# IE7374 18008 Machine Learning Operations SEC 01 Fall 2025

# Project Scoping Report

# Book Recommendation System

---

**Team Members:**

Purva Agarwal

Ananya Asthana

Karan Goyal

Shivam Sah

Shivani Sharma

Arpita Wagulde

## 1. Introduction

This project aims to design and implement a book recommendation system using the Goodreads dataset which offers information including user reviews, user book interactions and detailed book metadata. By leveraging this dataset, the system will generate personalized book suggestions based on individual user preferences and reading history.

The recommendation system will employ both collaborative filtering and content based filtering techniques to ensure diverse and relevant recommendations. Beyond model development, the project emphasizes the integration of MLOps practices with a particular focus on monitoring recommendation quality, detecting data drift and establishing retraining mechanisms. Since click-through rate (CTR) is a live metric not directly available in the dataset, user roles and interactions will be simulated to approximate CTR and evaluate the system's performance. Thresholds such as retraining the model when user engagement falls below a defined benchmark (example, clicking on fewer than 2 out of 10 suggested books) will be incorporated to ensure the system remains adaptive and effective.

The overarching goal is to build a production-ready recommendation system complete with monitoring, drift detection and automated retraining capabilities.

## 2. Dataset Information

1. **Dataset Introduction**:

   The core dataset for this project consists of Goodreads Book Reviews dataset. We will use the UCSD Book Graph for this. This dataset contains book ratings and reviews from Goodreads.com, making it ideal for our recommendation system. The entire interaction dataset is very huge so we will focus on genre specific dataset. Its purpose is to provide manageable real-world user-item interaction data, allowing us to build a model that can predict a user's preference for unseen books within that genre. Subsequently, the datasets will be expanded to include all the genre's interaction datasets, to expand the recommendations made.

2. **Data Card**:

   **2.1 Dataset Name:**
   Goodreads Book Graph Datasets

   **2.2 Dataset Size:**
   Initial size estimated for user-book interaction data per genre is approximately 22 Million records per genre. The size will grow significantly upon broadening the dataset to include all genres.

   **2.3 Dataset Format:**
   The core interaction data is csv, with metadata which is stored in json

   **2.4 Data Types:**
   *Interaction And Reviews Data:*
   Text (User Id, Book Id, Review Id, Review Text)
   Integer (Rating, No of votes, No of Comments)
   DateTime (Date Added, Read At)
   Boolean (Is Read)

   *Book Metadata:*
   Text (Book Id, Authors, Title, Popular Shelves)
   Float (Average Rating)
   Integer(Ratings Count)

3. **Data Sources**:

   The dataset was collected and provided by Mengting Wan, Julian McAuley's research group at University of California, San Diego.

URL: https://cseweb.ucsd.edu/~jmcauley/datasets/goodreads.html

4. **Data Rights and Privacy**:

   The GoodReads dataset is public and anonymous. It does not contain any Personal Identifiable Information (PII) as user Ids are anonymized. The data is also provided for only academic purposes. Therefore, there are no specific data rights and privacy concerns for this data.

## 3. Data Planning and Splits

### Data Preprocessing:

1. **Data Cleaning:**

   1.1. Remove duplicate user book interactions.

   1.2. Handle missing values in book metadata (e.g. drop books without descriptions, Estimate missing genres)

   1.3 Normalize text fields (lowercasing, removing special characters, tokenization for reviews).

   1.4 Normalize input features to ensure clean and consistent data

2. **Feature Engineering:**

   2.1 Encode categorical variables (e.g authors, genres) using one-hot encoding or embeddings.

   2.2 Represent books using TF-IDF or word embeddings for descriptions.

3. **Filtering:**

   3.1 Build a user–book rating matrix for collaborative filtering.

### Data Splitting:

To evaluate recommendation performance and support MLOps retraining simulations:

**Global Split:**

Training set (70%) - Used for learning model parameters and building the recommendation engine.

Validation set (15%) - Used during development for hyperparameter optimization and model selection.

Test set (15%) - Held out for unbiased final evaluation of system performance.

