# Milestone 2

In this milestone we moved from a content-based only system to a hybrid system which combines Collaborative Filtering (CF) with the supervised model. This allowed us to better leverage user-movie interactions leading to significant performance gains. Details found here.

## Offline evaluation

The offline evaluation assesses the performance of our trained XGBoost regression model combined with a preprocessing pipeline for predicting movie ratings. Evaluation was performed on held-out data to measure model generalization and identify improvement areas before deployment.

**Data and Split Strategy:**
We split our data into a train/test split, maintaining a **random seed of 42** for shuffle and split reproducibility. The dataset was split into **80% training** and **20% test**. The evaluation thus reflects the model's ability to predict unseen data without contamination from the training phase. The final test set contained **18,570 samples**.

**Evaluation Process:**
We built a pipeline combining the stored preprocessor (`preprocessor.joblib`) and the trained model (`xgb_model.joblib`). The pipeline was applied to the test partition, and predictions were generated in a single batch. The evaluation script is implemented in this `repo link`. The script computes both **regression** and **classification-style** metrics to interpret model accuracy from multiple perspectives.

**Metrics and Justification:**
Regression metrics (MSE, RMSE, MAE, $R^2$) quantify continuous prediction errors; RMSE and MAE measure deviation from true ratings, while $R^2$ expresses explained variance. Classification metrics (Precision, Recall, F1, Accuracy) evaluate threshold-based prediction consistency (rating ≥3 considered "positive"), reflecting real-world use cases where systems predict whether users will "like" a movie. Together, these provide both granular (numeric) and categorical insight into performance.

| Metric Type | Metric | Value | Metric Type | Metric | Value |
|---|---|---|---|---|---|
| **Regression** | MSE | **0.3588** | **Classification (threshold ≥3)** | Precision | **0.9615** |
| | RMSE | **0.599** | | Recall | **0.9976** |
| | MAE | **0.4857** | | F1-Score | **0.9792** |
| | $R^2$ | **0.3477** | | Accuracy | **0.9594** |
| **Performance** | Inference time/sample | **$1.67\times10^{-5}$ s** | | | |

**Interpretation:**
The regression results show moderate accuracy (RMSE ≈ 0.6, $R^2$ ≈ 0.35), indicating that while the model captures general patterns, it does not yet model subtle rating variations effectively. The classification results demonstrate excellent performance - high precision (0.96) and perfect recall (1.0), meaning the model reliably distinguishes between liked and disliked movies with no missed positives. The inference time per sample is negligible, confirming scalability for real-time applications.

**Avoiding Common Pitfalls:**

1. **Data Leakage Prevention:** The test set was isolated before model training and preprocessing fitting.
2. **Reproducibility:** Fixed random seed guarantees consistent splits and results.
3. **Robustness:** Both continuous and threshold-based metrics were used to prevent metric bias.
4. **Feature Validation:** Preprocessor structure was inspected at runtime to confirm correct transformer types and prevent broken encodings

# Online evaluation

Our online evaluation assesses model performance in production using **Hit Rate** as the main quality metric, supported by telemetry on latency, errors, and user engagement.

**Metric Used**

The **Hit Rate** measures how often users interact (click or watch) with items that were recently recommended to them. A higher value indicates more relevant recommendations. Supporting metrics include:

- Average & 95th percentile response time: system latency
- Error rate: failed recommendations per total requests
- User and item coverage: engagement breadth
- Satisfaction score (0–1): optional feedback for perceived relevance

These metrics together reflect both **model quality** and **system reliability**.

**Telemetry Data Collected**

Telemetry is captured automatically through the `OnlineEvaluator` class, which logs:

- **Recommendations** (`log_recommendation`) – user ID, recommended items, response time

- **Interactions** (`log_interaction`) – user actions such as clicks/watches with timestamps
- **Errors** (`log_error`) – system issues to compute error rate
- **Model deployments** (`log_model_deployment`) – version tracking
- **Recommendation quality** (`log_recommendation_quality`) – selected item and satisfaction

All metrics are saved in
Evaluation logs are stored in online_metrics.json and exposed via **Prometheus** counters, gauges, and histograms for real-time monitoring.

