain, nous pouvons extraire des réponses des documents.

```
@app.post("/answer/")
async def get_answer(query_request: QueryRequest):
    # Search for matching documents using Chroma
    matching_docs = db.similarity_search(query_request.query)

    # Extract the content from the matching documents to form the context
    context = "\n\n".join([doc.page_content for doc in matching_docs])

    # Concatenate additional context with context from Chroma database
    if query_request.context:
        context += "\n\n" + query_request.context

    # Format the input to the chain using the prompt template
    formatted_input = prompt.format(context=context, question=query_request.query)

    # Run the chain and get the answer
    answer = chain.run(input_documents=matching_docs, question=query_request.query)

    # Extract the response part after the prompt and question
    response_start = "Répondez à la question en français en vous basant sur le contexte ci-dessus : "
    response = answer.split(response_start)[-1].strip()

    # Remove the question from the response
    if response.startswith(query_request.query):
        response = response[len(query_request.query):].strip()

    return {response}
```

0.5 Déploiement

Le déploiement de notre application est géré à l'aide de Docker, qui nous permet de containeriser notre application pour assurer une portabilité et une cohérence entre les différents environnements (développement, test et production). Dans cette section, nous détaillons le contenu de notre fichier Docker et expliquons chaque étape.

0.5.1 Fichier Docker

```

FROM python:3.11.5-slim

WORKDIR /app

RUN pip install --upgrade pip

COPY requirements.txt /app/requirements.txt

RUN pip install --no-cache-dir -r /app/requirements.txt

COPY . /app

COPY chromaa /app/chromaa

EXPOSE 80

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]

```

0.5.2 Explication du Fichier Docker

Voici une explication détaillée de chaque ligne de ce fichier Docker :

- **FROM python:3.11.5-slim** : Cette ligne spécifie l'image de base à utiliser pour notre conteneur Docker. Nous utilisons une version allégée (*slim*) de Python 3.11.5, qui inclut uniquement les éléments essentiels pour exécuter des applications Python, réduisant ainsi la taille de l'image et améliorant l'efficacité du conteneur.
- **WORKDIR /app** : Cette ligne définit le répertoire de travail du conteneur. Tous les chemins suivants seront relatifs à `/app`.
- **COPY requirements.txt /app/requirements.txt** : Cette ligne copie le fichier `requirements.txt` de notre machine locale vers le répertoire de travail du conteneur. Ce fichier contient la liste des dépendances Python nécessaires pour notre application.
- **RUN pip install --no-cache-dir -r /app/requirements.txt** : Cette commande installe les dépendances listées dans `requirements.txt` en utilisant `pip`. L'option `--no-cache-dir` permet de ne pas mettre en cache les fichiers téléchargés, réduisant ainsi la taille du conteneur.
- **COPY . /app** : Cette ligne copie tous les fichiers et dossiers du répertoire de travail local vers le répertoire de travail du conteneur. Cela inclut notre code source et tous les autres fichiers nécessaires pour exécuter l'application.

- **COPY chromaa /app/chromaa** : Cette ligne copie spécifiquement le dossier **chromaa** de notre machine locale vers le répertoire de travail du conteneur. Ce dossier pourrait contenir des données ou des configurations spécifiques nécessaires pour notre application.
- **EXPOSE 80** : Cette directive informe Docker que le conteneur écoute sur le port 80 pour les connexions HTTP. Cela permet de mapper ce port à un port de l'hôte lors de l'exécution du conteneur.
- **CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]** : Cette commande spécifie le programme à exécuter lorsque le conteneur démarre. Nous utilisons **uvicorn**, un serveur ASGI pour les applications FastAPI, pour lancer notre application en spécifiant le module **main**, l'hôte **0.0.0.0** (pour accepter les connexions de n'importe quelle adresse IP) et le port **80**.