### 4. GitHub Repository

Book Recommendation System Github Repository

- Folder Structure:

| Folder/File | Description |
|---|---|
| `notebooks/` | Jupyter notebooks for exploration and prototyping |
| `src/` | Source code (data processing, feature engineering, modeling, recommendation) |
| `pipeline/` | ML pipeline orchestration (e.g., Prefect, MLflow) |
| `serving/` | Model API implementation (e.g., FastAPI or Flask) |
| `models/` | Trained models and serialized objects |
| `tests/` | Unit tests for core components |
| `dockerfiles/` | Docker images and related files for reproducibility and deployment |
| `requirements.txt` | Project dependencies (alternative: Poetry pyproject.toml) |
| `.dockerignore` | Files/folders to exclude from Docker build context |
| `.gitignore` | Files/folders to exclude from version control |
| `README.md` | Project overview and usage instructions |

- Readme.md file includes:
  1. Introduction to Book Recommendation System (Goodreads + MLOps)
  2. Team Members
  3. Project Architecture Overview
  4. Data Sources
  5. Installations and Getting Started
  6. Folder Structure

## 5. Project Scope

### 5.1 Problems

- Generic book recommendations don't account for individual reading preferences
- Static recommendation models become stale as user preferences evolve
- Lack of feedback loops to improve recommendation quality
- No systematic way to detect when recommendations are performing poorly
- Difficult to balance between popular books and personalized suggestions

### 5.2 Current Solutions

- Basic collaborative filtering: Amazon-style "Users who bought X also bought Y”
- Content-based filtering: Recommendations based on book metadata similarity
- Hybrid approaches: Combining multiple techniques but without continuous learning
- Manual curation: Goodreads lists and human-curated recommendations

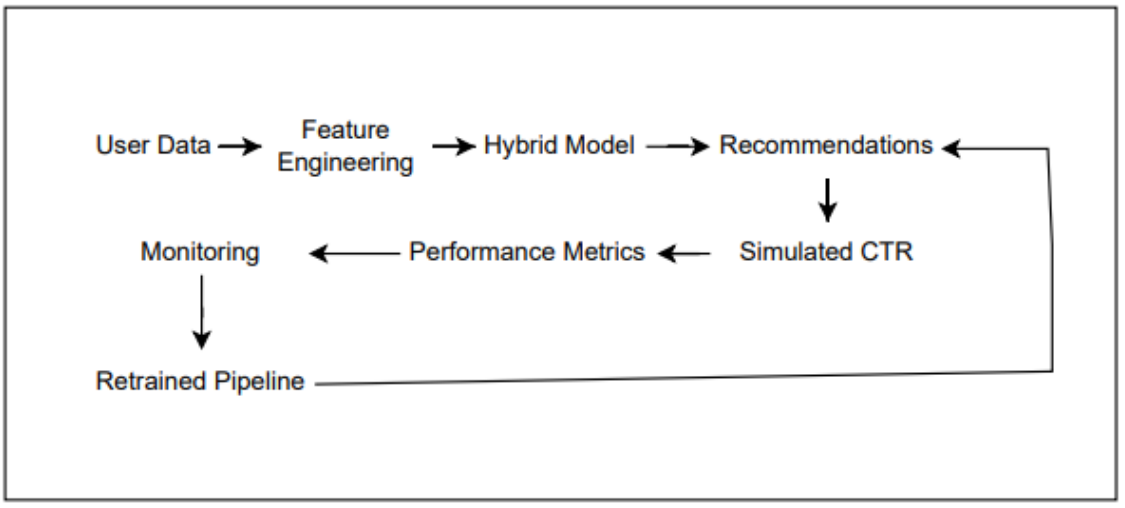
### 5.3 Proposed Solutions

- Hybrid recommendation engine: Combine collaborative filtering and content-based filtering.
- Simulated feedback loop: Implement click-through rate simulation to evaluate recommendations
- Automated retraining pipeline: Trigger model retraining when CTR falls below threshold
- MLOps monitoring: Track model performance, data drift, and system health

## 6. Current Approach Flow Chart and Bottleneck Detection

Current Approach (Traditional Recommendation Systems):


Proposed Approach:




Bottlenecks:

- Data Processing: Large dataset require efficient processing
- Cold Start Problem: New user/books have insufficient data
- Scalability: Real time recommendations for new users
- Feedback simulation: CTR is not available in dataset, must be simulated

**7. Metrics, Objectives, and Business Goals**

To evaluate the recommendation system, we will track:

- Offline Metrics (from historical data):
    1. Precision@K / Recall@K – measures relevance of the top K recommended books.
    2. NDCG (Normalized Discounted Cumulative Gain) – measures ranking quality.
    3. Diversity Score – evaluates a variety of recommendations across genres/authors.

