Milestone 3

Containerization

The inference service is containerized using Docker to ensure a reproducible and isolated runtime environment. The service image is built on top of `python:3.10-slim` and includes only the system dependencies required for model inference (e.g., `libgomp1` and build tools for XGBoost-based models). Application dependencies are managed with **PDM**, with `pyproject.toml` and `pdm.lock` copied early in the build to enable layer caching and reproducible installs.

The service container runs a Flask-based API (`app.py`) that loads recommendation models from disk at startup. Model artifacts and data directories are mounted into the container as volumes, allowing model updates without hard-coding artifacts into the image. The service exposes port `8080` and is configured entirely through environment variables and mounted paths, making the container portable across environments.

Deployment is orchestrated with **Docker Compose**, which defines the inference service lifecycle and networking. Containers are rebuilt and restarted automatically after a successful model promotion triggered by the retraining pipeline. The deployment script (`deploy_app.py`) performs a controlled rebuild and restart of the service, ensuring that updated application code is deployed while model-level traffic routing (stable vs. canary) is handled dynamically at runtime by the application.

Relevant implementation files:
- `docker/Dockerfile`
- `docker/docker-compose.yml`
- `src/app.py`
- `deploy_app.py`
- `cronjob.sh`

Automated model updates

Model updates are handled by a fully automated retraining and deployment pipeline that runs periodically without manual intervention. Each run begins with a **data refresh stage** that executes the existing `DataPipeline`, which materializes new user, interaction, and rating logs into structured datasets while preventing duplicate ingestion across runs (`src/data_downloader.py`). Dataset growth is summarized in a timestamped **data refresh report** (`data/data_refresh_reports/`), which records cumulative statistics and deltas relative to the previous successful refresh.

Retraining is **gated by minimum thresholds** on newly observed data accumulated since the last successful retrain (`data/retrain_state.json`): at least **50 new users** and **100 new interactions**. These thresholds were selected to balance responsiveness and stability—small data increments were found to produce noisy parameter updates and unstable ranking behavior, while waiting for large batches unnecessarily delays adaptation. Users and interactions are monitored (rather than items or ratings alone) because growth along these dimensions directly affects collaborative filtering signals and downstream feature distributions, which dominate model performance. If the thresholds are not met, the pipeline exits early without retraining or deployment.

When the gating criteria are satisfied, the pipeline executes a **multi-stage retraining workflow**: collaborative filtering embeddings are recomputed, features are rebuilt, model hyperparameters are tuned using randomized search, and a new candidate XGBoost model is trained. Evaluation is performed against a **fixed offline holdout dataset** (`offline_eval_data.parquet`) that remains unchanged across runs, ensuring metrics are directly comparable across model versions and preventing evaluation leakage. The test users in this holdout set are explicitly excluded from all training stages, preventing data contamination.

Model promotion is governed by **RMSE as the primary decision metric**. A candidate model is promoted only if it improves over the currently deployed model by a small epsilon margin (ϵ = 1e-6), which prevents churn from statistically insignificant metric fluctuations while still allowing consistent improvements to propagate. Candidates that fail to meet this criterion are automatically rejected and never trigger deployment.

On promotion, the retraining job exits with a success code that **automatically triggers deployment**. The inference service is rebuilt and restarted via Docker Compose, while model-level traffic routing (stable vs. canary) is handled dynamically at runtime by the application. This design cleanly decouples retraining from online serving and avoids coupling model updates to container image contents.

Relevant implementation files:
- Retraining orchestration: `src/retrainer.py`
- Retraining logic and promotion rules: `src/retrain_manager.py`
- Cron-triggered execution: `cronjob.sh`
- Service deployment trigger: `deploy_app.py`
- Logs: `logs/retrain.log`

Releases

For this milestone, we implemented a fully automated canary-release process that gradually deploys new model versions while continuously monitoring live-traffic performance. The release flow is driven by a lightweight load-balancing mechanism inside our inference service and a separate evaluator that decides whether to ramp up, abort, or promote the new model.

