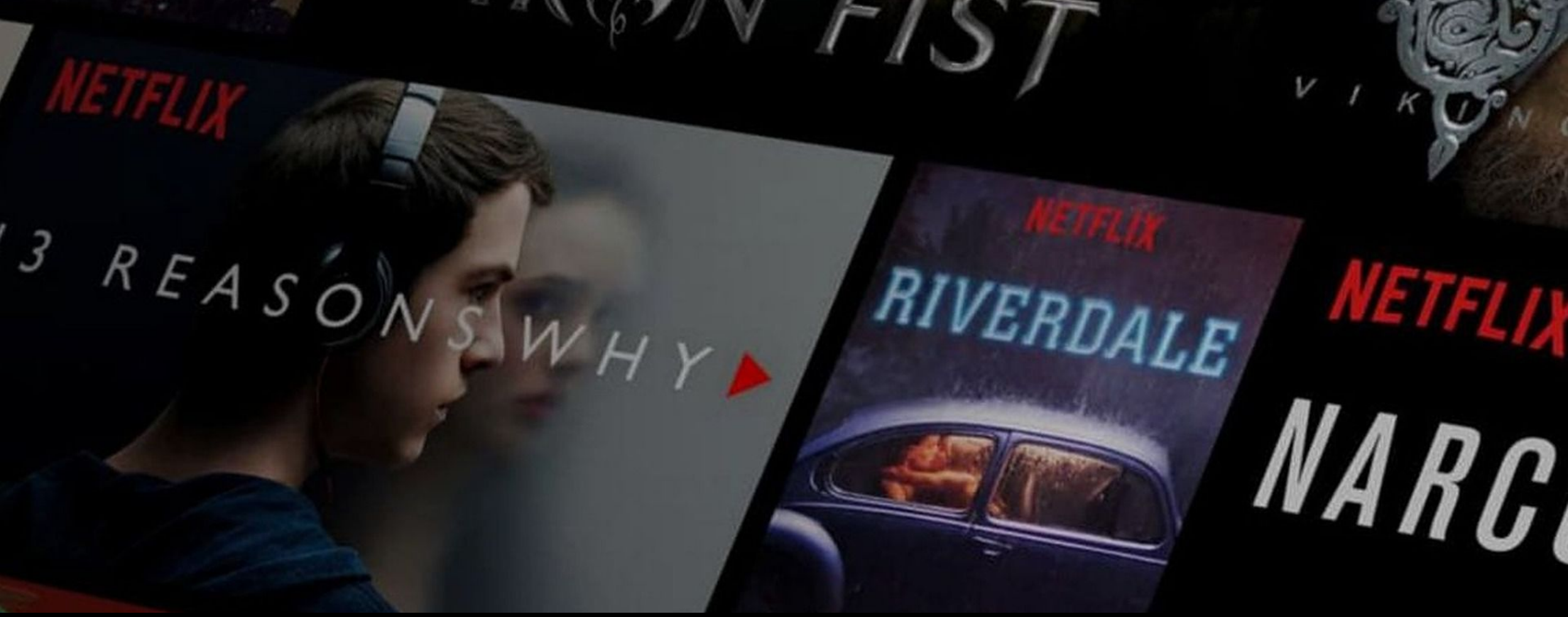


Navigating the Challenges of Building a Movie Recommender System

Keshar Wakharkar, Cynthia Iradukunda, Tanner Balluff, Prashaman Pokharel, Ryan Orth

Key Design Decisions

Alternating Least Squares Model

We chose to use an ALS model for the recommendation algorithm

1. High Hit@20 Rate (Accuracy)
2. Low Serving Cost
3. Compact Artifacts

Popularity-based Fallback

To address limitations of the ALS model, we also implemented a popularity-based fallback for users not in the training data.

Kubernetes Deployment

We deployed the recommendation service using Kubernetes, with three Flask pods behind a Kubernetes Service and Nginx for port mapping.

These key design decisions, including the choice of recommendation algorithm, fallback approach, and deployment strategy, were critical to building a production-ready movie recommendation system.

