

# Description

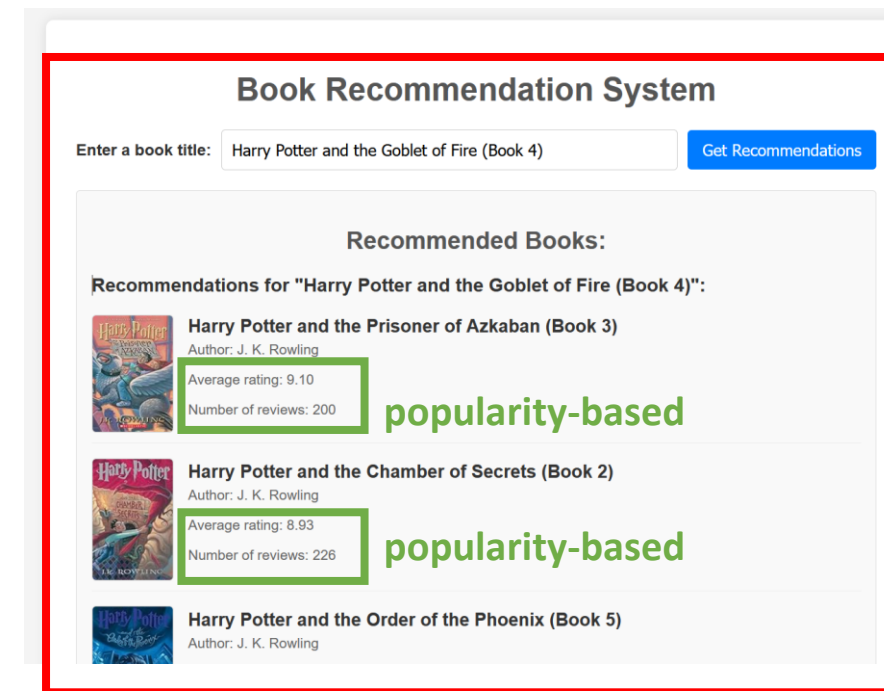
This project demonstrates a basic implementation of a book recommendation system without relying on a live database for book additions or real-time updates. The proposed recommendation system is built using Books.csv and Ratings.csv datasets sourced from Kaggle and provides recommendations based on the learned relationships between books and user ratings.

The system uses a collaborative filtering technique (developed in Jupyter Notebook) to compute similarity scores between books. These scores are stored in similarity\_scores.pkl file and used together with pivot\_table.pkl for generating recommendations. In addition to this, the project has been enriched with popularity-based features. Specifically:

- The books\_dataframe.pkl file contains average ratings and number of reviews for each book title.
- These values were calculated using only the titles present in the pivot table to ensure alignment with the recommendation model.
- **The website UI displays these popularity metrics alongside the recommended titles, offering more insight to users.**

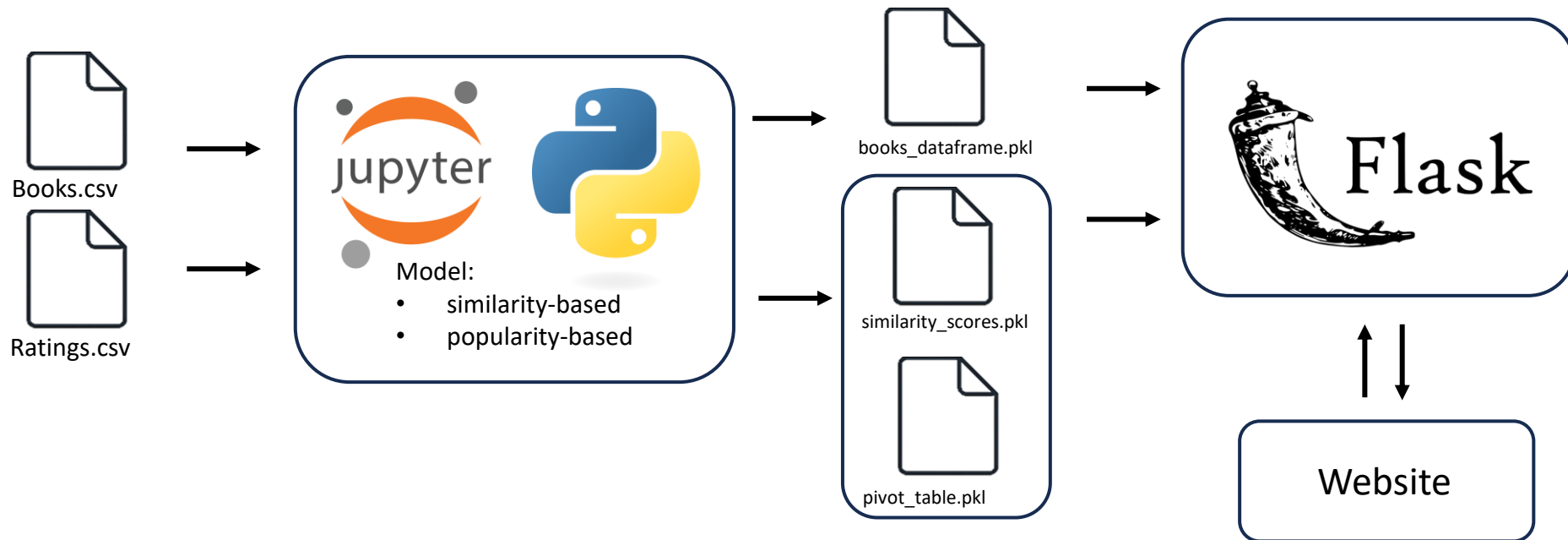
GitHub repo: <https://github.com/Milanomac/book-recommendation-ml-project>

