Scalable Real-Time Recommendation Engine

Swarup E (swarupe@iisc.ac.in) Novoneel C (cnovoneel@iisc.ac.in) Sarthak S (sarthak1@iisc.ac.in)
Rakshit R (rrakshit@iisc.ac.in)

1. Problem

1.1 Definition

The objective of this project was to build a **scalable, real-time recommendation engine** capable of ingesting high-velocity interaction streams and serving personalized content with sub-100ms latency. Unlike traditional systems that rely on batch processing, our system is designed to update its recommendation model incrementally as new data arrives.

Formally, given a continuous stream of rating events $E = \{(u, m, r, t)\}$, where u is a user, m is a movie, r is a rating, and t is a timestamp, the system must maintain a dynamic co-occurrence matrix and generate a ranked list of top-K movies for user u instantly upon request.

1.2 Motivation

In the current digital landscape, the value of data decays rapidly. Industry leaders like Netflix and Amazon attribute a significant percentage of their engagement to recommendation engines, and use them to surface relevant content and retain users. However, traditional Lambda architectures—which maintain separate batch and speed layers—suffer from code duplication and synchronization issues. Batch-only systems fail to capture "in-the-moment" user intent. For instance, if a user starts watching a new genre, a batch system might take 24 hours to reflect this change. Our motivation was to bridge this gap using a **Kappa Architecture**, ensuring that the "speed layer" is the only layer, thereby reducing complexity while improving data freshness. In this format, as new ratings arrive, they are immediately incorporated into the model. This keeps the system responsive while also spreading the computation evenly over time. This also avoids the overhead of having repeated full-dataset scans.

1.3 Design Goals

We set the following design goals for our project at the outset:

- **Low Latency:** Recommendation retrieval time must be under 100ms to support interactive applications.
- **Scalability:** The system must handle the MovieLens 20M dataset and scale horizontally with increased throughput.
- **Freshness:** Model updates (similarity calculations) should reflect in the system within minutes of data ingestion.
- **Fault Tolerance:** The pipeline must recover automatically from node failures without data loss.

1.4 Scalability and Performance Goals

- **Throughput:** Support ingestion of 10,000+ events per second.
- **Storage:** Efficiently manage sparse data structures (millions of users/items) in memory.
- **Serving:** Support 4,000+ concurrent read requests per second.

1.5 Success Criteria and Rationale

We set the following criteria to evaluate the performance of our pipeline:

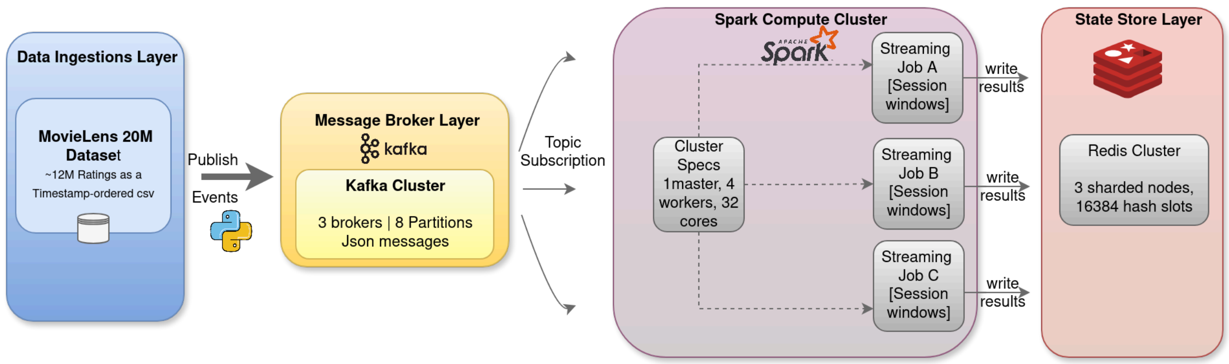
- Precision@10 >= 0.25: an industry standard value for recommender systems.
- Recall@10 >= 0.20: with users having 10-50 relevant items, capturing 20% with 10 recommendations was justifiable.
- Inter-User Similarity <= 0.3: would ensure that the recommendations weren't just the most popular items

We will later see how some of these criteria would have to be re-evaluated as the implementation progressed.