How releases are triggered.
Whenever a new model (v2) is trained and passes our offline evaluations, the CI pipeline prepares it for release by activating the canary mechanism. Instead of replacing the running model immediately, the pipeline assigns a small initial canary percentage (10%), meaning only 10% of live requests are served by v2 while the remaining 90% continue using v1. This allows us to deploy new models without service interruption and to observe their real-world performance early.

How the release process is automated.
All online traffic is routed probabilistically: v2 is selected with probability equal to the current canary percentage. The inference server logs, for each recommendation, which model version served it and the items shown. User interactions (clicks) and optional satisfaction ratings are also logged. A separate evaluator script periodically examines these logs and computes:

- CTR overall and separately for v1 and v2
- A fairness ratio comparing satisfaction across demographic groups
- Counts of recommendations, interactions, and quality events

Based on these metrics, the evaluator makes an automated decision:

- **RAMP:** If v2's CTR is healthy (≥10% absolute CTR and ≥70% of v1's CTR) and fairness ratios are acceptable, the evaluator increases the canary percentage by +10%.
- **ABORT:** If v2 underperforms or triggers a fairness concern, the canary is immediately rolled back to 0%, sending all traffic back to the stable model.
- **PROMOTE:** Once the canary reaches 100% and metrics remain healthy, the evaluator promotes v2 to become the new stable version and resets the canary percentage to 0% for the next cycle.

Because routing decisions and configuration are re-read on each request, none of these steps require downtime.

Evidence of successful and aborted releases.
Our system logs every decision into a structured event log that includes the metrics used, the decision made, and the before/after traffic percentages. These logs contain:

- A **correctly aborted release**, showing a drop in v2 CTR relative to v1 (e.g., v2 falling below the 70% ratio threshold), followed by an automatic rollback to 0%.
- A **successful rollout**, where v2's CTR remains consistently strong and the canary percentage progresses from 10% → 20% → … → 100%.
- A **successful promotion**, where the evaluator confirms stable performance at 100% and transitions v2 to become the new stable model.

These event entries collectively demonstrate incremental rollout, automated decisions, no disruption to service, and correct handling of both successful and failed releases.

Relevant implementation files:

- Log file: `feedback_events.json`, `online_metrics.json`
- Release implementation: `src/app.py`, `evalulate_feedback_online.py`

Provenance

We implemented a lightweight, CSV-based provenance tracking system that records the full lineage of models, training data, and predictions for our movie recommendation system. Through it, any past recommendation can be traced to which version of the model, pipeline code commit, and training dataset produced it.CSVs require no database setup, are human-readable, Git-trackable, and work without any external dependencies.
They provide sufficient performance for our system scale (<10M predictions) and allow simple auditing by opening the files directly in Google Sheets or Excel.

Provenance Architecture
The system uses three CSV files stored in: models

1. **model_provenance.csv** — Model Versioning
   Each model trained here is logged with a unique model version, git commit and branch, training data identifier, evaluation metrics, framework/library versions, and an artifact path. Git metadata is automatically captured during training.
2. **data_provenance.csv** — Data Lineage
   Tracks every dataset that was used for training, including a unique data ID, SHA-256 file hash that assures integrity, row and column counts, missing value statistics, and temporal coverage - date_range_start/end.
3. **prediction_events.csv** — Prediction Logging
   Each model prediction logs a request ID (UUID), model version, user ID, timestamp, SHA-256 hash of input features, and inference latency.

Traceability
It answers three central questions about any past recommendation:

- Which model generated this prediction?
  Look for the request ID in prediction_events.csv → model_version.
- Which code and pipeline version was used?
  Use the model_version in model_provenance.csv → git_commit, git_branch, framework versions, dvc reference.
- Which training dataset was used?
  From model_provenance.csv → training_data_id → join with data_provenance.csv.