The system is a hybrid of **similarity-based** and **popularity-based** recommendations.



**similarity-based**

# Current project architecture



# Analysis and prototype

- The analysis and thought process are described in more detail in the jupyter notebook inside \research folder in GitHub repo.
- Implicit/explicit ratings considerations

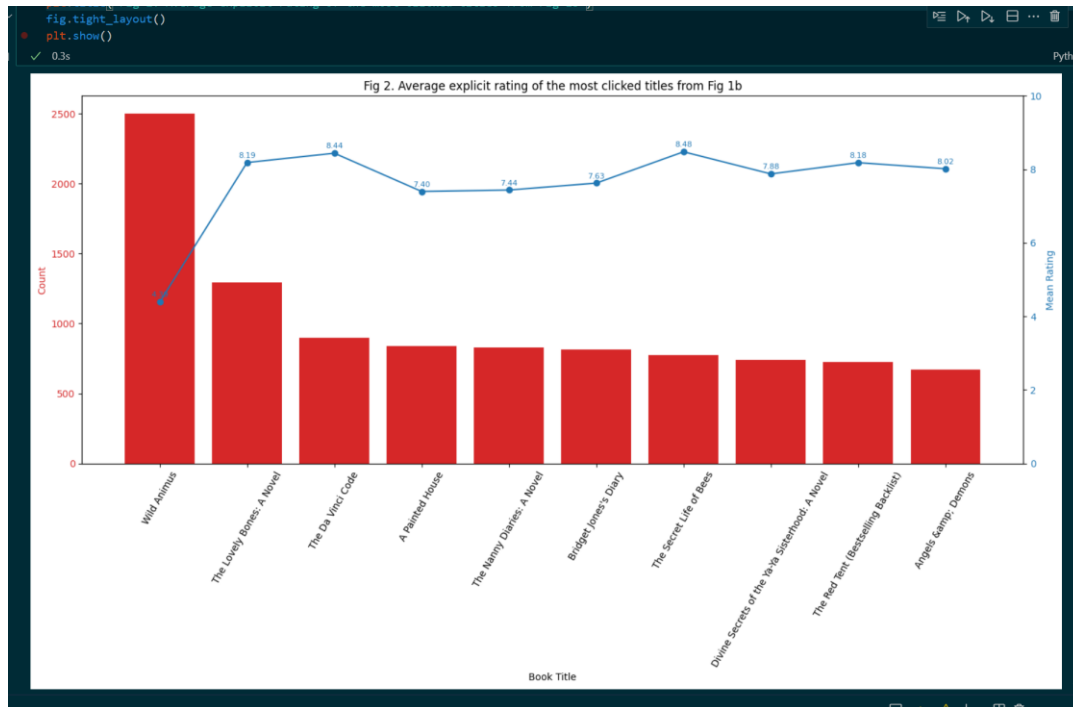


Fig 1a. Explicit Feedback: Top 10 titles with best rating (for books with +100 reviews)

