PLAY-EPIC

Documentation

Welcome to the `Play Epic`!

This is the documentation (approach) of my RAG application for Game Recommendation built with Python 🐍.

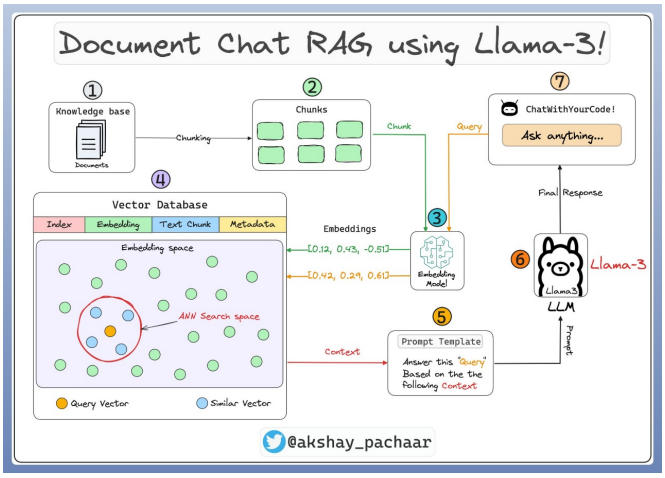
https://playepic.streamlit.app/

https://github.com/jigarsiddhpura/GameRecommendation

Table of Contents

📝 Research

Before diving directly into the code, I began by researching RAG, reading AWS blogs, and medium articles, bookmarked Twitter posts, and understanding the architectures. Below is one of the blogs I read to enhance my understanding of the workflow.



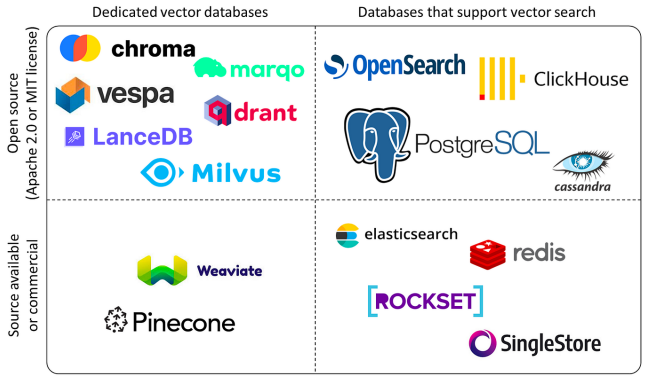
What is RAG? - Retrieval-Augmented Generation Explained - AWS

What is Retrieval-Augmented Generation how and why businesses use Retrieval-Augmented Generation, and how to use Retrieval-Augmented Generation with AWS.

https://aws.amazon.com/what-is/retrieval-augmented-generation/

⌨️ Technology Review

This was followed by reading about which framework (llama_index / langchain) and database (ChromaDB, Pinecone, Weaviate) to use and surfing existing code snippets of RAG. The majority of the resources referred to llama_index, so I went with that option. Then I followed the docs of llama_index which was pretty self-explanatory.



☁️ External Data preparation

After the research stage, I transitioned to the coding phase. To start, I sought out existing external data that could be applied to my project. I explored open-source free APIs on RapidAPI and Python packages that offered game data. Eventually, I discovered SerpApi, which offers Google Search APIs. I opted to use its Google Play Store API for my project.

Google Play Store API - SerpApi

Use SerpApi's Google Play Apps Store API to scrape Google Play Store.

