**Assess existing legacy software, and review the source code if available, appreciating that part or all of the system may be implemented using an obsolete programming language.**

Reviewing source code I checked all of dependencies which we used in the project. These dependencies are managed by requirements.txt file, which helps prevent compatibility issues. The versions in use such as: Flask: 3.0.3, gunicorn: 20.1.0, pytest: 7.1.2, pandas: 2.2.3 are relatively recent but may bring potential issues. Just Pick use OAuth2 for user authentication, implemented via the oauthlib.oauth2.WebApplicationClient. This allows the application to delegate authentication to trusted identity providers, which means we do not need to store or manage user passwords. However, when assessing legacy software it's critical to ensure that libraries like oauthlib, as well as other core dependencies, are kept up to date. In addition, proper error handling is essential when integrating with external services such as authentication providers or search engines. While the OAuth2 implementation appears functionally correct, we didn’t apply error handling has not been across the system(semantic search bar).

**Integration LLaMa 3.2 with Flask**

We chose to use LLaMa 3.2 version, which allow interpreting user queries for movie recommendations. Fine-tuned on specific movie datasets and user interaction data.

My understanding of LLaMa I presented in mural and on diagram UML.(Kloda, 2025)

**API Call**:

I specifically assessed the API call implementations for efficiency Async

HTTP calls are handled properly using httpx.AsyncClient, which is great for non-blocking input output. Timeout is set globally, and the error catching implemented with try/except blocks to prevent outright crashes.

I refactored Indexer.py, modified search\_similar method, which now returns a list of movie information strings. It ensure the query input is a string and raises a ValueError if it's a boolean.  
I also add error handlingchanged \_\_init\_\_ method, which includes try-except blocks when attempting to read the 'movie\_index' file and 'movie\_metadata.json' is not found, an empty self.metadata dictionary is initialized, and a message is printed. This makes the class more robust to missing files.

I removed the dependency on GPU resources for Faiss indexing as my hardware was only able to use CPU. While GPUs can offer performance benefits for large-scale vector search, the current deployment environment necessitates a CPU-only solution. I’m planning to explore the possibility of using GPU as it would impact on performance especially that database is expected to grow.

**CORS Configurations:**

* The use of CORS with specific origin restrictions and methods is a fine practice, but as the CORS specification and browser security standards continue to evolve, it's worth ensuring that the origins and methods you allow are still optimal and secure for your use case.

**Cashe responses**: Repeated Queries: If your app gets many repeated or similar queries, consider caching the results of LLaMa's responses. This can be done with in-memory caching (using Redis)

Resources:

[Llama 3.2 Guide: How It Works, Use Cases & More | DataCamp](https://www.datacamp.com/blog/llama-3-2)

Mural – understanding of llama

UML