VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Big Data

Assignment Report

Personalized News Recommender System using Elasticsearch and Kafka

---

Advisor(s):	Assoc. Prof. Thoại Nam	
Student(s):	Nguyễn Ngọc Song Thương	2252803
	Trần Tuấn Kiệt	2252410
	Nguyễn Phúc Thanh Danh	2252102

HO CHI MINH CITY, JUNE 2026

Member List

No.	Full Name	Student ID	Contribution
1	Nguyễn Ngọc Song Thương	2252803	100%
2	Trần Tuấn Kiệt	2252410	100%
3	Nguyễn Phúc Thanh Danh	2252102	100%

Contents

List of Figures

List of Tables

1 Introduction

1.1 Overview

News recommendation systems have become essential for modern digital publishers facing the challenge of information overload. With thousands of articles published daily, users struggle to find content that matches their interests, while publishers need to maximize engagement and reader satisfaction. Traditional recommendation approaches often fail to capture the dynamic nature of user interests and the real-time requirements of news consumption.

The emergence of streaming data platforms and distributed computing has enabled new approaches to recommendation systems. Unlike static batch processing, streaming architectures can respond to user behavior in real-time, providing more relevant and timely recommendations.

1.2 Project Objectives

This project aims to build a personalized news recommendation system that:

1. **Trains efficiently on large-scale datasets**: Leveraging lazy loading and embedding caching strategies to handle datasets with millions of user interactions without requiring all data in memory.

2. **Processes streaming data in real-time**: Using Apache Kafka for event streaming and time-window batching to update user session embeddings as users interact with articles.

3. **Provides low-latency recommendations**: Utilizing Elasticsearch for fast KNN (K-Nearest Neighbors) vector search and Redis for caching session states and candidate articles.

4. **Implements a two-stage recommendation pipeline**: Following industry best practices with a candidate generation stage (Session Model) and a personalized reranking stage (Reranker).

1.3 System Overview

The proposed system consists of two main components:

- **Training Pipeline**: Processes historical user behavior data to train neural network models. The Session Model learns to encode user reading sessions into dense vectors, while the Reranker learns to personalize article rankings based on user demographics and session context.
- **Serving Pipeline**: Handles real-time user events through Kafka, updates session embeddings using the trained Session Model, retrieves candidate articles via Elasticsearch KNN search, and reranks them using the Reranker model.

The system is built using PyTorch for deep learning, Elasticsearch for vector storage and similarity search, Redis for session caching, and Apache Kafka for event streaming. This architecture enables horizontal scalability and real-time responsiveness while maintaining recommendation quality.

2 Background

2.1 Recommender Systems

Recommender systems have become an essential component of modern digital platforms, helping users discover relevant content amid information overload. These systems can be broadly categorized into three paradigms: collaborative filtering, content-based filtering, and hybrid approaches[?].

Collaborative filtering methods identify patterns from user-item interactions, recommending items that similar users have engaged with. Content-based methods, in contrast, leverage item features to recommend content similar to what a user has previously consumed. Modern deep learning approaches have enabled hybrid systems that combine both signals, achieving superior performance across various domains[10].

News recommendation presents unique challenges compared to other domains. Articles have short lifespans, requiring systems to handle a constantly refreshing item catalog. User interests in news are highly dynamic, often influenced by current events. Additionally, the cold-start problem is particularly severe as new articles continuously enter the system without any interaction history[9].

Recent advances in neural news recommendation have introduced sophisticated architectures for modeling user behavior. Session-based recommendation using recurrent neural networks has proven effective for capturing sequential patterns in user browsing behavior[4]. The two-tower architecture, popularized by YouTube and other large-scale

platforms, separates candidate retrieval from ranking to achieve both efficiency and personalization at scale[2].

2.2 Elasticsearch for Vector Search

Elasticsearch is a distributed search and analytics engine built on Apache Lucene. Originally designed for full-text search, Elasticsearch has evolved to support dense vector similarity search, making it suitable for embedding-based retrieval in machine learning applications[3].

The platform stores data as JSON documents distributed across multiple shards, enabling horizontal scaling as data volume grows. For vector search, Elasticsearch provides the `dense_vector` field type, which stores high-dimensional embeddings alongside traditional document fields. Similarity search is performed using script score queries that compute cosine similarity, dot product, or Euclidean distance between query vectors and indexed document vectors.

2.2.1 Approximate Nearest Neighbor Search

Exact nearest neighbor search becomes computationally prohibitive as dataset size grows, requiring $O(n)$ comparisons for each query. Approximate Nearest Neighbor (ANN) algorithms trade a small amount of accuracy for dramatic speedups, enabling sub-linear search times over millions of vectors[5].

Two popular ANN indexing strategies are:

- **IVF-Flat (Inverted File Index)**: Partitions the vector space into clusters using k-means. At query time, only vectors in nearby clusters are searched. While memory-efficient, it requires training on representative data and may miss relevant vectors at cluster boundaries.
- **HNSW (Hierarchical Navigable Small World)**: Constructs a multi-layer graph where each node connects to its approximate nearest neighbors. Queries traverse the graph from coarse upper layers to fine-grained lower layers, achieving logarithmic search complexity $O(\log n)$. HNSW offers superior recall-speed trade-offs but requires more memory than IVF-based methods.