0.6 Interfaces de l'Application et Description

0.6.1 Description de l'Application Chatbot :

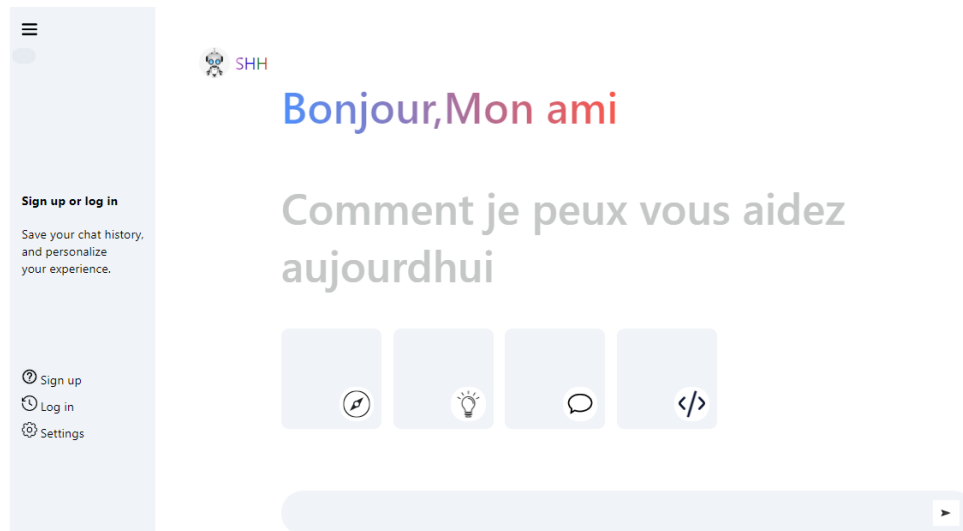
L'application chatbot est une plateforme interactive conçue pour faciliter l'interaction entre les utilisateurs et un agent conversationnel intelligent. Elle offre une interface utilisateur conviviale et intuitive, permettant aux utilisateurs de communiquer avec le chatbot pour obtenir des réponses à leurs questions sur les histoires de jeux vidéo pour les enfants et accéder à diverses fonctionnalités.

0.6.2 Fonctionnalités Principales

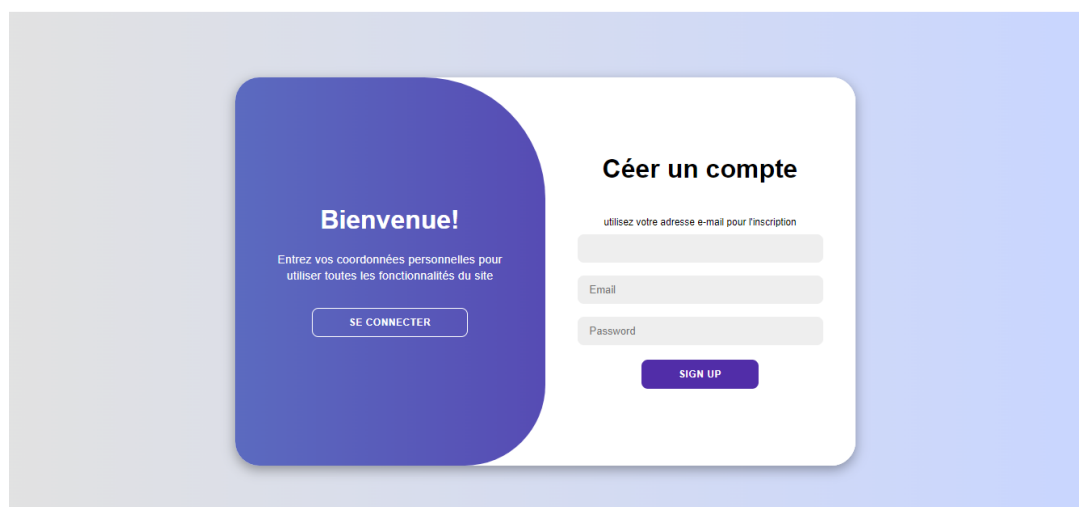
- **Historique des Chats** : Les utilisateurs peuvent consulter l'historique de leurs interactions passées avec le chatbot, ce qui leur permet de retrouver facilement des informations précédemment échangées.
- **Nouveau Chat** : Les utilisateurs ont la possibilité d'initier de nouveaux chats avec le chatbot, leur permettant de poser des questions sur divers sujets, y compris l'histoire des jeux vidéo.
- **Authentification et Gestion de Compte** : L'application propose des fonctionnalités d'authentification sécurisée, permettant aux utilisateurs de créer et de gérer leur compte utilisateur pour une expérience personnalisée.

0.6.3 Interfaces de l'Application

- **Interface de Présentation** : L'interface de présentation est la première interface de l'application, accessible avant même que l'utilisateur ne se connecte. Elle offre une vue initiale du chatbot sans fonctionnalités spécifiques. Pour utiliser les fonctionnalités avancées de l'application, l'utilisateur est invité à créer un compte