This provides full traceability from prediction → model → code commit → training data.

Concrete Example

- Suppose a prediction has a request ID “8f1c-42….
- prediction_events.csv shows the model version “v20251202T173325Z-d14d5332”.
- In model_provenance.csv, this version is linked to the git commit d14d5332 and the training_data_id “data_v2”.
- In data_provenance.csv, “data_v2” refers to training_data_v2.csv with a given SHA-256 file hash.

- A reviewer can thus fully reconstruct the model, pinpoint the exact code snapshot, and obtain the correct training dataset.

Integration & Testing

- Training scripts register new models, the prediction service logs every inference, and GitLab CI automatically records git metadata.
- Unit tests cover model registration, dataset registration, and tracing predictions.

Relevant implementation files:
- Core implementation: `src/provenance.py.`
- Tests: `tests/test_provenance.py`
- Offline evaluation: `evaluation/Offline/offline_eval.py` (lines 135-160)
- CSV files: `models/*.csv.`
- CI/CD: `.gitlab-ci.yml` (lines 95-130)

Conceptual Analysis of Potential Problems

Process Used to Analyze Feedback Loops

We first considered how user interactions might reinforce the model's behaviour: Could certain patterns present in the data repeat and amplify over time? We examined the offline logs, genre diversity statistics, and popularity distribution, then used online telemetry to confirm that these patterns also emerged during live traffic. For each pattern, we questioned whether the system might drift toward a narrow set of items or types of users. This helped us identify two clear feedback loops. (Analysis results of the data are stored in offline_feedback_analysis.json)

Feedback Loop 1: Popularity–Interaction Reinforcement

Our offline analysis shows that there is a moderate correlation of **0.24** between movie popularity and the number of interactions that consumers give to those movies. This number might seem low, but it can still create a cycle where popular items are recommended more. More importantly, we found that **10% of the top movies receive roughly 50% of all interactions**, which means a small number of titles have most of the engagement. When these titles are repeatedly shown at the top of recommendation lists, they receive more exposure so people are more likely to click or rate them, and as a result, make them even more likely to be recommended. This feedback loop has two sides. On one side, the system can help quickly converge on well-known items that many users like. On the other hand, less popular items have the possibility of never being shown, and in the long run, it can cause the loss of diversity. We can detect the issue by tracking the click-through rate, which can tell us if users are satisfied with what's recommended to them(more explanation in the latter part). The mitigation for this can be to implement exposure constraints into the ranking stage so that long-tail items at least get minimal visibility. Another approach can be to inject a small randomness or exploration into the candidate generation step.

Feedback Loop 2: Genre Diversity Compression

The offline genre analysis showed that users tend to interact with only about three genres on average, and the 10th percentile is zero genres. This means a large portion of users have extremely narrow viewing patterns. If the model keeps recommending only the genres each user interacted with before, the system never encourages users to explore new areas. This creates a loop: narrow user behaviour leads to narrower recommendations, and can lead to even narrower user behaviour. The result is that the system becomes overly confident in very limited user profiles. We detected the problem by analyzing the offline data, but if in the future, the average or median continues to fall, the system may be reinforcing this loop. To reduce the risk, we can diversify candidates by genre and ensure that each user sees items outside their dominant categories. This does not need to be aggressive because even slight diversification can break the loop and introduce healthier behaviour.

Fairness Considerations

Process Used to Analyze Fairness

We took three steps: First, we located where fairness harms could appear. Since the dataset contains gender labels and clear imbalance, we focused on whether one group receives lower quality recommendations. Second, we examined if any fairness goal conflicts with accuracy or engagement. For example, increasing exposure to minority groups might reduce short-term CTR. Third, we considered how fairness interacts with popularity bias since popularity tends to amplify the patterns of majority groups.