2.2.2 HNSW in Elasticsearch

Elasticsearch uses **HNSW** as its default ANN algorithm for `dense_vector` fields, making it well-suited for real-time recommendation workloads. The HNSW index is con-

figured through two key parameters:

- `m`: The number of neighbors each node connects to (default: 16). Higher values improve recall but increase memory usage and indexing time.
- `ef_construction`: The size of the dynamic candidate list during index construction (default: 100). Higher values produce a higher-quality graph at the cost of slower indexing.

In our implementation, we use Elasticsearch's default HNSW configuration to index 768-dimensional contrastive embeddings for over 125,000 articles. At query time, the `ef` parameter controls the search accuracy-speed trade-off. The KNN search retrieves the top-100 candidate articles in milliseconds, enabling real-time candidate generation for the recommendation pipeline.

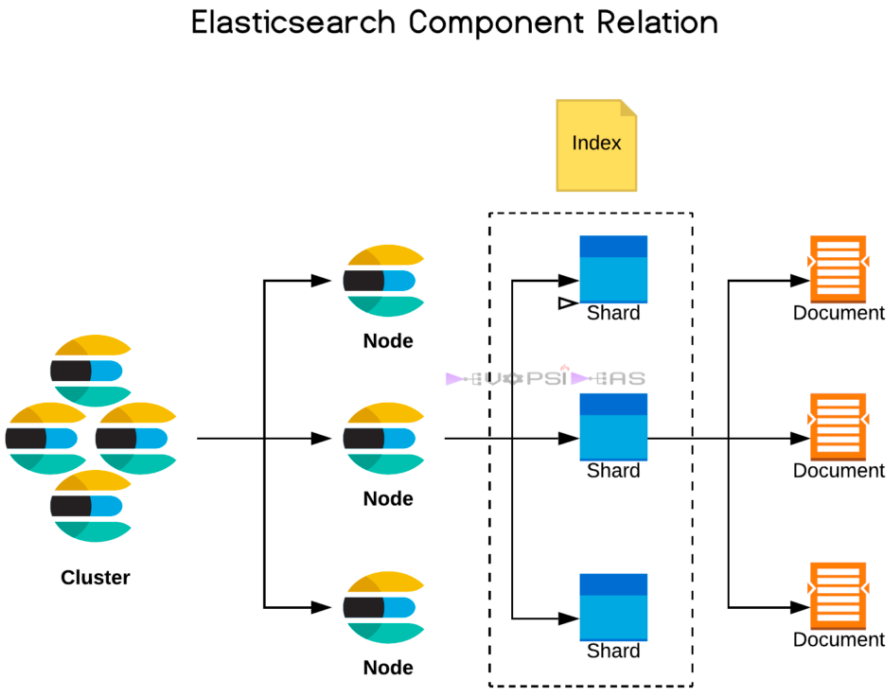

Figure 2.1: Elasticsearch components (source: Internet)

In recommender systems, Elasticsearch serves dual purposes. First, it acts as a feature store for article embeddings and user profiles, providing fast retrieval during both training and inference. Second, it enables approximate nearest neighbor search for candidate generation, efficiently identifying the most similar items to a given query vector from millions

of candidates. The combination of HNSW indexing with Elasticsearch's distributed architecture allows the system to scale horizontally while maintaining sub-millisecond query latencies.

2.3 Apache Kafka for Event Streaming

Apache Kafka is a distributed event streaming platform designed for high-throughput, fault-tolerant data pipelines. Originally developed at LinkedIn to handle their activity stream data, Kafka has become the de facto standard for real-time data integration in modern architectures.

Kafka organizes data into topics, which are partitioned and replicated across a cluster of brokers. Producers publish events to topics, while consumers subscribe to topics and process events in order. This publish-subscribe model decouples data producers from consumers, allowing each component to scale independently. Events are persisted to disk with configurable retention, enabling both real-time processing and historical replay.

For recommendation systems, Kafka enables the transition from batch processing to real-time updates. User interactions such as clicks, reads, and scrolls can be streamed to the recommendation engine as they occur, allowing the system to adapt to changing user interests immediately rather than waiting for the next batch update cycle. This streaming architecture is particularly valuable for news recommendation, where user interests can shift rapidly based on breaking events[7].

The combination of Kafka for event ingestion, Elasticsearch for vector storage and retrieval, and Redis for caching forms a common technology stack for production recommendation systems, providing the low-latency and high-throughput characteristics required for real-time personalization at scale.

3 Dataset

3.1 Overview

This project uses the **Ekstra Bladet News Recommendation Dataset (EB-NeRD)**, a large-scale Danish news recommendation dataset released for the ACM RecSys Challenge 2024[1]. The dataset was collected from user behavior logs at Ekstra Bladet, one of Denmark's largest news publishers, during a 6-week period from April 27th to June 8, 2023.

EB-NeRD provides a comprehensive benchmark for news recommendation research,

Figure 3.1: EB-NeRD: Ekstra Bladet News Recommendation Dataset (Access at: https://recsys.eb.dk/)

capturing real user interactions including clicks, reading time, scroll percentage, and device information. The dataset addresses both technical and normative challenges in designing effective and responsible recommender systems for news publishing.

3.2 Dataset Versions

The dataset is available in three versions to accommodate different computational resources and experimentation needs:

Table 3.1: EB-NeRD Dataset Versions

Version	Users	Size	Purpose
Demo	5,000	20 MB	Quick experimentation
Small	50,000	80 MB	Development and testing
Large	1,000,000+	3.0 GB	Full-scale training

For this project, we train on the **Large** version to demonstrate the system's ability to handle million-scale datasets efficiently.