- Simulated Online Metric:

    1. Click-Through Rate (CTR) – assumed by simulating user clicks on suggested books.

2. Thresholded CTR retraining – if fewer than X% of books are clicked in recommendations, the system will trigger retraining to improve personalization.

**Objectives:**

- **Technical Objective**: Build a scalable MLOps pipeline to train, evaluate, and retrain a recommendation model on Goodreads-like data.

- **Analytical Objective**: Test different recommendation approaches (collaborative filtering, content-based, hybrid) and compare performance.

- **Operational Objective**: Automate retraining using simulated CTR as a trigger, mimicking a production feedback loop.

**Business Goals:**

- **User Engagement**: Increase the likelihood of users discovering relevant books, thereby improving platform stickiness.

- **Personalization**: Provide tailored recommendations that adapt to user preferences and behaviors over time.

- **Efficiency**: Showcase how retraining based on user engagement can optimize model performance while reducing wasted recommendations.

## 8. Failure Analysis

### 8.1 Potential Risks

**Data Quality Issues:** The initial dataset may contain significant noise like incorrect ratings or duplicate entries, which could degrade the model performance.

**Model Performance Degradation (Concept Drift):** The model trained on static historical data can become less relevant over time as user taste evolves or new books are introduced.

**Recommendation Diversity Failure:** The model might get stuck recommending only the most popular books or books that are almost identical to what the user just read (filter bubble trap). This limits a user's discovery of new things.

### 8.2 Pipeline Failure Analysis and Mitigation Strategies

| Stage | Potential Failure | Mitigation Strategies |
|---|---|---|
| Data Acquisition | Compressed JSON and CSV files fail to decompress and load correctly due to corruption or load size | Implement robust error handling, use memory efficient streaming or chunking techniques for loading large files. |
| Data Preprocessing | Data leakage occurs between train/test/validation sets, leading to optimistically false offline metric results | Strictly enforce time-based or user-based splitting strategies. |
| Model Training | Model fails to generalize | Fine-tune recommendation engine, try different algorithms, or a combination of algorithms. |
| Prediction Diversity | Model gets stuck into a "Filter Bubble" trap, limiting user discoverability | Incorporate a diversity-boosting term in the ranking (e.g., re-ranking based on diversity). Use hybrid models to introduce novelty. |
| Cold Start | The model is unable to provide any recommendations for a new user or a newly added book. | Create a fallback mechanism: for new users, use content-based features (genres, popular shelves) from book metadata to suggest trending items. For new items, rely on simple popularity or recentness metrics until interactions are logged |
| Deployment | Server issues, scaling problems | Use cloud-based infrastructure, implement auto-scaling, conduct load testing, have a robust deployment pipeline, |

| | | |
|---|---|---|
| | | monitor system performance |
| Monitoring. | Drift in model accuracy goes undetected, leading to poor online CTR. | Establish automated monitoring of key online metrics (like average predicted rating vs. actual rating on new user feedback). Set up alerts for significant drops in CTR, Precision@k, Recall@K and NDCG |

**9. Deployment Infrastructure**

We want our book recommendation system to run smoothly in real life, not just in a classroom. To do that, we'll set it up so everything is **automated, repeatable, and easy to scale**.

**Cloud Platform:** Google Cloud Platform (primary choice, but AWS or Azure are possible alternatives if cost/features dictate). Using GCP allows integration with **GKE (Google Kubernetes Engine)**, **GCS (Google Cloud Storage)**, and **Artifact Registry** for smooth MLOps.

**Servers & Services:**

- **Web/API Server:** The recommender system is exposed via a **FastAPI service** running in **Docker containers**, orchestrated by **Kubernetes**. Requests flow through a Cloud Load Balancer → Nginx Ingress Controller → FastAPI pods.

- **Database/Storage:**
    - User-book interactions, processed datasets, and trained models stored in **Google Cloud Storage (GCS)** buckets.
    - Optional **CloudSQL** or **AlloyDB** can back MLflow's metadata store.