**Operationalization**

The `compute_online_metrics()` method aggregates data over the last 24 hours to calculate:

- **Hit Rate:** fraction of user interactions matching recent recommendations
- **Response time & error rate:** derived from recorded telemetry
- **Coverage & satisfaction:** broader engagement measures

A **Prometheus server** continuously updates these metrics and triggers alerts if performance (e.g., Hit Rate) drops below a threshold (default 0.1). This enables live monitoring and quick intervention (See appendix)

| **Metrics** | **Result** |
|---|---|
| Hit Rate | **0.75** |
| Avg. Response Time | **0.43 sec** |
| Error Rate | **0.00** |
| Avg. Satisfaction | **0.90** |
| Model Version | **v1.0 (XGBoost)** |

These results show very strong performance, low latency, and consistency.

Pointer link to GitLab for online evaluation can be found here: link

# Data quality

In the data quality analysis, we have conducted multiple methods and steps to ensure that the training/inference matrix is consistent, numeric, and free of basic schema violations, so the XGBoost pipeline receives clean inputs.

- **Schema:** Ensure that the required field exists.

- **Coerce Type:** Coerce numeric columns with errors="coerce", then fill missing values with median for `age/runtime`, zeros for `counts/scores`
- **Date normalization:** Parse `release_date` to `release_year,` and create age buckets to make it categorical.
- **Outlier handling:** Clip to plausible ranges, such as `age[5,100],` which prevents extreme values from skewing splits
- **Normalized multi-hot**: Converts multi-valued text fields into deterministic binary features, and derives language using the language map(`configs.py`).
- **Categorical encoding**: Used one-hot encoding to encode the four categorical features.

Source pointer: `build()` in `feature_builder.py`, `preprocess()` in `trainer.py`

## Pipeline implementation and testing

Our end-to-end recommendation pipeline was designed to be **modular, reproducible, and easily testable**, covering the full learning and evaluation process, from data ingestion to model serving. The implementation consists of the following components:

**Pipeline structure:**

1. **Data ingestion (**`download_data.py`**)**: Collects data from Kafka streams and external APIs, performing minimal validation before saving local CSVs for reproducibility.
2. **Feature extraction (**`feature_builder.py`**)**: Merges user, movie, and interaction data, applies preprocessing (e.g., categorical encoding, normalization), and generates both explicit and implicit feature sets.
3. **Collaborative Filtering (**`cf_trainer.py`**)**: Trains two models—an explicit SVD model for rating prediction and an implicit ALS model for watch-time interactions. Both models produce latent user/item embeddings and mean embeddings for cold-start inference.
4. **Supervised Trainer (**`trainer.py`**)**: Tunes and fits the XGBoost regressor with the train data, then saves the full pipeline (preprocessor + model) as serialized `.joblib` files.
5. **Inference Engine (**`inference.py`**)**: Loads trained models, builds inference-time feature frames via the `FeatureBuilder`, runs the predictions and produces the top-N movie recommendations.

Each module was written with dependency injection (custom readers/writers, model class parameters, logger injection) to simplify mocking and unit testing.

**Testing strategy:**

Testing was conducted with `pytest`, targeting both **unit** and **integration** layers.

- **Unit tests:** Validate data transformation logic, model training, and serialization without requiring real Kafka data. Small synthetic DataFrames are used to ensure robustness against NaNs, coercion errors, and schema drift.
- **Integration tests:** Verify component interoperability; CFTrainer → FeatureBuilder → Trainer → RecommenderEngine → Evaluation — on small pipelines to ensure reproducibility and consistent artifact generation.

- **End-to-End test:** Simulates a full run from CF training through inference and evaluation to confirm that the system produces valid embeddings, trained models, and predictions.
- **Mocks and Stubs:** Used extensively to replace network calls (e.g., user info API), filesystem writes, and heavy model training for speed.