3.3 Dataset Statistics

The full EB-NeRD dataset encompasses:

- **Users**: Over 1 million unique users

- **Articles**: Over 125,000 Danish news articles
- **Impressions**: More than 37 million impression logs
- **Interactions**: Over 251 million user-article interactions

3.4 Data Structure

Each dataset split (train/validation) contains two main files:

3.4.1 behaviors.parquet

Contains impression-level user behavior data:

Table 3.2: Behaviors Data Schema

Field	Description
impression_id	Unique identifier for each impression
user_id	Anonymized user identifier
impression_time	Timestamp of the impression
article_ids_inview	List of article IDs shown to the user
article_ids_clicked	List of article IDs clicked by the user
read_time	Time spent reading (seconds)
scroll_percentage	How far the user scrolled (0-100)
device_type	Device category (mobile, tablet, desktop)
is_sso_user	Whether user is logged in via SSO
is_subscriber	Whether user has a subscription
gender	User gender (anonymized)
age	User age bucket

3.4.2 history.parquet

Contains aggregated user reading history:

Table 3.3: History Data Schema

Field	Description
user_id	Anonymized user identifier
article_id_fixed	List of previously read article IDs
read_time_fixed	Corresponding reading times
scroll_percentage_fixed	Corresponding scroll percentages
impression_time_fixed	Timestamps of historical reads

3.4.3 articles.parquet

Contains article metadata and content:

Table 3.4: Articles Data Schema

Field	Description
article_id	Unique article identifier
title	Article headline
subtitle	Article subtitle
body	Full article text
category_str	Article category (e.g., news, sports, entertainment)
topics	List of associated topics
published_time	Publication timestamp

3.5 Pre-trained Embeddings

The dataset provides pre-trained 768-dimensional contrastive embeddings for all articles. These embeddings are trained using a contrastive learning approach that captures semantic similarity between articles based on their content. We use these embeddings as input features for both the Session Model and Reranker, rather than training embeddings from scratch.

3.6 News Categories

Articles are categorized into the following groups:

- **NWS** - General News
- **SPT** - Sports
- **ENT** - Entertainment
- **CRM** - Crime
- **PFI** - Personal Finance
- **AGC** - Auto-generated Content
- **MSC** - Miscellaneous

4 Proposed Architecture

The system follows a two-stage recommendation architecture commonly used in industry-scale recommender systems. This approach separates candidate generation from personalized ranking, enabling both efficiency and accuracy at scale.

4.1 Training Pipeline

4.1.1 Model Architecture

The recommendation system employs a two-tower architecture with a reranker, following industry best practices used by companies like Google, Meta, and Netflix[8?].

Session Model (Candidate Generation) The Session Model encodes a user's reading session into a 768-dimensional vector for efficient KNN retrieval. The architecture consists of:

1. **Behavior Fusion Layer**: Combines article embeddings with behavioral features (read time, scroll percentage, position) through a learned transformation.
   - Input: Article embeddings (768-dim) + Behavior features (3-dim)
   - Architecture: Linear(3→64) → ReLU → Linear(64→768)
   - Output: Fused embeddings = Article embeddings + Behavior embeddings