2. Approach and Methods

2.1 High-Level Design: Kappa Architecture

We adopted the Kappa Architecture to unify real-time processing and historical data analysis. In this design, all data is treated as a stream. The immutable log (Kafka) serves as the source of truth, allowing us to reprocess history by simply replaying the stream with different logic if necessary.



Data Ingestions Layer
MovieLens 20M Dataset
~12M Ratings as a Timestamp-ordered csv
Publish Events
Message Broker Layer
kafka
Kafka Cluster
3 brokers | 8 Partitions Json messages
Topic Subscription
Spark Compute Cluster
Spark
Cluster Specs 1master, 4 workers, 32 cores
Streaming Job A [Session windows]
Streaming Job B [Session windows]
Streaming Job C [Session windows]
write results
write results
write results
State Store Layer
Redis Cluster
3 sharded nodes, 16384 hash slots

Figure 1: High-Level End-to-End Kappa Architecture Diagram]

2.2 Architecture and Data Model

1. Data Publish Layer (Python based data replay):

We use the MovieLens 20M dataset and use a kafka based python publisher to replay the events in order of time.

2. Data Ingestion Layer (Apache Kafka):

We chose Kafka as it is the de-facto standard for event streaming. Kafka has durable message storage with replay capabilities, and can be scaled horizontally through partitioning. Along with Kafka, we decided to use KRaft and not Zookeeper as it had simpler operations and did not require a separate cluster to be managed. We configured the Kafka cluster with 3 brokers in KRaft mode. The ratings topic was configured with 8 partitions and a Replication Factor (RF) of 2 to ensure high availability. The data source was the MovieLens 20M dataset, simulated as a continuous stream of JSON events.

3. Stream Processing Layer (Apache Spark Structured Streaming):

We selected Spark Streaming as it has native Kafka integration and a mature ecosystem - with a unified batch and streaming API.
Spark is the core of our ETL and logic. We deployed a Spark cluster (1 Master + 4 Workers) with a total of 32 cores and 32GB RAM. We optimized the processing by decoupling the logic into three distinct operations:

- **Job A (Pair Generation):** Identifying pairs of movies rated by the same user within a specific session window.
- **Job B (Similarity Computation):** A background thread that aggregates pair counts to calculate the Conditional Probability $P(i|j) \approx \frac{count(i,j)}{count(i)}$
- **Job C (Scoring):** Generating candidate recommendations for active users.

4. Serving Layer (Redis Cluster):

For the serving layer, we chose Redis in Cluster mode (3 master nodes) due to its in-memory performance. Redis has sub-millisecond latency for reads as well as rich data structures such as sorted sets. We utilised Cluster mode to be able to handle large co-occurrence updates without single node bottlenecks. We utilized specific data structures for different access patterns:

- **Hashes:** For storing user metadata and rating history.
- **Sorted Sets (ZSET):** For storing pre-computed similarity lists (e.g., item:sim:123) and final recommendations (e.g., usr:rec:456). This allows $O(\log N)$ retrieval of top-K items.

2.3 Big Data Platforms Used

- **Apache Kafka:** To handle back-pressure and buffer high-velocity input.
- **Apache Spark:** To provide fault-tolerant state management via HDFS/checkpointing and micro-batch processing.
- **Redis:** To provide the necessary read-throughput for the frontend API, decoupling the heavy computation (Spark) from the read path.

2.4 ML Methods Used

We implemented **Item-Item Collaborative Filtering** using a streaming co-occurrence approach. Unlike matrix factorization which requires expensive iterative retraining, co-occurrence methods are additive, meaning we can increment counters without retraining the model. As new rating (u, m_{new}) arrives, we update the relationship between m_{new} and all other items m_{old} in the user's history.

To handle the $O(N^2)$ complexity of pair generation for active users, we implemented **Session Sampling**, wherein we cap the history considered for any single session to the 50 most recent items, which restricts the computational cost per micro-batch.