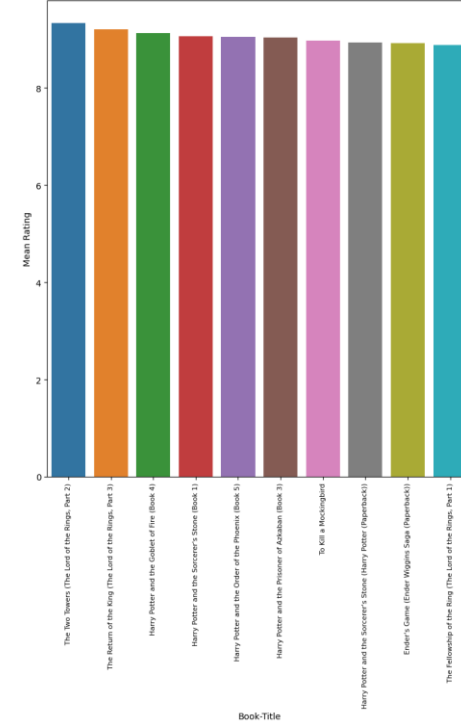
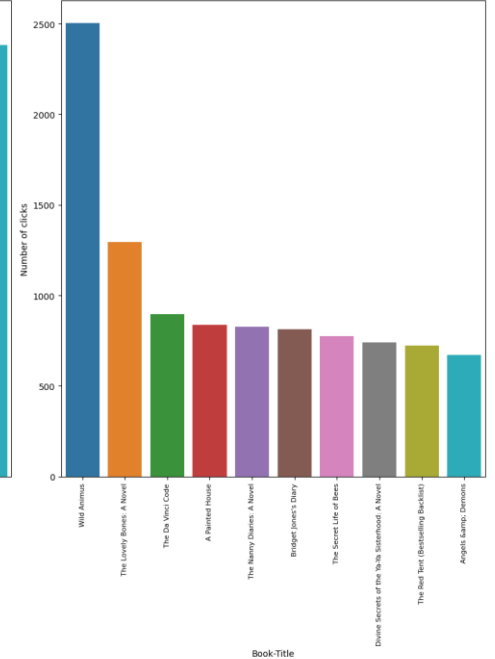
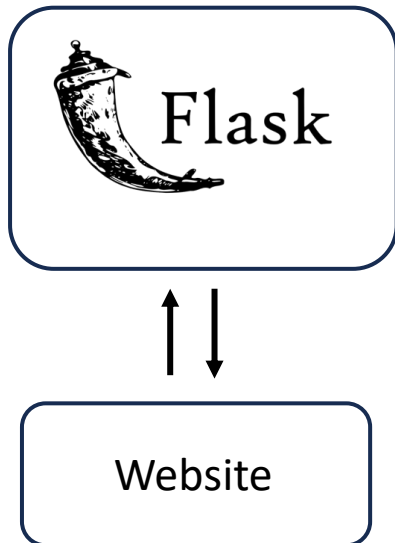


Fig 1b. Implicit Feedback: Top 10 titles with most clicks (popularity)



# Flask API and website

- Defined endpoint (/recommend)
  - getting book recommendations.
- The website sends HTTP POST request(s) and **Flask API responds with 2 possible HTTP responses** (containing JSON data).
  1. API response “**recommendations**” (if an exact title is entered 5 books will be recommended)
  2. API response “**suggestions**” (if a title is not exact get\_close\_matches from difflib finds similar titles and it provides up to 7 suggestions for the title.)






Website:  
First script.js checks if data from the API contains a suggestions array. If it doesn't it creates a button for each suggestion. When one of suggestions is clicked it populates input field and triggers the function again.

### Book Recommendation System

Enter a book title:

**Recommended Books:**

Recommendations for "Harry Potter and the Goblet of Fire (Book 4)":

	<b>Harry Potter and the Prisoner of Azkaban (Book 3)</b> Author: J. K. Rowling Average rating: 9.10 Number of reviews: 200
	<b>Harry Potter and the Chamber of Secrets (Book 2)</b> Author: J. K. Rowling Average rating: 8.93 Number of reviews: 226
	<b>Harry Potter and the Order of the Phoenix (Book 5)</b> Author: J. K. Rowling

### Book Recommendation System

Enter a book title:

**Recommended Books:**

Did you mean?