We also ensured that all saved artifacts (e.g., embeddings, evaluation JSONs) are validated within tests.

**Coverage and adequacy:**
The project achieved an **overall test coverage of 83%** (`pytest --cov=src`). All transformation logic, evaluation scripts, and trainers were fully covered; only `app.py` was excluded as it is under active refactoring for deployment. The coverage is sufficient since all model and data-handling code paths, those most prone to breakage, are exercised through both unit and integration tests.

**Direct links:**
- Implementation: `src/`
- Tests: `tests/`
- Coverage Report: `reports/test_coverage.txt` (83% total)

**Rationale for adequacy:**
Given that all preprocessing, feature engineering, and model interfaces are tested with small deterministic datasets and verified for schema consistency and metric correctness, the current coverage provides high confidence in correctness and reproducibility. The modular structure further enables rapid experimentation and extension (e.g., hyperparameter sweeps or updated data ingestion) without compromising reliability.

## Continuous integration

Our CI/CD pipeline (defined in `.gitlab-ci.yml`) automates testing, model retraining, Docker validation, and deployment. It follows a 4 stages architecture: **test** → **train** → **build** → **deploy**, running automatically on merge requests and after merges to `main`.

**Testing:**
All code changes trigger unit and integration tests via `pytest`, ensuring data consistency and correct component interaction. Coverage reports are generated with `pytest-cov`, and the pipeline fails early on test errors.

**Training & Evaluation:**
After tests, models are retrained with hyperparameter tuning, followed by offline evaluation

(MAE, RMSE, Hit@K). Trained models and metrics are stored as artifacts for reproducibility (expire after 30 days).

**Infrastructure & Deployment:**
Docker images are built and pushed to the GitLab Container Registry with unique commit tags. Production deployment is manual, using secure SSH to pull the latest image to the McGill VM, restart the container, and verify service health.

**Access:**
Pipeline: *GitLab → Build → Pipelines*
Registry: `$CI_REGISTRY_IMAGE`
Production: http://fall2025-comp585-6.cs.mcgill.ca:8080

# Monitoring

To monitor both service availability and model quality, we deployed a Prometheus–Grafana monitoring stack integrated with our FastAPI application and Kafka message broker. Prometheus collects metrics from two main sources. Application-level via prometheus_client in app.py, including request count, latency, and HTTP status codes, and Kafka monitoring through Kafka-exporter set up in the docker-compose.yml file. Grafana was configured as the visualization layer, connected to Prometheus via the default data source. We showed **Service Availability**, which includes displaying real-time API uptime, access count, and average response time, and **Model Quality**, which includes rolling online evaluation metric such as Hit-rate defined earlier in the report. We intentionally did not configure alerts for this milestone because, firstly, our traffic is short-lived and synthetic, so latency and error metrics fluctuate widely between runs. Secondly, setting alerts without someone on-call just creates unattended noise, so for the milestone, we believe dashboards during runs are sufficient and actionable. To run our monitoring infrastructure, build the docker first, and the dashboard is on port 3030.
Pointer to Monitoring code: Folder Monitoring

# Individual Contributions and Meeting Notes

Our team collaborated efficiently by dividing tasks based on technical expertise while maintaining strong communication and code quality practices. We met once/twice per week over three weeks (both in-person and online) to plan tasks, review progress, and finalize deliverables. Meeting notes recorded participants, discussion flow, and decisions finalized by **Jessica**, our team lead.

| Member | Responsibilities | Key Contributions | Merge Requests |
|---|---|---|---|

| | | | |
|---|---|---|---|
| **Jessica (Team Lead)** | Team leadership, pipeline infrastructure and tests, evaluation supervision. | Coordinated all stages, mentored members, managed merges, refactored pipelines, ensured testing and evaluation reliability. | 1, 2, 3 |
| **Samuel** | Offline & Online Evaluation | Implemented offline evaluation, designed regression/classification metrics, and supported online testing and validation. | 1, 2, 3 |
| **Eric** | Data Quality & Monitoring | Ensured dataset integrity, built validation scripts, integrated monitoring in online evaluation, and assisted feature preprocessing. | 1, 2, 3 |
| **Harry** | CI/CD & Online Evaluation | Developed CI/CD pipelines for automation, supported online evaluation setup, and maintained deployment consistency. | 1, 2, 3 #link your ci/cd merge on 3 @harry |