3. Evaluation

3.1 Experiment Design

We simulated a production environment using a temporal split of the MovieLens 20M dataset. The stream replay was accelerated to stress-test the infrastructure.

- **Training Phase:** Initial 80% of timestamp-ordered data replayed to build the similarity model.
- **Testing Phase:** Subsequent 20% of data used to evaluate recommendation quality.
- **Hardware:** The cluster was deployed using Docker containers on a Linux host with 10 CPUs and 10GB allocated to the driver.

3.2 Challenges and Changes

Since the raw MovieLens dataset exhibits significant variability in user activity, many users have very few ratings. To counter this, we applied three filters:

1. At least 50 ratings per user
2. Rating Threshold at 3.5/5
3. Minimum of 5 positive test-set items

This helped us to ensure sufficient data availability and stable evaluation metrics.
We also encountered an issue where a global temporal split caused 82% of the users to become cold-start users - they had no ratings in the training period. To fix this, we used a per-user temporal split, using the first 80% of each user's ratings for training and the remaining 20% for testing. This helped us to preserve the chronological order while eliminating the cold-start issues.

3.3 Scalability and Performance Metrics

Throughput and Latency:

We processed a total of 14M events due to our filtering mechanisms. Initially, our batch processing time was approximately 500 seconds, causing a growing lag. After implementing "Redis Pipelining" (batching 5,000 commands per network round trip) and separating the similarity computation into a background thread, we achieved significant improvements.

Metric	Pre-Optimization	Post-Optimization
Ingestion Rate	~2,000 events/sec	~10,000 events/sec
Micro-Batch Duration	500 seconds	10 - 20 seconds
Serving Latency	5ms	0.21ms (median)
Serving Throughput	500 req/sec	4,485 req/sec

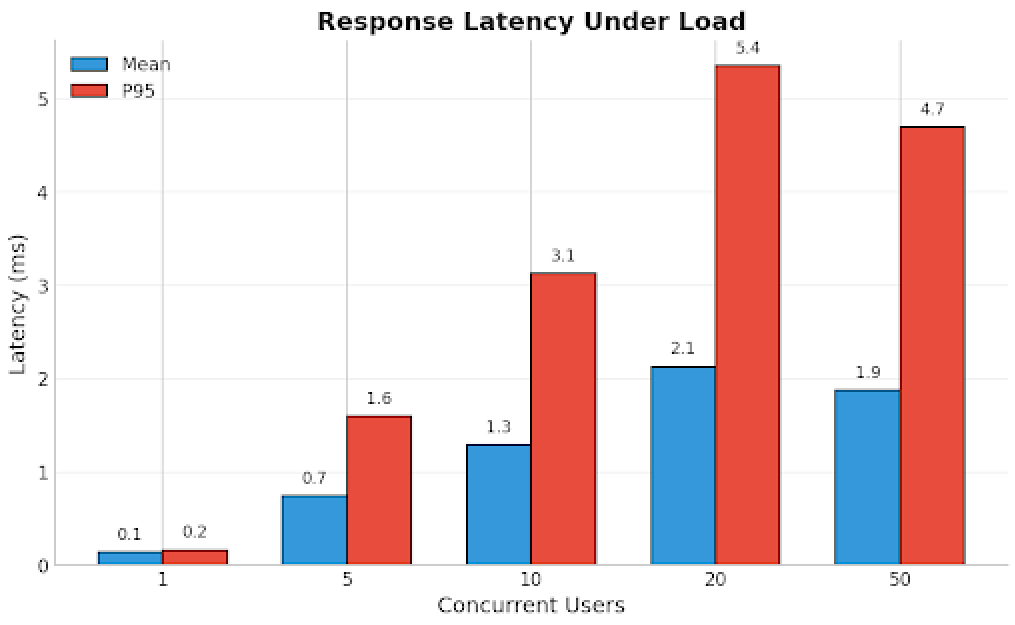


Figure 2: Latency vs. Concurrent Users Plot

Resource Utilization:

The system stabilized at approximately 200,000 keys in Redis across 3 shards. Spark memory usage remained stable due to the aggressive cleanup of state using watermarking (set to a 1-day tolerance for late data).