Fairness Issue 1: Training Data Gender Imbalance

The offline logs show a major imbalance where **male users have 77,217 interactions**, and **female users have 15,631**. This leads to a system that learns much more from male viewing data and patterns. Even in the hypothetical case when the average rating is similar offline, the volume difference still can mean that the model internalizes male behavior far more strongly. We also saw this reflected in the generated online quality data, and believe it still has a high probability of happening in the real-world situation. When Female users had **lower average ratings** than male users, it suggests lower satisfaction. This is because when the system learns a lot more from male signals, recommendations may lean towards male but not female. A reasonable mitigation for that is to adjust training weights so that female interactions count more during learning.

Fairness Issue 2: Popularity Bias Hurting Minority Preference Groups

Another fairness-related aspect of popularity bias is that, assuming female users prefer more niche genres, if the model already pushes popular titles to the top, those will be underrepresented. This means that female users will have fewer recommendations matching their interests, despite their tastes being coherent and valid in reality. Detection can potentially come from comparing genre exposure by group or checking the average rank positions of items preferred by minority groups. A mitigation for this issue is to also introduce exposure constraints, such as enforcing a minimum share of recommendations from underrepresented genres or using balanced candidate sampling.

Analysis of Problems in Log Data

How We Analyzed Telemetry

Our telemetry analysis used the online logs that capture recommended items, clicks, ratings, and response times. We ran the feedback-detection script, which computes CTR and gender-based satisfaction ratios. We then compared those metrics against the thresholds defined for healthy behaviour. We understand that our traffic generation script can be limited and not reflect the real-world data accurately, but we have made the effort to design it so that if we have real-world traffic, we are also able to analyze it and provide accurate feedback. (Online logs are stored in online_metrics.json and analysis stored in feedback_events.json)

Feedback Loop Observed Online

To monitor the feedback loops, we decided to use CTR, which measures whether users clicked on the items recommended to them. This is because if a system is healthy, then it should have steady engagement. When a feedback loop occurs, especially the one driven by popularity bias or narrow genre exposure, CTR usually drops because users don't think the recommendations are interesting. In our telemetry logs, the overall CTR was only 2.99%, which is below the expected 10% threshold defined in our detection script. In this case, this low CTR is consistent with the feedback loops identified earlier. Even though the interactions are generated, we believe this situation reflects the real situation. This is because we know the offline data showed that only a small set of movies received the most interactions. As a result, users had fewer opportunities to click on content that matched their tastes if their tastes are different than the popular ones, which can reduce engagement. Therefore, we believe monitoring CTR over time will be important, since further decreases may indicate that the recommendations are becoming stale or overly repetitive.