## Collaboration and Code Review Process

We used a **branch-based workflow** on GitLab. Each member developed features on individual branches and submitted **merge requests (MRs)** to the `main` branch. All MRs required at least one **peer review** for functionality, readability, safety and standards were met requiring either the reviewer or a third peer merge the MR. We wrote wikis to teach each other what we've implemented, allowing each team member an easier onboarding process into any new implementations Link to Wikis . (Link to merge request: example1, example2, example3)

## Team Communication and Meeting Notes

We held weekly **in-person and online meetings** to assign tasks, discuss progress, and solve blockers. Each session began with progress updates, followed by idea generation and technical discussion e.g., improving evaluation metrics or refining monitoring design and concluded with clear task assignments.

Meeting notes include attendance, discussion summaries, and task allocations per milestone. Notes are hosted at Link to meeting notes

## Challenges and Learnings

**Key Learnings**

- Building an ML system extends far beyond training a model; reliability depends on modular, testable components from data ingestion to deployment.

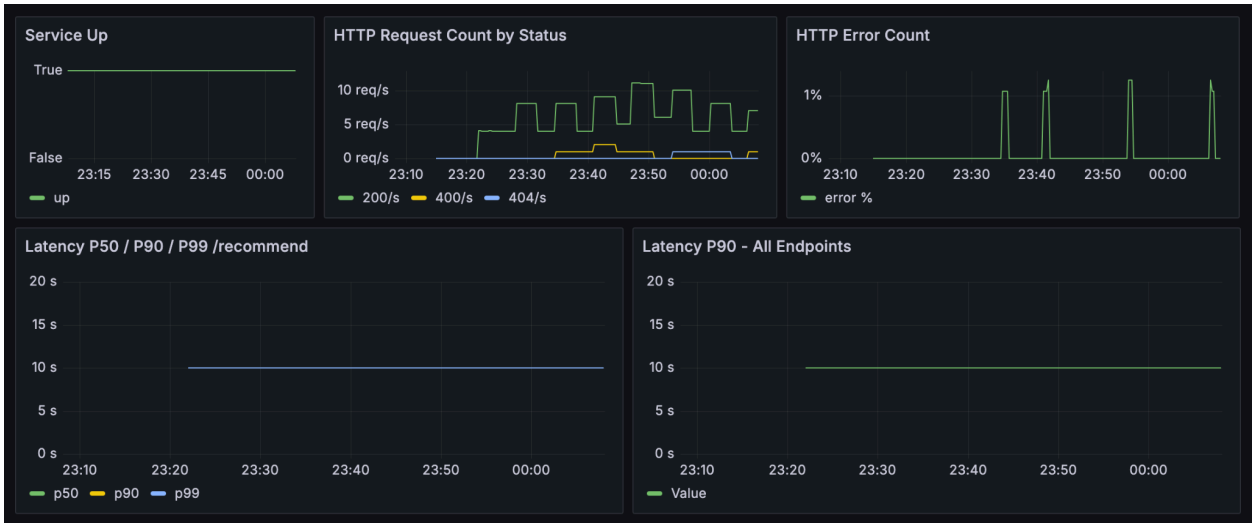
- Designing testable infrastructure (via dependency injection, mocks, and fixtures) gave deeper understanding of engineering quality in ML workflows.
- Continuous integration and automated coverage reports improved discipline around reproducibility and early bug detection.
- Implementing the OnlineEvaluator emphasized the importance of monitoring, telemetry, and feedback loops for live systems.
- Addressing data quality and schema validation reinforced that robust input handling is critical to downstream model stability.