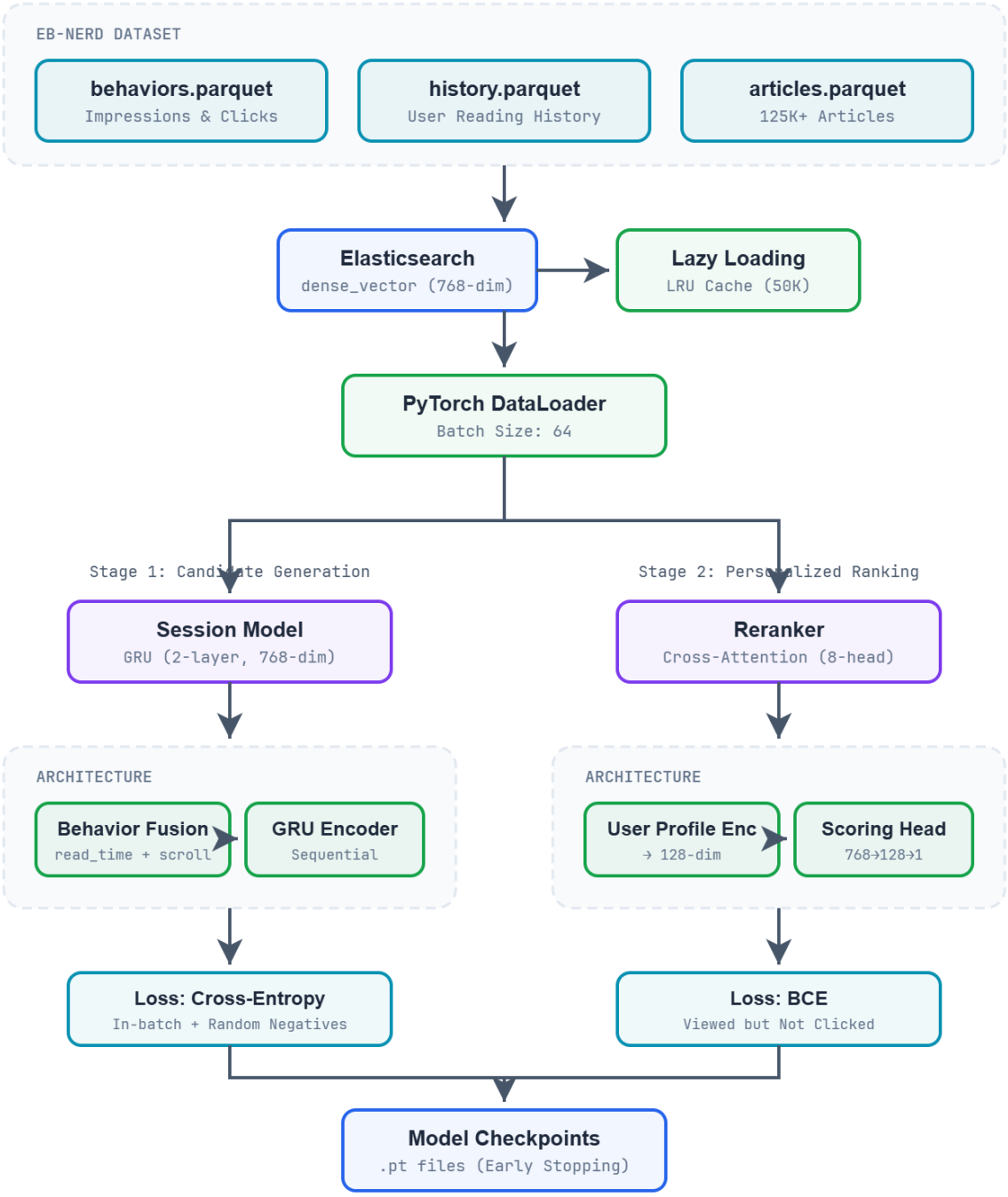

Figure 4.1: Architecture for training pipeline

2. **GRU Encoder**: Processes the sequence of fused embeddings to capture temporal patterns.
    - Architecture: 2-layer GRU with hidden size 768
    - Output: Final hidden state as session embedding (768-dim)

Why GRU over Transformer?

- Sessions are short (5-20 articles) - minimal parallelism benefit from attention
- Supports incremental updates as users read new articles
- O(1) memory complexity vs $O(n^2)$ for self-attention
- Faster inference at scale

Reranker (Personalized Ranking) The Reranker scores and reranks candidate articles using cross-attention between user representation and article embeddings:

1. **User Profile Encoder**: Encodes demographic features into a dense vector.
    - Inputs: device_type, gender, age_bucket, is_sso_user, is_subscriber
    - Embeddings: device(5→16), gender(3→8), age(10→16), binary features
    - Architecture: Concat → Linear(42→64) → ReLU → Linear(64→128)
2. **User Query Generation**: Combines user profile with session embedding.
    - Concat([user_profile(128), session_embedding(768)]) → (896)
    - LayerNorm(896) → Linear(896→768) → user_query
3. **Cross-Attention**: Attends to candidate article embeddings.
    - 8-head attention with embed_dim=768
    - Query: user_query, Key/Value: candidate embeddings
4. **Scoring Head**: Produces relevance scores for each candidate.
    - Interaction: user_query * candidates (element-wise)
    - Linear(768→128) → ReLU → Linear(128→1)

4.1.2 Training Objectives

Table 4.1: Training Objectives for Each Model

Model	Loss Function	Negative Sampling
Session Model	Cross-entropy	In-batch + random negatives
Reranker	Binary cross-entropy	Viewed but not clicked articles

4.1.3 Efficient Data Loading

Training on million-scale datasets requires careful memory management. We implement:

Lazy Loading from Elasticsearch Instead of loading all article embeddings into memory, we query Elasticsearch on-demand during training:

- Embeddings are fetched in batches as needed
- LRU cache (50,000 entries) reduces redundant queries
- Supports datasets larger than available RAM

Embedding Caching Strategy

Listing 4.1: Elasticsearch Embedding Provider

```
class ElasticsearchEmbeddingProvider:
    def __init__(self, host, port, cache_size=50000):
        self._client = ElasticsearchClient(host, port)
        self._cache = LRUCache(maxsize=cache_size)

    def get_embeddings(self, article_ids):
        # Check cache first
        cached = {id: self._cache[id] for id in article_ids
                  if id in self._cache}
        missing = [id for id in article_ids if id not in
           cached]
```