<input type="button" value="Harry Potter and the Goblet of Fire (Book 4)"/>	Author: J. K. Rowling, Average rating: 9.24, Number of reviews: 178
<input type="button" value="Harry Potter and the Sorcerer's Stone (Book 1)"/>	Author: J. K. Rowling, Average rating: 9.06, Number of reviews: 124
<input type="button" value="Harry Potter and the Chamber of Secrets (Book 2)"/>	Author: J. K. Rowling, Average rating: 8.93, Number of reviews: 226
<input type="button" value="Harry Potter and the Prisoner of Azkaban (Book 3)"/>	Author: J. K. Rowling, Average rating: 9.10, Number of reviews: 200
<input type="button" value="Harry Potter and the Order of the Phoenix (Book 5)"/>	Author: J. K. Rowling, Average rating: 8.93, Number of reviews: 226

# Future project architecture (proposal)

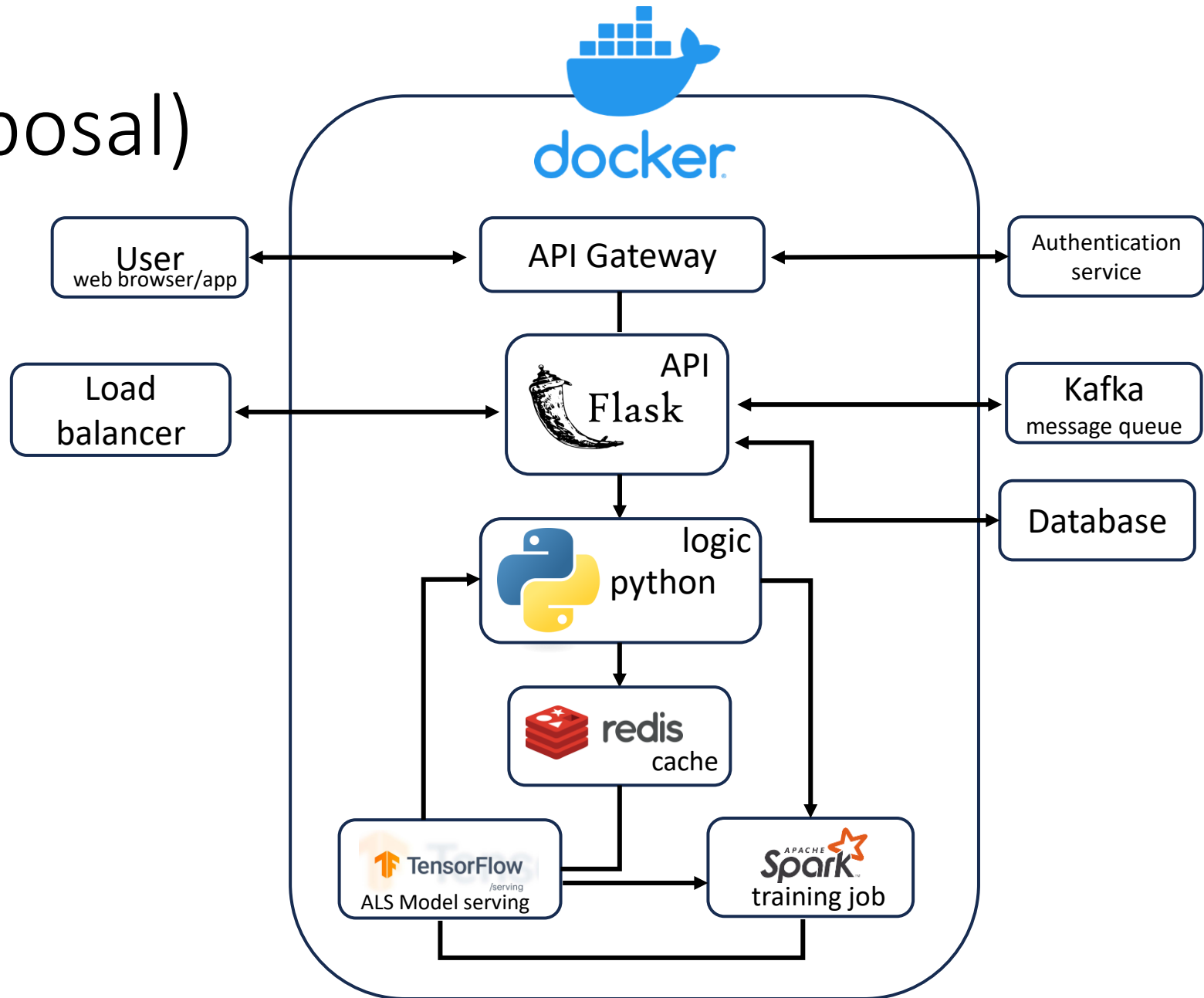
## Improvement notes:

- Users.csv file would have to be included into the model and user, book, and ratings data would be stored in a database.
- Authentication added.
- Flask API now would cover more tasks.
- More robust model: ALS would be implemented and trained within the Spark processing framework to generate the recommendations.
- Docker included for containerization.