- **Model Serving:** Trained models are versioned and tracked in **MLflow** (registry). The best-performing model is containerized and deployed automatically.

**Orchestration & Automation:**

- **Terraform (IaC):** Defines VPC, GKE clusters, Artifact Registry, GCS buckets, and monitoring stack. The same configuration can spin up **dev, test, and prod**

environments.

- **Airflow (or Prefect):** Runs retraining pipelines, ETL jobs, and evaluation DAGs. Automatically triggers retraining if **data drift** or CTR drop is detected.

- **GitHub Actions:** Handles **CI/CT/CD**—lint/tests, image builds, artifact pushes, triggering Airflow training workflows, and Kubernetes deployments.

**Caching:** Redis may be introduced to store frequently accessed recommendations and reduce latency.

**Logging & Monitoring:**

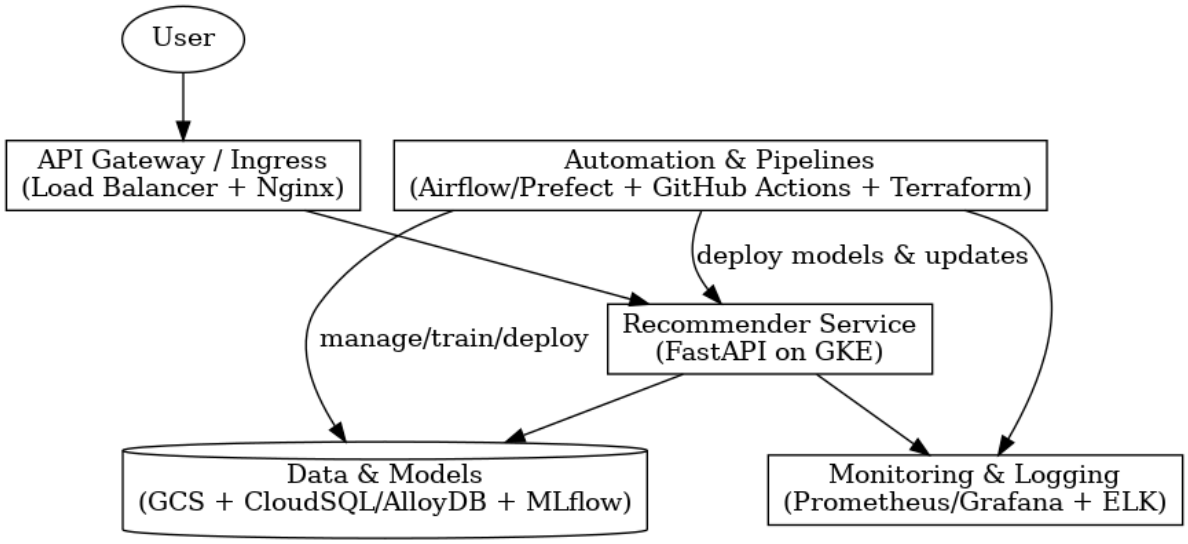
- **ELK (Elasticsearch, Logstash/FluentBit, Kibana):** Collects and analyzes logs.

- **Prometheus + Grafana:** Tracks system health (CPU, memory, API latency) and model metrics (Precision@K, CTR). Alerts trigger if performance dips below thresholds.

**Scalability & Reliability:**

- Kubernetes Horizontal Pod Autoscaler (HPA) adjusts FastAPI pods based on traffic.
- New models will be rolled out using Blue/Green or Canary deployment strategies. Canary deployments will serve the new model to a small fraction of users before full promotion, allowing safe rollouts and live performance testing against the current version.
- GCS object versioning and CloudSQL backups ensure recovery options.

**Flow Diagram -**




**10. Monitoring Plan**

Just building the system isn't enough. We need to watch it constantly to make sure it stays accurate, fast, and useful.