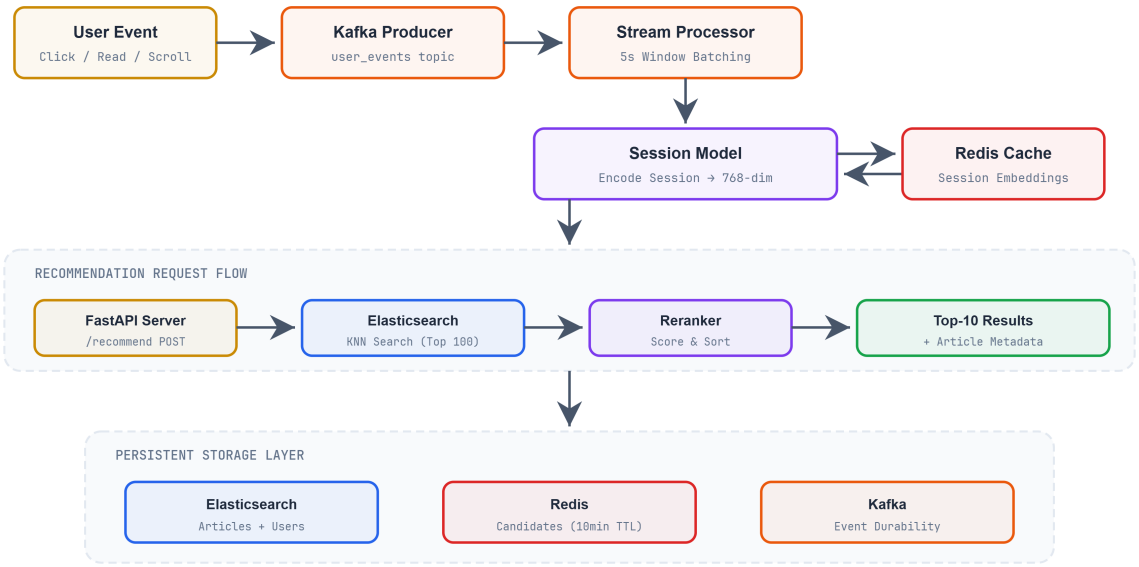

Figure 4.2: Architecture for Serving the Recommender System

```
11
12          # Fetch missing from Elasticsearch
13          if missing:
14              fetched = self._client.get_embeddings_batch(
                    missing)
15              for id, emb in fetched.items():
16                  self._cache[id] = emb
17              cached.update(fetched)
18
19          return cached
```

4.2 Serving Pipeline

4.2.1 System Components

The serving architecture consists of four main components:

1. **Kafka Producer (Event Simulator)**: Publishes user behavior events to Kafka topic.

2. **Stream Processor (Kafka Consumer)**: Consumes events, updates session embeddings.

3. **FastAPI Server**: Handles recommendation requests, orchestrates the pipeline.

4. **Storage Layer**: Elasticsearch for vectors, Redis for session cache.

4.2.2 Streaming Data Format

User behavior events are published to Kafka in JSON format:

Listing 4.2: Streaming Event Format

```
{
    "user_id": "123456",
    "article_id": 9876543,
    "read_time": 45.2,   # seconds
    "scroll_pct": 85.0, # 0-100
    "device_type": 1,    # 0=unknown, 1=mobile, 2=tablet, 3=
       desktop
    "timestamp": "2024-01-01T12:00:00"
}
```

4.2.3 Stream Processing

The Stream Processor implements time-window batching to balance latency with processing efficiency. Events are accumulated over a configurable time window, defaulting to 5 seconds. During this window, the processor continuously polls the Kafka consumer for new messages, collecting all incoming user behavior events into a batch.

Once the time window closes, the processor groups the accumulated events by user ID and updates each user's session history in Redis. For each affected user, the system computes a new session embedding using the trained Session Model, incorporating the latest reading behavior. Finally, the updated session embeddings are stored back in Redis, and any cached candidate lists for those users are invalidated to ensure fresh recommendations on the next request.

Listing 4.3: Time-Window Batch Processing

```
while True:
    # Accumulate events over window
    window_start = time.time()
    batch_events = []

```

```
    while (time.time() - window_start) < window_seconds:
        messages = consumer.poll(timeout_ms=500)
        for record in messages:
            batch_events.append(record.value)

    # Process batch
    if batch_events:
        user_events = group_by_user(batch_events)
        for user_id, events in user_events.items():
            update_session_history(user_id, events)
            compute_session_embedding(user_id)
```

4.2.4 Recommendation Flow

When a user requests recommendations, the system orchestrates several components to generate personalized results. The process begins by retrieving the user's session embedding from the Redis cache, which represents their recent reading behavior as a dense vector.

Using this session embedding, the system performs a KNN search in Elasticsearch to identify the top 100 most similar articles based on cosine similarity. These candidates are cached in Redis for 10 minutes to avoid redundant searches for subsequent requests from the same user. In parallel, the system fetches the user's demographic information from Elasticsearch or Redis, including attributes such as device type, age, gender, and subscription status.

With both the candidate articles and user demographics available, the Reranker model scores each candidate based on the combined context of session behavior and user profile. The candidates are sorted by their reranking scores, and the top 10 articles are returned to the user along with their metadata, completing the recommendation request.

4.2.5 API Endpoints

Table 4.2: REST API Endpoints

Endpoint	Method	Description
/health	GET	Service health check
/users/active	GET	List users with active sessions
/demo/{user_id}	GET	Get recommendations with history
/recommend	POST	Get personalized recommendations
/event	POST	Record user reading event

4.2.6 Role of Apache Kafka

Apache Kafka serves as the backbone for real-time event streaming in our recommendation system. By decoupling event producers from consumers, Kafka allows the user activity tracking components to operate independently from the recommendation engine. This separation enables each component to scale and be deployed independently, simplifying system maintenance and evolution.

Kafka provides strong durability guarantees by persisting all events to disk, ensuring that user behavior data survives service restarts without loss. The platform's partitioning mechanism enables parallel processing across multiple consumers, allowing the system to scale horizontally as event volume grows. Within each partition, Kafka maintains strict event ordering, which is essential for consistent session updates where the sequence of user actions matters.

The system uses a single topic (`user_events`) with automatic partition assignment. Each Stream Processor instance operates with a unique consumer group ID to avoid offset conflicts during development and testing.

5 Evaluation

5.1 Evaluation Metrics

We evaluate the recommendation quality using standard ranking metrics commonly adopted in news recommendation research and the RecSys Challenge 2024[6].