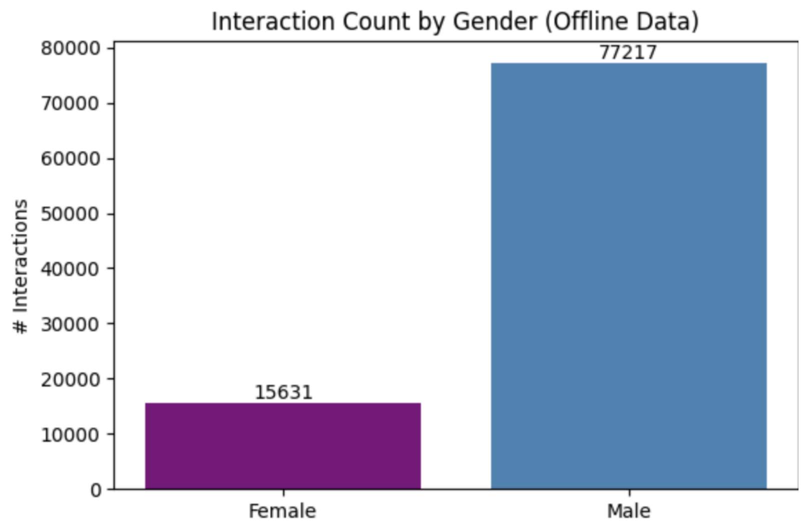
Fairness Issue Observed Online

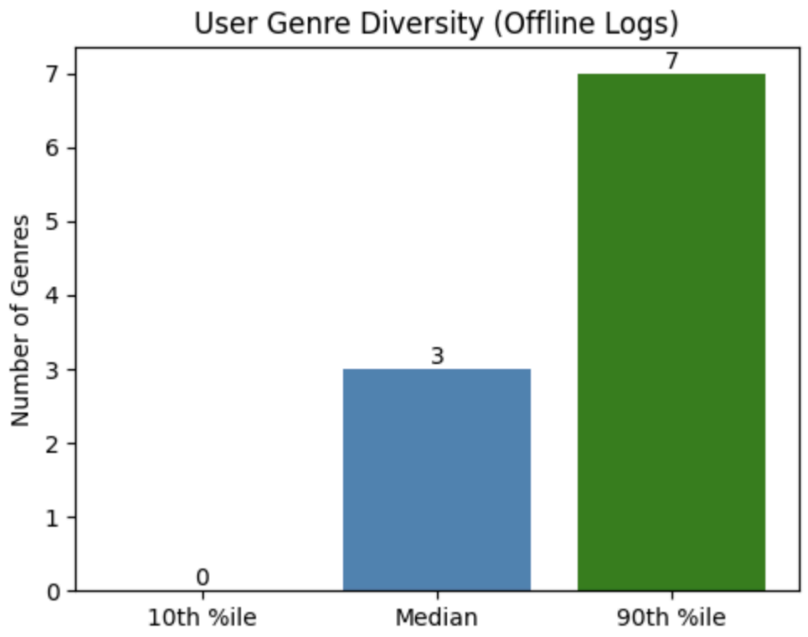
We use: Satisfaction Ratio = Avg rating (Female) / Avg rating (Male)

- **ratio = 1:** perfect fairness
- **ratio < 0.85 or ratio > 1.15:** imbalance detected

After our simulated traffic, results show: Male average rating: 2.77, Female average rating: 2.25, and Satisfaction Ratio Calculated: 0.81, which is below our fairness threshold. We believe this scenario can happen in a real-world situation because of the gender imbalance in data, which is affecting real-time performance. We believe the satisfaction ratio is useful because it gives a simple numerical measure that can be monitored after every deployment. If the ratio consistently stays below the target range, we know the system needs adjustments such as balanced sampling, higher training weight for minority groups, or slice-based evaluation before release.

Exhibit 1: Offline Data Visualization





Reflection on the Recommendation Service

The design, implementation, and deployment of the recommendation service involved coordinating data engineering, model training, evaluation, deployment, and monitoring within tight resource constraints. The most challenging aspect of this project was not the implementation of the models themselves but operationalizing their entire lifecycle-especially making retraining, evaluation, and deployment robust and repeatable in a constrained environment.

Major issues included compute and memory limitations. These factors directly drove many design decisions. For instance, it was impossible to perform full retraining on the entire dataset within the available CPU and memory budget during most hyperparameter tuning or feature construction. To keep things tractable, we capped the size of the training data and used representative sampling during tuning and training. This does introduce some approximation error; however, the retraining pipeline remains useful since real changes are driven by distributional shifts in user behaviour, not an absolute dataset size. The gating mechanism is in place to ensure retraining occurs only when there is enough new signal, thereby limiting large resource usage.

The second major challenge was pipeline stability under automation. Adding in data refresh, retraining, evaluation, model promotion, and container redeployment exposed multiple modes of failure: Docker state conflicts, environment mismatches, and limited observability in cron-based execution. Ensuring idempotency—safe restarts, lockfiles, early exits—required careful handling and is one of the most significant learnings from this work. Testing such pipelines locally is substantially easier than validating under cron and container isolation, which makes debugging lower and more error-prone.

We currently track artifacts and outputs of the retraining pipeline manually. While this works for our current system and use case, it is not scalable and prone to errors. The service will benefit from a more sophisticated system like MLflow or MetaFlow, which tracks the retraining lifecycle and artifacts in much granular detail and traceability.