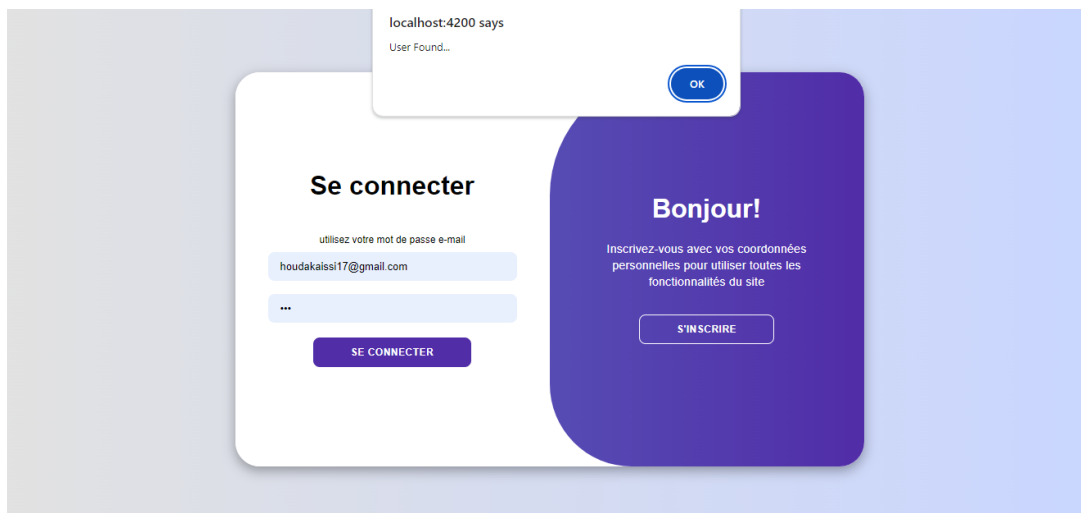
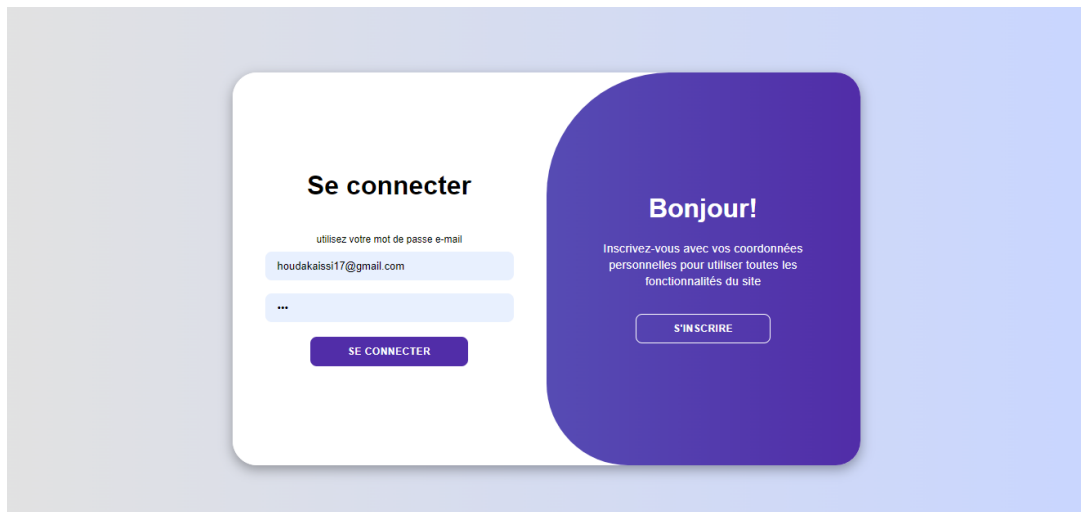


- **Interface de Création de Compte** : L'interface de création de compte est la deuxième étape de l'expérience utilisateur. Ici, les utilisateurs peuvent entrer leurs informations personnelles telles que leur nom, leur mot de passe et leur adresse e-mail pour créer un compte. Cette étape est nécessaire pour accéder aux fonctionnalités complètes de l'application.



- **Interface de Connexion** : Une fois le compte créé, les utilisateurs peuvent accéder à l'interface de connexion pour se connecter à leur compte. Ils doivent

saisir leur adresse e-mail et leur mot de passe pour accéder à l'application chatbot avec toutes ses fonctionnalités.



- **Interface du Chatbot** : L'interface du chatbot est l'endroit où les utilisateurs peuvent interagir avec l'agent conversationnel. Ici, ils peuvent poser des questions sur divers sujets, y compris l'histoire des jeux vidéo. L'application offre des fonctionnalités telles que la visualisation de l'historique des chats et la possibilité d'initier de nouveaux chats avec le chatbot.

Exemple de l'application : demander au chatbot l'histoire de jeu les challenges l'objectif, les participants de jeu video fortnite et demande de generation de jeu similaire a celle ci. . .

New chat

Recent

quel est l'histoire ...

Quels sont les perso...

quels sont les chall...

quel est l'objectif ...

générer un exemple c...



quel est l'histoire de jeu de video de FORTNITE?

Response:

L'histoire du jeu de Fortnite se déroule dans une ère post-apocalyptique où la terre vient de vivre une tempête historique qui a éliminé 98% de la population. Il existe deux versions du jeu, Sauver le monde et Battle Royale. La version Sauver le monde est inspirée du jeu Minecraft et permet aux participants de construire des forts et de combattre des monstres pour collecter des ressources et sauver le monde. La version Battle Royale, quant à elle, connaît un succès planétaire et permet à 100 joueurs de s'affronter dans une arène jusqu'à ce qu'il ne reste plus qu'un seul survivant.