5.1.1 NDCG@K (Normalized Discounted Cumulative Gain)

NDCG@K measures ranking quality by considering both relevance and position, with higher positions weighted more heavily. For a ranked list of K recommendations, NDCG@K is defined as:

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{rel_i}{\log_2(i+1)} \tag{5.1}$$

$$\text{NDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}} \tag{5.2}$$

where $rel_i \in \{0, 1\}$ indicates whether the article at position i is relevant (clicked), and IDCG@K is the ideal DCG computed from the perfect ranking. The logarithmic discount ensures that relevant articles appearing at higher positions contribute more to the score. This metric is particularly suitable for news recommendation where the order of presented articles significantly impacts user engagement.

5.1.2 Hit Rate@K

Hit Rate@K, also known as Recall@K, represents the proportion of test cases where at least one relevant article appears in the top-K recommendations:

$$\text{Hit Rate@K} = \frac{1}{|U|} \sum_{u \in U} \mathbb{1}\left(\sum_{i=1}^{K} rel_{u,i} > 0 \right) \tag{5.3}$$

where U is the set of all test users, $rel_{u,i}$ indicates relevance of the i-th recommendation for user u, and $\mathbb{1}(\cdot)$ is the indicator function. This metric captures the system's ability to include relevant content within the recommendation list.

5.1.3 MRR (Mean Reciprocal Rank)

MRR computes the average of reciprocal ranks of the first relevant article across all test cases:

$$\text{MRR} = \frac{1}{|U|} \sum_{u \in U} \frac{1}{\text{rank}_u} \tag{5.4}$$

where rank_u is the position of the first relevant article in the recommendation list for user u. If no relevant article appears in the list, the reciprocal rank is zero. This metric emphasizes the importance of ranking the most relevant article as high as possible,

reflecting user behavior where attention concentrates on top positions.

5.2 Experimental Setup

5.2.1 Hardware and Software Environment

All experiments were conducted on a workstation with the specifications detailed in Table 5.1.

Table 5.1: Experimental Environment Specifications

Component	**Specification**
	Hardware
CPU	Intel Core i5-14600K (14 cores, 20 threads)
RAM	64 GB DDR5
GPU	NVIDIA GeForce RTX 5070 Ti (16 GB VRAM)
	Software
Operating System	Ubuntu 24.04 LTS (WSL2)
Python	3.11.13
PyTorch	2.1.0 with CUDA 12.1
Elasticsearch	8.11.0
Redis	7.2
Apache Kafka	3.6.0

5.2.2 Training Configuration

Models were trained on the EB-NeRD large dataset with 25,356 validation samples. Training utilized mixed-precision (FP16) on the GPU with the hyperparameters specified in Table 5.2. Early stopping with patience of 5 epochs was applied to prevent overfitting.

Table 5.2: Training Configuration

Parameter	**Value**
Batch Size	64
Learning Rate	0.001
Optimizer	Adam
Max Epochs	100
Early Stopping Patience	5 epochs
Embedding Dimension	768
GRU Hidden Size	768
GRU Layers	2
Dropout	0.1
Negative Samples	10

The Session Model converged after 34 epochs (7.3 minutes), while the Reranker converged after only 7 epochs (54.2 seconds), demonstrating efficient training enabled by the pre-trained contrastive embeddings and GPU acceleration.

5.2.3 Session Model Performance

The Session Model generates session embeddings for KNN-based candidate retrieval from Elasticsearch. Training converged after 34 epochs with early stopping, achieving a final loss of 2.3526. Table 5.3 presents the evaluation results on the validation set.

Table 5.3: Session Model Evaluation Results (Validation Set)

Checkpoint	**NDCG@10**	**Hit Rate@10**	**MRR**	**Samples**
session_model_best	0.4399	0.8498	0.3239	25,356

The Session Model achieves a Hit Rate@10 of 84.98%, demonstrating strong recall in retrieving relevant candidates. However, NDCG@10 of 0.4399 and MRR of 0.3239 reveal a limitation: while the model successfully identifies relevant articles within the candidate

set, it struggles to rank them optimally. This gap between recall and ranking quality motivates the need for a dedicated reranking stage.

5.2.4 Reranker Performance

The Reranker addresses the ranking limitations of the Session Model by incorporating additional signals unavailable during embedding-based retrieval. Training converged after only 7 epochs, achieving a loss of 0.2902. Table 5.4 demonstrates substantial improvements across all metrics.

Table 5.4: Reranker evaluation results (Validation set)

Checkpoint	NDCG@10	Hit Rate@10	MRR	Samples
reranker_best	0.4709	0.8763	0.3538	25,356

5.2.5 Two-Stage Ranking Improvements

Table 5.5 quantifies the improvements achieved by adding the Reranker after Session Model retrieval, demonstrating why a two-stage approach outperforms retrieval alone.

Table 5.5: Improvement from Reranking

Metric	Session Model	With Reranker	Relative Gain
NDCG@10	0.4399	0.4709	+7.0%
Hit Rate@10	0.8498	0.8763	+3.1%
MRR	0.3239	0.3538	+9.2%

The results reveal a clear pattern: while the Session Model excels at candidate retrieval (high Hit Rate), the Reranker significantly improves ranking quality (NDCG and MRR). The 9.2% improvement in MRR is particularly notable, indicating that the Reranker is effective at promoting the most relevant article to higher positions. This is critical for news recommendation, where users typically focus on the first few items.