https://serpapi.com/google-play-api

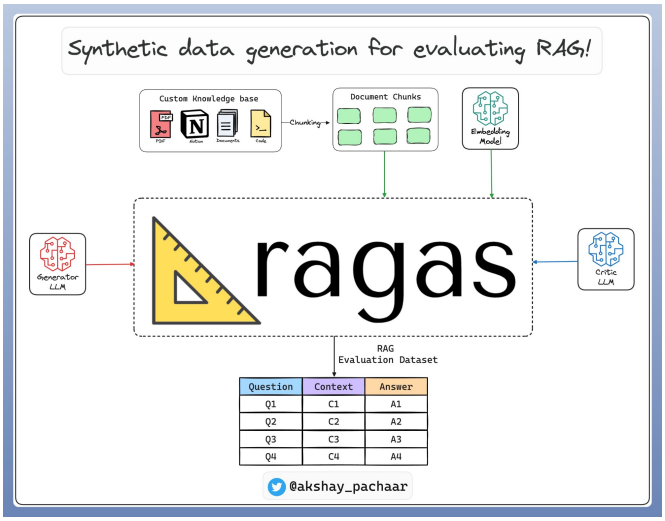
👨‍💻 Model Development

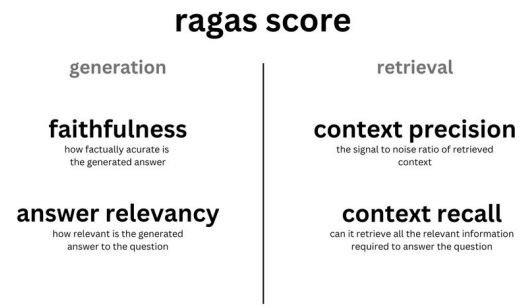
Next, I began constructing the RAG model while concurrently reading the llama_index documents. The construction of the RAG model can be broken down into the following key stages:

1. Using the LLM: Used Anthropic model during retrieval and response-synthesis
2. Loading and Ingestion: This initial stage involved the preparation and intake of the data, setting the groundwork for

further steps.

3. Indexing and Embedding: Here, the ingested data was organized and embedded into a suitable format with Snowflake - a text embedding model, enabling efficient storage and retrieval.
4. Storing and Querying: Post-indexing, the data is stored in a structured manner in ChromaDB which facilitates easy querying of information when required.
5. Evaluation: The final stage involved assessing the performance and effectiveness of the model using RAGAS, ensuring it meets the intended objectives and standards.



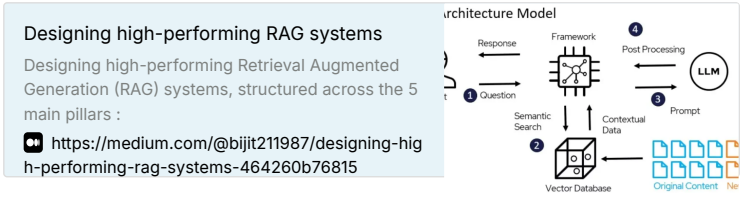


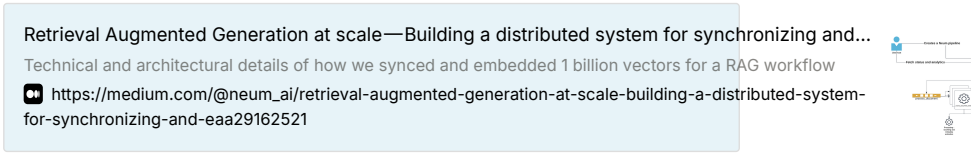
📏 Scaling the model

Once the model was built, I proceeded to research ways to scale it up. Various articles and resources provided insights into different strategies to scale up a recommendation system.

Among different methods like Clustered/ partitioned storage; sharding and distribution of index for parallel retrieval; HNSW for approximate nearest neighbors search; Quantization of embeddings; and the like, I

decided to implement Delta Indexing, a method that only indexes the changes between versions to save on computing resources.





📐 Evaluation

I performed **synthetic test data generation** to evaluate my model with the help of Ragas.

I've calculated the *Faithfulness score* to determine if the answer is faithful to the retrieved contexts (in other words, whether there's a hallucination) and the *Relevancy score* to determine if the retrieved context and the answer are relevant and consistent for the given query.

eval.csv

✨ UI and Deployment

In the later stage, I implemented and deployed the UI using Streamlit. Initially, I faced a compatibility issue between chromaDB and SQLite in the production environment due to a silly mistake while configuring Linux env. However, I was able to resolve it with the following solution, which proved very helpful.