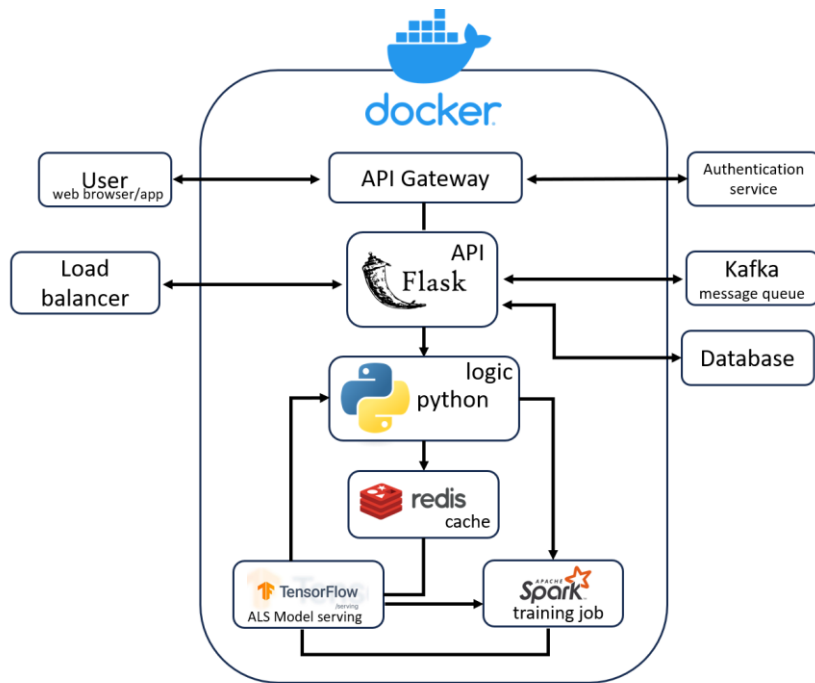
Further description on the next slide

Inspiration:

<https://github.com/Theglassofdata/Netflix-Redis-Cache-Recommendation-Engine/blob/main/README.md>



# Description



1. **Docker** to containerize the application components (Flask API, model serving, Spark workers, etc.) for consistent deployment and scalability.
2. **User Authentication** (i.e. Auth0 / Flask login for smaller apps)
  - The user interacts with the system through a web browser or mobile app.
  - **The API Gateway** routes authentication requests to the Authentication Service.
    - API Gateway handles routing, load balancing, and potentially other concerns like rate limiting.
3. **Flask API:**
  - Receives requests from the API Gateway.
  - Interacts with the database to get user data, book data, and ratings.
  - Optionally, it might get real-time recommendations from the Model Serving component.
  - Puts model update requests to the message queue.
4. **Load Balancer** (distributes incoming traffic across multiple instances of the app for scalability and availability).
5. **Spark** (batch processing - re-training the recommendation models).
6. **Database** (PostgreSQL or MongoDB - stores all app data).
7. **Kafka** – message queue; model update requests triggered by new data added by users.
8. **TensorFlow serving** (for ALS model serving - serves the trained model for real-time recommendations).
9. **Redis cache** (caches frequently accessed data for faster retrieval).

# ALS Model

Alternating Least Squares algorithm is a strong candidate for the recommendation model, and it would be implemented and trained within the Spark processing framework to generate the recommendations.

Why and how it fits for this use case:

- **Collaborative Filtering:** ALS is a popular and effective algorithm for collaborative filtering. It's designed to handle large datasets and is particularly well-suited for Implicit feedback data (like user ratings).
- **Matrix Factorization:** ALS works by factorizing the user-item interaction matrix (user-book ratings) into two lower-dimensional matrices: one representing user preferences and the other representing book (item) characteristics.
- **Spark Implementation:** Spark's MLlib library provides an efficient and scalable implementation of the ALS algorithm. This makes it a good choice for training the model on large datasets of user ratings.
- **Model Training:** Spark would be used to train the ALS model periodically (daily / weekly) to incorporate new user ratings and update the recommendations.
- **Model Serving:** The trained ALS model (or its factors) would then be transferred to the Model Serving component (TF serving / Triton Inference Server) to generate real-time recommendations for users.