Why sequential modeling alone is insufficient The Session Model captures sequential reading patterns through its GRU encoder, learning which articles tend to follow others in user sessions. However, this approach has inherent limitations:

1. **Fixed embedding space**: KNN retrieval operates in a fixed embedding space where similarity is computed purely from article content and session patterns. It cannot dynamically adjust rankings based on user-specific context.

2. **No user personalization**: The Session Model treats all users identically given the same reading history. A 25-year-old mobile user and a 50-year-old desktop subscriber with identical sessions receive the same candidates in the same order.

3. **Coarse-grained scoring**: Cosine similarity provides a single similarity score without considering nuanced interactions between user preferences and article features.

What the Reranker adds The Reranker overcomes these limitations by introducing:

1. **User demographics**: Device type, age, gender, and subscription status enable personalized ranking. Mobile users may prefer shorter articles; subscribers may value premium content.

2. **Cross-attention**: The attention mechanism allows the model to learn complex interactions between user context and article features, going beyond simple similarity matching.

3. **Fine-grained scoring**: Each candidate receives a dedicated relevance score computed from the full user-article interaction, enabling precise reordering.

The 7.0% NDCG improvement and 9.2% MRR improvement validate that these additional signals provide meaningful personalization that pure sequential modeling cannot achieve.

5.2.6 Full Pipeline Performance

Table 5.6 presents the end-to-end evaluation of the complete two-stage pipeline.

Table 5.6: Full pipeline evaluation results (Validation set)

Configuration	NDCG@10	Hit Rate@10	MRR	Samples
Session + Reranker	0.4684	0.8755	0.3508	25,356

5.2.7 Inference Performance

Despite adding a second model, the pipeline maintains real-time latency. Table 5.7 shows that the Reranker adds minimal overhead.

Table 5.7: Inference time statistics (25,356 samples)

Component	Total Time	Avg per Sample
Session Model	47.8 sec	1.89 ms
Reranker	12.5 sec	0.49 ms
Full Pipeline	48.6 sec	1.92 ms

The Reranker operates at 0.49ms per sample—adding less than 0.5ms to the pipeline while delivering 7-9% metric improvements. This favorable accuracy-latency trade-off justifies the two-stage design for production systems where both quality and responsiveness matter.

5.3 Analysis

5.3.1 Why Two Stages beat one

The experimental results demonstrate a fundamental insight: retrieval and ranking are complementary but distinct tasks that benefit from specialized models.

The Session Model optimizes for *recall*—ensuring relevant articles appear somewhere in the candidate set. Its GRU encoder learns temporal patterns like "users who read sports articles tend to read more sports," enabling efficient embedding-based retrieval. With Hit Rate@10 of 84.98%, it successfully narrows 125,000+ articles to 100 high-quality candidates.

The Reranker optimizes for *precision*—ensuring the most relevant articles appear at the top. Its cross-attention mechanism learns fine-grained user-article interactions like "young mobile users prefer this article format over that one." The 9.2% MRR improvement shows it effectively identifies the single best article for each user.

Attempting to solve both tasks with a single model would require either:

- Running expensive cross-attention over all 125,000 articles (computationally infeasible), or
- Sacrificing personalization depth to maintain retrieval efficiency

The two-stage design achieves the best of both worlds: broad recall from efficient embedding search, followed by deep personalization from focused reranking.

5.3.2 The value of user features

The Reranker's 7.0% NDCG improvement stems primarily from incorporating user demographics that the Session Model cannot access. These features enable:

- **Device-Aware Ranking**: Mobile users may prefer concise articles suitable for on-the-go reading, while desktop users may engage more with long-form content.
- **Age-Appropriate Content**: Reading preferences vary across age groups, from topic selection to writing style preferences.
- **Subscriber Prioritization**: Subscribers represent engaged users who may value premium or exclusive content differently than casual readers.

Without the Reranker, these personalization signals would be entirely absent from the recommendation pipeline, leaving significant value on the table.

6 Demonstration

6.1 Recommender Engine Functional Demonstration

To demonstrate the real-time recommendation pipeline, the system requires four concurrent processes that simulate a production environment. Figure 6.1 illustrates how these components interact to process user events and generate personalized recommendations.