Our Recommendation Algorithm

- Factorizes a deduped, confidence-weighted user–movie matrix into latent user/item vectors using alternating least squares (tuned: factors, reg, α, iterations)
- Fast on sparse data: scoring = simple user·item dot products → strong offline accuracy, low latency in production
- Limitations: time-unaware (can resurface already-watched titles) and weak for cold-start users/items until retraining

Deployment and Orchestration

CI/CD Pipeline

- Jenkins builds a multi-stage Docker image with BuildKit + metadata
- Runs Trivy security scan
- Executes scripts/k8s-deploy.sh to import the image into k3s and roll out the service

Runtime Architecture

- 3-replica Kubernetes Deployment, recommendation service served via gunicorn
- Exposed through a Kubernetes Service (NodePort), fronted by nginx
- Rolling updates, startup/liveness/readiness probes on /health/*
- Resource limits set, hostPath mount to versioned model symlink

Monitoring and Observability

Prometheus Challenges

We used Prometheus for metrics collection, but it frequently shut down due to memory exhaustion, even after increasing the VM's memory allocation and tuning retention settings.

Overly Aggressive Scraping

We were accidentally scraping metrics from multiple identical pods at a 5-second interval, effectively multiplying the workload for no gain. Reducing the scrape frequency and limiting the scrape target to a single pod would have dramatically improved stability.

Managed Metrics Backend

With more resources, we would have deployed a managed metrics backend like Thanos or Cortex and externalized storage instead of relying on the local node. This would have provided more robust and scalable monitoring capabilities.