3.4 Feature Metrics (Quality)

We evaluated the recommendation quality using standard Information Retrieval metrics. It is important to note that pure Collaborative Filtering often trades precision for coverage.

- **Precision@10:** 0.089 (Almost 9% of top-10 recommendations were actually watched/rated highly by the user in the near future).
- **Recall@10:** 0.044 (On average across all users or evaluations, the engine successfully found a relevant item within its top 10 recommendations 4.4% of the time)
- **Inter-User Similarity:** 0.14 (Indicating a low degree of shared preferences between two users)

Additionally, we computed some other commonly utilised metrics:

- **Hit Rate:** 0.456 (In 45.6% of cases, the target movie appeared in the recommendation list).
- **Catalog Coverage:** 99% (The system is not just recommending popular items; the "long tail" is well-represented).

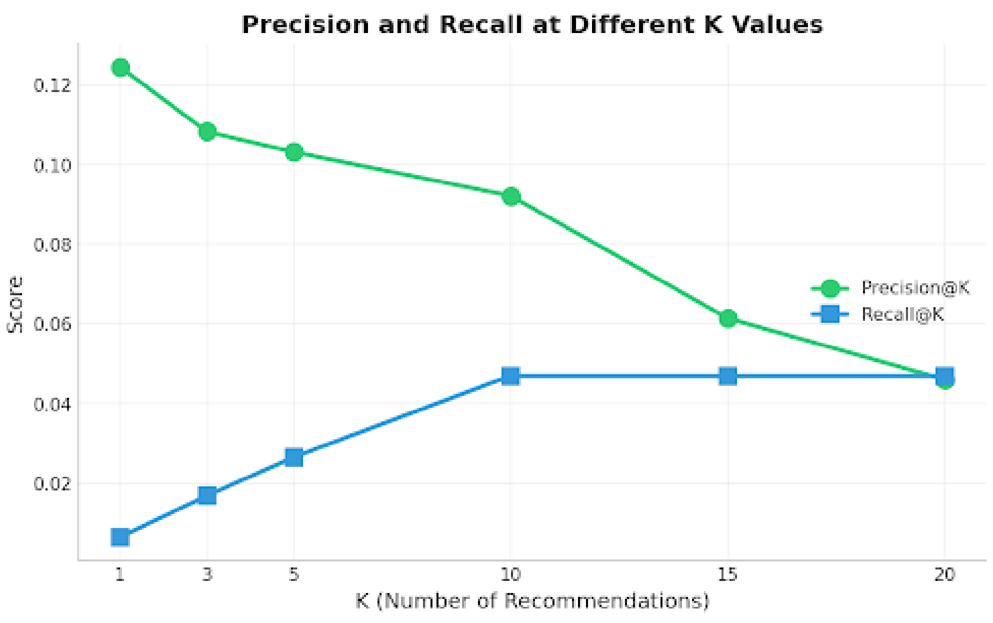


Figure 3: Precision@K and Recall@K Curves

Here we see the classic precision-recall tradeoff - as K increases, precision drops and recall rises. The precision decreases because more recommendations lead to a higher chance of irrelevant items, while the recall increases and plateaus after K=10, as adding more recommendations can no longer find new items.

3.5 Analysis

We fell short of our targets in terms of the accuracy metrics for Precision@K and Recall@K. We attribute this to the simplicity of the model and the lack of more expressive signals such as movie metadata, genres, or embeddings. However the desirably low inter-user similarity scores show that the system successfully avoided collapsing into popularity-based recommendations.

The results demonstrate a successful trade-off between latency and accuracy. While sophisticated deep learning models (like Neural Collaborative Filtering) might achieve higher precision, they cannot easily support the sub-millisecond serving latency and real-time updates we achieved.

Separating the jobs was an important optimization. By calculating similarities in a background job (Job B) rather than inside the streaming micro-batch, we decoupled the expensive aggregation logic from the ingestion logic. This allowed the stream to consume data as fast as Kafka could deliver it. In the graph below for batch-processing time per job, Job A dominates, since [air generation is $O(n^2)$ per session, making it the bottleneck. Job B runs asynchronously and doesn't block the pipeline, and aggregation (Job C) is fast.