Quels sont les personnages de jeu Fortnite ?

Response:

Les personnages de jeu Fortnite sont des participants dans un monde post-apocalyptique.

New chat

Recent

quel est l'histoire ...

Quels sont les perso...

quels sont les chall...

quel est l'objectif ...

générer un exemple c...

Log out

Settings



Response:

Les personnages de jeu Fortnite sont des participants dans un monde post-apocalyptique.

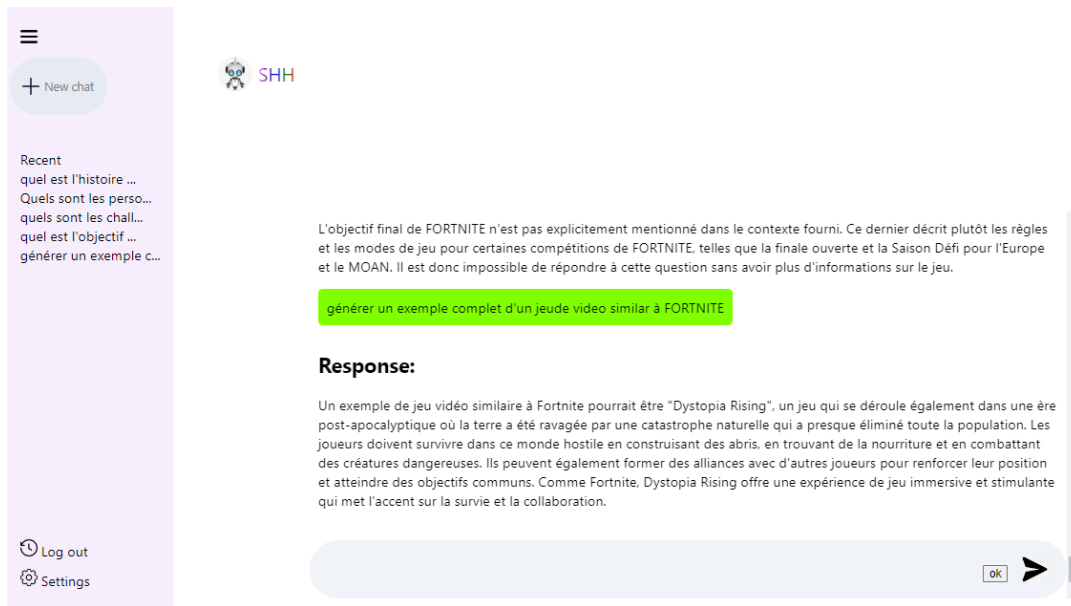
quels sont les challenges de FORTNITE?

Response:

Les challenges de FORTNITE sont des missions hebdomadaires qui permettent aux joueurs de gagner des récompenses. Ces défis sont souvent difficiles à accomplir, mais ils offrent une expérience de jeu unique et gratifiante. Les joueurs doivent utiliser leur créativité et leur stratégie pour terminer les défis et montrer leur maîtrise du jeu. Les challenges sont une partie importante de l'expérience de jeu de FORTNITE et encouragent les joueurs à continuer de jouer et de s'améliorer.

quel est l'objectif final de FORTNITE?

Response:



0.7 Conclusion

En conclusion, le développement de notre chatbot pour la génération d'histoires de jeux vidéo a été réalisé en utilisant une combinaison avancée de technologies backend, frontend et MLOps. L'utilisation de FastAPI, LangChain, Chroma, Mistral, LLaMA2, OpenAI et Hugging Face a permis de créer une application performante et évolutive. Le déploiement efficace via Docker et Kubernetes assure une portabilité et une gestion optimisée des ressources. Ce projet met en lumière l'importance de l'intégration de technologies de pointe pour créer des solutions innovantes dans le domaine des jeux vidéo, tout en soulignant les défis et les opportunités liés à la génération automatique de contenu narratif. L'approche méthodique et les choix technologiques adoptés posent une base solide pour des améliorations futures et une adaptation continue aux évolutions technologiques.

0.8 Bibliographie

- <https://blog.futuresmart.ai/using-langchain-and-open-source-vector-db-chroma-for-heading-1-introduction>
- <https://www.mlexpert.io/blog/fine-tuning-llama-2-on-custom-dataset>
- https://mlabonne.github.io/blog/posts/A_Beginners_Guide_to_LLM_Finetuning.html
- https://huggingface.co/docs/transformers/perf_train_gpu_one#anatomy-of-models-memory
- <https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2>

- <https://huggingface.co/meta-llama/Llama-2-7b-chat-hf>
- https://github.com/SHHChatbot?view_as=public