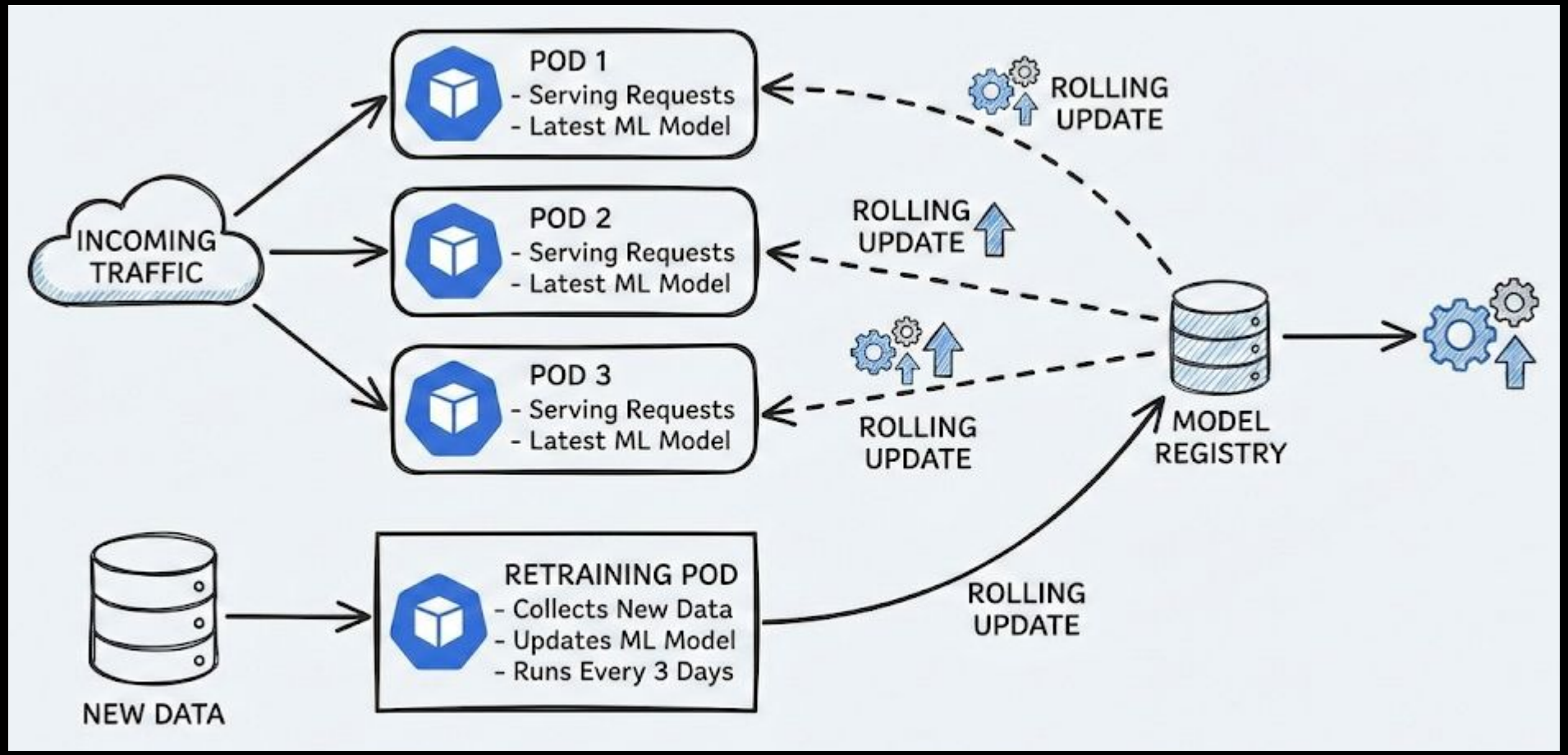
Missing Metrics

Our real-time recommendation metrics were incomplete, missing critical details such as per-movie recommendation frequencies and their temporal change rates. For example, we couldn't detect if a specific movie's recommendation rate spiked 50% in 24 hours making forensic analysis of security incidents nearly impossible in a real-world production setting

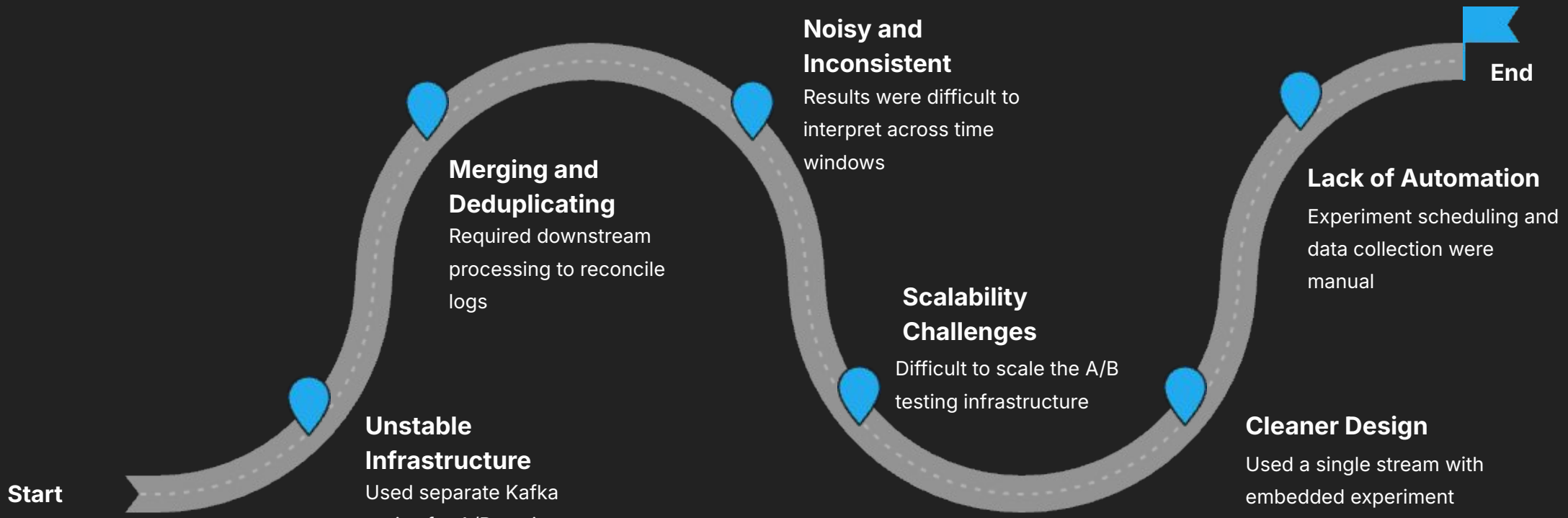
Importance of Monitoring

We recognized that production-grade monitoring requires more than dashboards. It demands tuned scrape configurations, proper retention settings, externalized metrics storage, model-versioned telemetry, and SLO-backed alerting to detect data lag, latency spikes, and coverage gaps before they impact users.