**System Performance:**

- Server resource usage (CPU, memory, network throughput in Kubernetes pods)
- API response latency (FastAPI endpoints under load)
- Request throughput and error rates (5xx/4xx failures)
- Container and pod health within Kubernetes (restarts, pending states)

**Model Performance:**

- **Precision@K, Recall@K, NDCG@K** from offline evaluation
- **CTR (Click-Through Rate, simulated)** – main live metric; retraining if fewer than X out of Y clicks are observed ($X \leq Y$)
- Drift detection (data drift, concept drift, prediction drift) through automated checks
- Evaluation of canary models before production rollout

**Data & Pipeline Health:**

- Data ingestion and preprocessing success/failure counts
- Retraining pipeline completion time and success rate
- Model registry consistency (new models properly versioned in MLflow)

**User Engagement (Simulated):**

- CTR over time (simulated user clicks)
- Distribution of genres/authors recommended → ensuring diversity

**Why Monitor:**

- Detect **model degradation** early (concept/data drift).
- Identify **system bottlenecks** (API slowness, scaling issues).
- Ensure **data integrity** across retraining workflows.
- Catch and fix **errors quickly** via logs and alerts.
- Improve the recommendation engine continuously using simulated feedback loops.
- Support **business goals**: maintaining personalization, engagement, and efficiency.

**Tools & Techniques:**

- **Prometheus + Grafana**: Real-time dashboards for latency, CPU/memory, CTR trends.
- **ELK Stack (Elasticsearch, Logstash/FluentBit, Kibana):** Centralized log collection and search.
- **MLflow:** Tracks experiments, model versions, and metrics.
- **Prefect/Airflow:** Orchestrates retraining when drift or CTR threshold breach is detected.
- **Alerting:** Email alerts for metric thresholds (e.g., low CTR, high API error rate, pod crash).

**Detailed Documentation:**
A separate monitoring handbook will be maintained covering:

- Exact metrics tracked (Precision@K, NDCG@K, CTR, latency, error rates).
- Monitoring stack configuration (Prometheus scrapers, Grafana dashboards, ELK pipelines).
- Alert thresholds and escalation policies.
- Incident response workflow (who gets notified, rollback procedures).

**11. Success and Acceptance Criteria**

1. **Model Performance**: Achieve at least:
    - Precision@10 ≥ 0.6
    - NDCG@10 ≥ 0.7

    - Simulated CTR ≥ 20% in test runs.

2. **Pipeline Reliability**: The MLOps pipeline should successfully handle:
    - Automated data preprocessing and splits.
    - Model training, evaluation, and retraining triggers.
    - Versioned model deployment with rollback options.

3. **Scalability**: Pipeline runs within reasonable compute constraints (≤ 6 hours per training cycle on available infrastructure).
4. The system must generate **top-10 book recommendations** for any given user profile.
5. The system must be able to **simulate CTR feedback** and automatically trigger retraining when performance drops below the threshold.
6. Code and pipeline must be fully reproducible using the provided GitHub repository and documented in the README.
7. A minimal working demo (CLI or notebook-based) must be available to showcase recommendation output and retraining behavior.

**12. Timeline Planning**

| Phase | Dates | Focus | Key Deliverables |
|---|---|---|---|
| **Setup & EDA** | Sept 26 – Oct 6 | Project setup, data cleaning, EDA | Cleaned data, initial insights, updated README |
| **Modeling** | Oct 7 – Oct 20 | Recommender systems (Content & Collaborative) | Baseline models, evaluation scripts |
| **Pipeline (MLOps)** | Oct 21 – Nov 3 | Modular pipeline, config, training automation | pipeline.py, run.py, config.yaml |
| **API + Docker** | Nov 4 – Nov 15 | FastAPI app, model inference, Dockerization | Dockerized API, /recommend endpoint |
| **Finalization** | Nov 16 – Dec 12 | Testing, optimization, documentation | Unit tests, final report, presentation/demo |

**13. Additional Information**

Team Communication: We will use Slack, Discord, or another communication tool for team communication and Trello, Jira, or another project management tool for task management.

Meetings: We will have regular team meetings (e.g., weekly or bi-weekly) to discuss progress, address issues, and plan next steps.

Code Style: We will follow consistent code style guidelines (e.g., PEP 8 for Python) to ensure code readability and maintainability.

Version Control: We will use Git for version control and follow a branching strategy (e.g., Gitflow) to manage code changes.

Testing: We will write unit tests and integration tests to ensure code quality and prevent regressions.

Documentation: We will thoroughly document the code, API endpoints, and system architecture