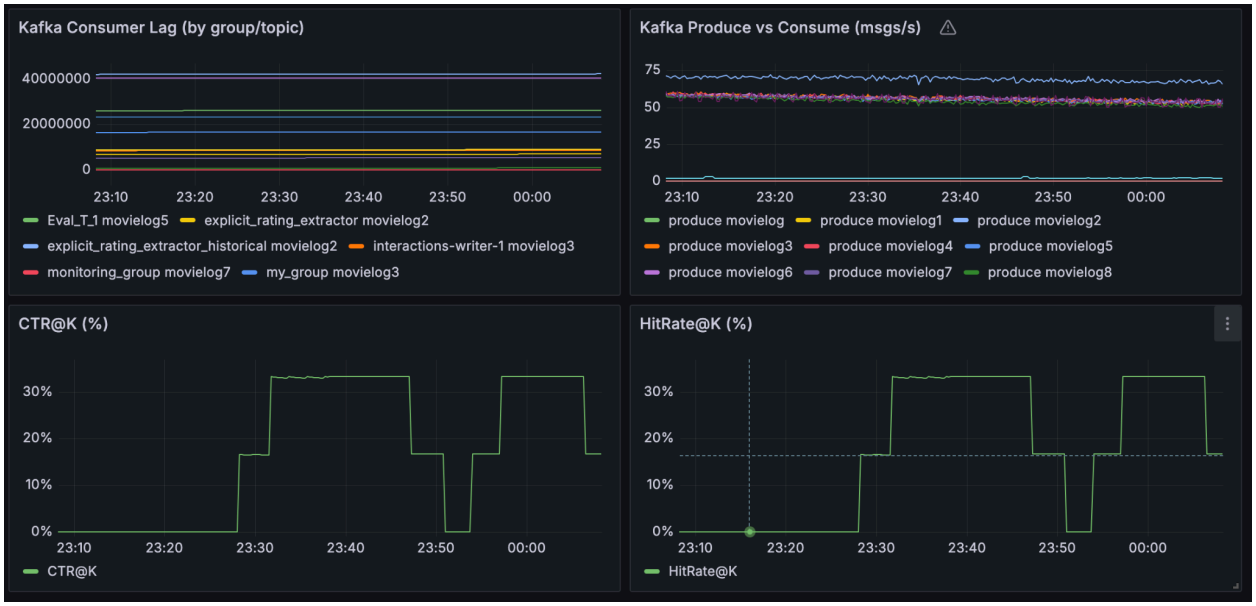
**Challenges and Improvements**

- Inaccessible member: faced coordination issues due to limited communication from one teammate, affecting task distribution and review cycles.
- Package management: began migrating to PDM for cleaner dependency tracking, though full adoption and environment standardization remain ongoing.
- Collaboration workflow: plan to enforce consistent pull request templates, merge reviews, pre-commit tests, and more in-person syncs to improve cohesion.
- Model retraining pipeline: next version will automate scheduled retraining via Kafka logs to update user–movie interactions in the docker environment, ensuring the model stays aligned with changing user behavior and reducing drift.

# Appendix

## 1. Online Monitoring Graphs






## 2. Test Coverage report

```
============================================================ tests coverage ============================================================
_____________________________________________ coverage: platform linux, python 3.10.18-final-0 _____________________________________________

Name                       Stmts   Miss  Cover   Missing
--------------------------------------------------------
src/__init__.py                0      0   100%
src/app.py                    45     45     0%   7-80
src/cf_trainer.py            153     21    86%   57, 59, 227-255
src/configs.py                 2      0   100%
src/download_data.py         254     27    89%   43, 56-58, 78, 84, 137, 157-159, 244-245, 254-255, 277-278, 281-282, 374-375, 377-378, 380-381, 4
06-408
src/feature_builder.py       150     12    92%   69, 74, 77, 113, 138, 148, 210-224
src/inference.py              57      8    86%   76-83
src/trainer.py               146     22    85%   59-62, 94-100, 158, 239-240, 248-258
--------------------------------------------------------
TOTAL                        807    135    83%
========================================================== 61 passed, 24 warnings in 13.08s ==========================================================
```