Retraining


POD 1
- Serving Requests
- Latest ML Model
POD 2
- Serving Requests
- Latest ML Model
POD 3
- Serving Requests
- Latest ML Model
INCOMING TRAFFIC
ROLLING UPDATE
ROLLING UPDATE
ROLLING UPDATE
MODEL REGISTRY
RETRAINING POD
- Collects New Data
- Updates ML Model
- Runs Every 3 Days
NEW DATA
ROLLING UPDATE

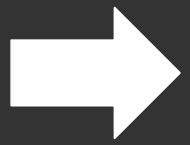
A/B Testing


Start
Unstable Infrastructure
Used separate Kafka topics for A/B testing
Merging and Deduplicating
Required downstream processing to reconcile logs
Noisy and Inconsistent
Results were difficult to interpret across time windows
Scalability Challenges
Difficult to scale the A/B testing infrastructure
Cleaner Design
Used a single stream with embedded experiment tags
Lack of Automation
Experiment scheduling and data collection were manual
End

Provenance, Versioning, and Feedback Loops

Provenance & Versioning

- When we train a model, we record its version, training configuration, data summary, and checksums of all artifacts in a structured manifest.
- Each recommendation response includes lightweight provenance identifiers for the active model and data version, and the same fields are also written into structured logs.
- By using the same provenance fields in serving and logging allows us to trace any prediction from the output back to the exact model and dataset behind it.



Feedback Loops

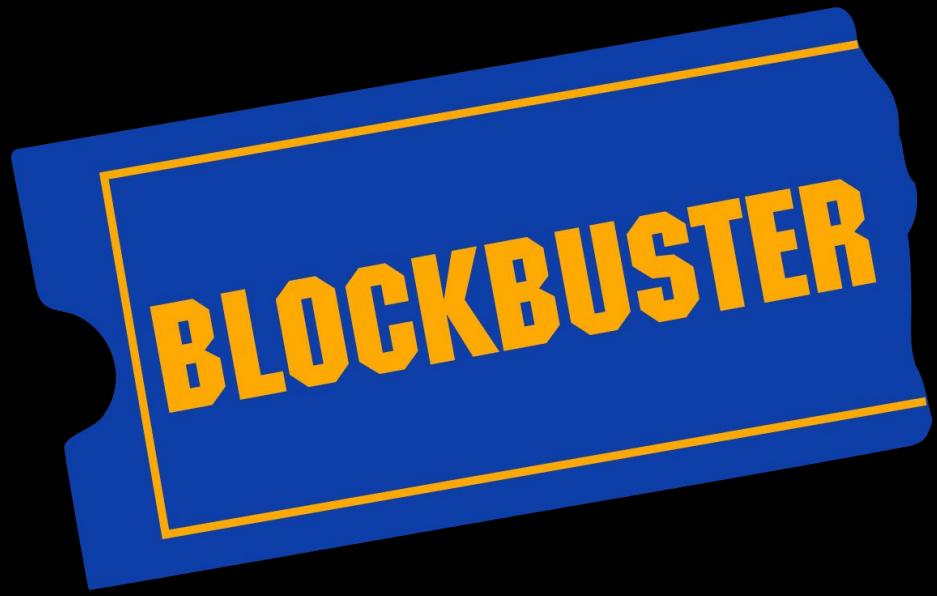
- For each window, we identified the top recommended movies, measured their share of total watches and watch trends in neighboring windows, and tracked catalog coverage and a concentration score.
- Recommendations were already concentrated on a relatively small set of movies, but the watch share of those top titles stayed roughly stable between windows.
- The system favors popular content, but in the period we analyzed we did not see a strong “rich get richer” escalation of popularity.
- Popularity bias remains a risk, so concentration and diversity metrics should be monitored over longer horizons and used to decide when to intervene.

What We Would Change Looking Back

- Use a managed CI/CD platform like **GitHub Actions** instead of self-hosted Jenkins, with built-in PR checks, coverage, and deployment validation
- Separate training and serving, and make data and models versioned artifacts with schema checks, deduplication, and drift detection
- Redesign monitoring and telemetry with leaner Prometheus scraping, external metrics storage, model-version logging, and clear SLO based alerts
- Rebuild experimentation and evaluation around a single tagged Kafka stream for A/B tests and chronological holdout sets for more realistic metrics
- We would add debiasing during training, more exploration for cold-start users, and monitor concentration metrics over time

Summary

- Built and operated an implicit-feedback ALS recommender behind a Kubernetes-backed, Jenkins-driven CI/CD pipeline, exposing a low-latency API for movie recommendations.
- Learned that versioned, reproducible artifacts (data, models, configs) and explicit data-validation / drift checks are critical to avoid silent failures and misleading metrics.
- Saw how fragile systems can be without robust monitoring and observability: Prometheus tuning, model-versioned telemetry, and clear SLOs are essential for debugging and incident response.
- Discovered that A/B testing architecture matters: a single, tagged event stream and automated experiment scheduling are far more scalable than ad-hoc, multi-topic comparisons.




EXIT
Carnegie Mellon University