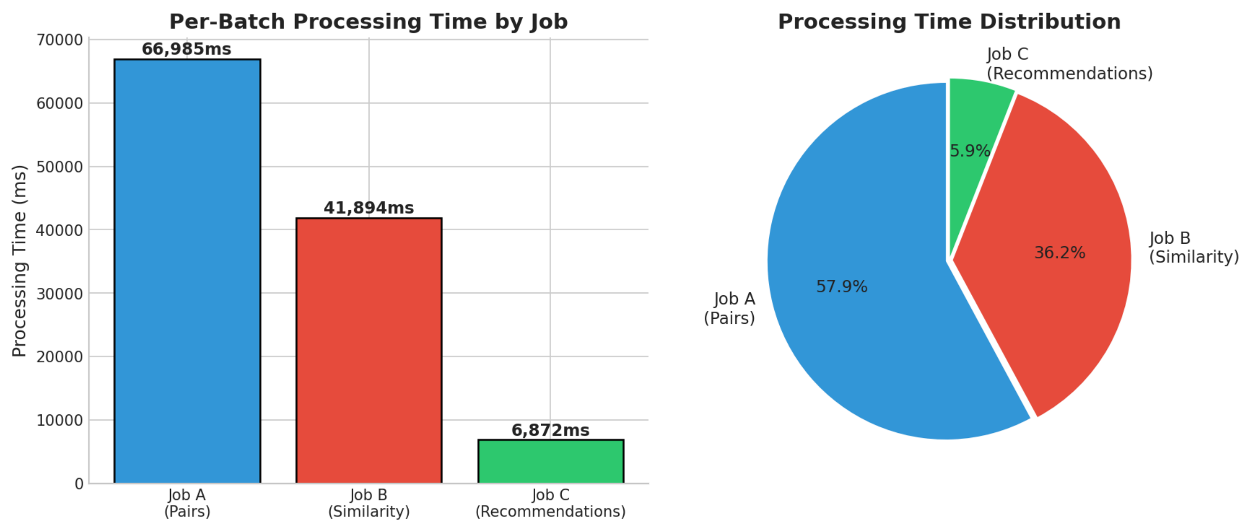


Figure 4: Per-Batch Processing Time

4. Summary

Metric	Target	Achieved
Throughput	10,000 events/sec	~5,000 events/sec
Processing Time	< 2 hours	~50 minutes
Serving Latency	< 100ms	0.21ms median
Serving Throughput	No target	4,485 req/sec
Precision@10	>= 0.25	0.089
Recall@10	>= 0.20	0.044
Inter-user Similarity	<= 0.30	0.140

4.1 Achievement of Design Goals

We successfully built and deployed a Kappa Architecture that met the latency and scalability requirements.

1. **Latency:** We achieved the <100ms goal, achieving 0.21ms median serving time via Redis.
2. **Scalability:** The system handled the full 20M dataset. The use of Redis Cluster means we can scale storage linearly by adding shards.
3. **Real-Time:** Recommendations are updated in near real-time. A user rating a movie influences the model within the next micro-batch window.

4.2 Comparison with Proposal

Our initial proposal suggested a simple sliding window approach. However, during implementation, we realized that strict windowing discarded valuable long-term user history. We adapted the design to use hybrid storage: keeping long-term counts in Redis while processing short-term interactions in Spark. This hybrid approach was more robust than the original proposal.

4.3 Potential Future Extensions

- **Hybrid Filtering:** Incorporating movie metadata (genres, actors) to solve the "Cold Start" problem for new items.
- **Approximate Nearest Neighbors (ANN):** As the item space grows beyond 100k items, exact K-NN search in Redis becomes expensive. Integrating a vector database or using Redis vector similarity search would improve scalability.
- **A/B Testing Framework:** Implementing a diversion layer to route live traffic to different algorithm versions to measure real-world engagement lift.