If this were deployed at scale, a number of things would need additional investment. First, training would need to be completely decoupled from the serving infrastructure and run on distinct computers: for example, batch jobs or managed pipelines. Second, artifact tracking and provenance would work best with tight integration with object storage and model registries rather than local usage of DVC. Finally, true canary deployment would require traffic-level monitoring and automated rollback; we currently simulate synthetic traffic data for our service. The service and implementation could benefit from real-world use/traffic feedback.

The project showed, in general, that system reliability and operational design are tougher than the modelling itself, at times when resources are limited.

Reflection on Teamwork

When reflecting on this stage, we think this project was generally quite effective in terms of teamwork, especially in dividing the responsibilities: data, modelling, deployment, and documentation. We had a high-level alignment on architecture early on, which avoided duplication, and having independent responsibilities of tasks allowed us to work in parallel and make good progress even though there was a heavy workload.

However, one of the biggest challenges we faced was asynchronous development. Because different components were implemented independently, their integration often caused some level of incompatibilities that appeared rather late in the game, especially around data formats, paths, and assumptions about execution environments; this required several rounds of coordination and refactoring near deadlines.

Furthermore, the quality of communication depended on the type of task. Technical design decisions were usually discussed without problems in the Discord Channel, but status visibility was sometimes lacking, especially when parts of the system failed without being noticed under automation. We think that by using more shared runbooks, checklists might have made this situation better.

Yet another challenge was that we often underestimate the complexity of some of the infrastructure we needed to implement. Elements such as retraining automation, Docker composition, and cron execution entailed heavy learning curves and required intense debugging. Also, we think there should be more knowledge sharing among team members. During the collaboration process, this often happened reactively rather than proactively.

In future collaborations, we would benefit from:

- Earlier end-to-end execution tests (even with mocked components)
- Clearer ownership boundaries with explicit integration contracts
- Regular lightweight syncs focused on **system health**, not just feature completion
- Shared debugging artifacts (logs, scripts, failure modes)

Despite these challenges, the team was able to deliver a functional, end-to-end system despite the fact that one of the team members disappeared halfway through the project. This experience taught us that in ML systems, effective teamwork is less about the quality of the individual models and more about coordination around interfaces, assumptions, and operational realities.

Individual Contributions and Meeting Notes

While the team worked together throughout the milestone, each of us split the task based on our previous milestones' experience. Harry worked on the containerization and canary release, while Eric worked on the conceptual analysis of potential problems and the analysis of problems in Log Data. Samuel worked on provenance for tracking model versions with the dvc reference. Finally, Jess worked on data retraining and helped all of us throughout the milestone.

Member	Responsibilities	Key Contributions	Merge Requests
Jessica (Team Lead)	Team leadership, pipeline infrastructure and tests, evaluation supervision.	Developed infrastructure for automatic, periodic model retraining using performance verification and quality metrics. Automated the deployment of updated models and documented the retraining pipeline with direct links to the implementation.	1, 2, 3
Samuel	Provenance, milestone report	Designed and implemented a provenance-tracking system that records model versions, pipeline/framework versions. Created a CSV-based lineage tracker supporting multiple evolving model versions, with clearly defined rules for traceability and debugging.	1, 2, 3
Eric	Conceptual Analysis of Potential Problems, Analysis of Problems in Log Data	Conducted a systematic analysis of potential feedback loops and fairness risks within the ML system, identifying key issues and their impacts. Implemented a monitoring and analysis pipeline to detect these issues in telemetry data, providing evidence-based findings with supporting artifacts. .	1, 2, 3

Harry	Containerization, Canary release	Implemented containerized model inference services and enabled zero-downtime version switching through load balancing. Built a fully automated continuous deployment pipeline with canary releases, offline quality checks, and scheduled model updates every 3 days.	1, 2, 3