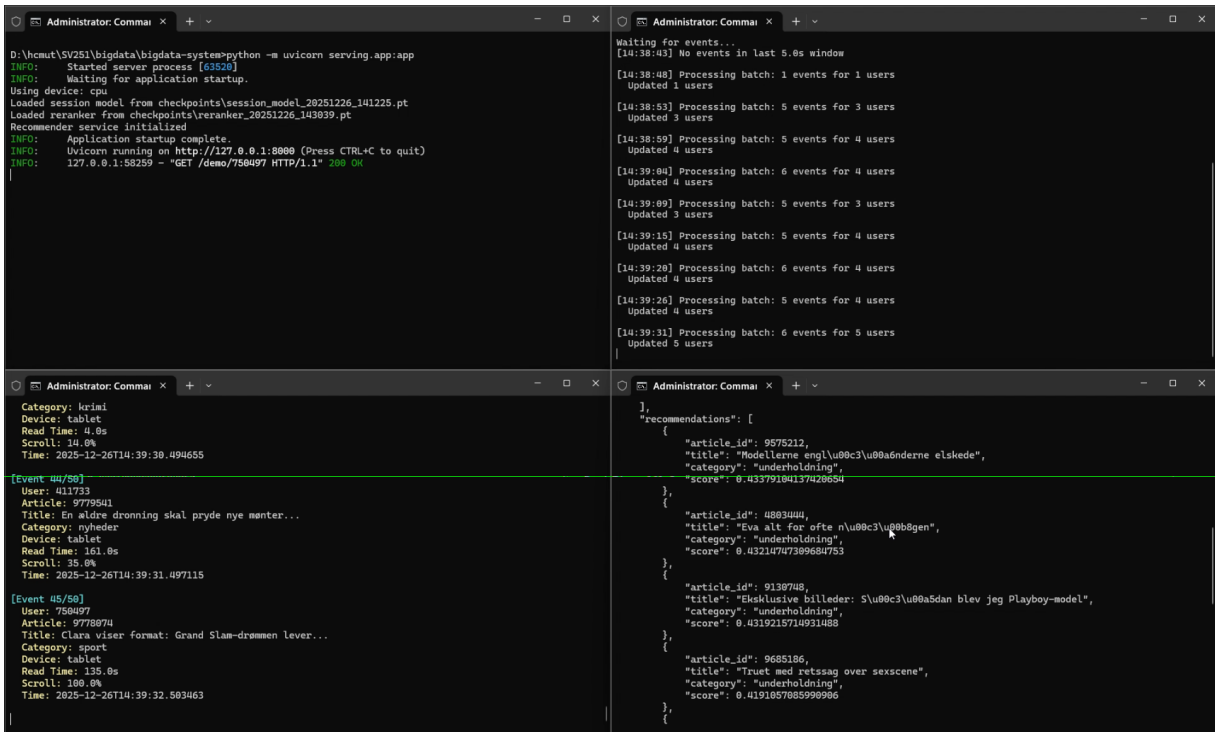

Figure 6.1: Whole demo flow (view the terminal demo video here)

6.1.1 System Components

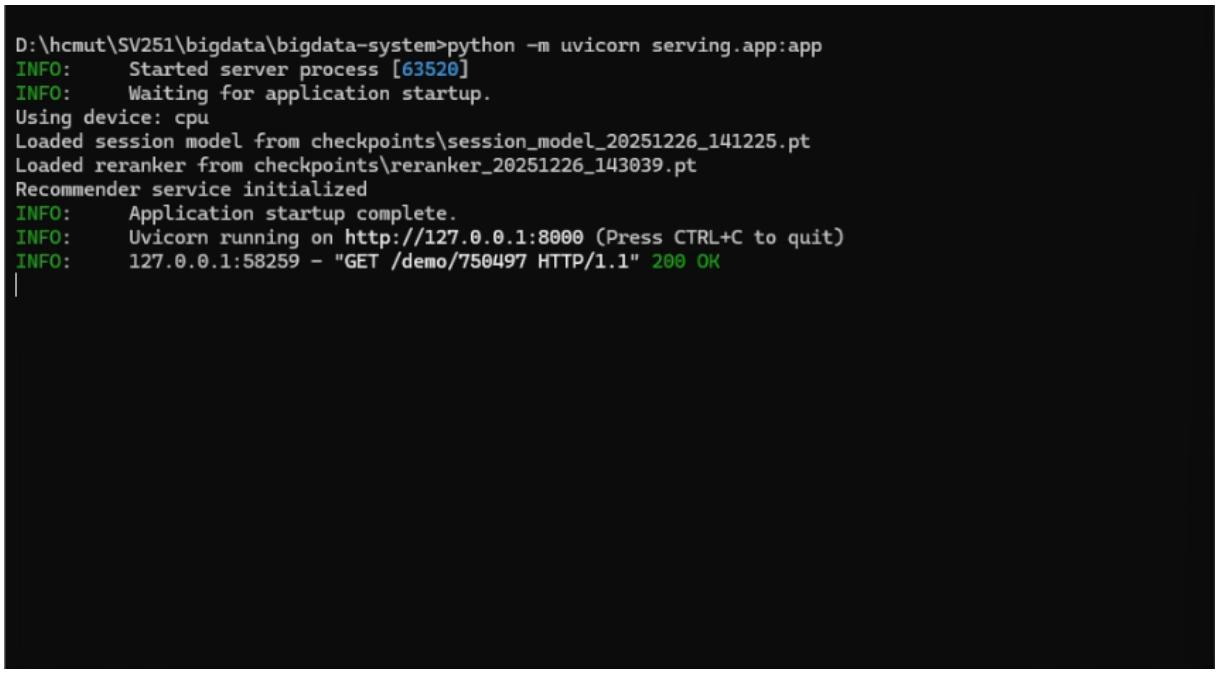

Figure 6.2: Terminal running FastAPI server - a gateway for recommendation

API Server The FastAPI server acts as the central gateway for the recommendation system. Upon startup, it loads the trained Session Model and Reranker checkpoints, initializes connections to Elasticsearch and Redis, and exposes RESTful endpoints for recommendation requests. The server handles three primary responsibilities: serving recommendation requests by orchestrating the two-stage pipeline, providing user session history for debugging and visualization, and exposing health and statistics endpoints for monitoring. The server runs on port 8000 and automatically selects the most recent model checkpoints from the checkpoint directory.

Listing 6.1: Starting the API Server

```
docker-compose up -d  # Start infrastructure services
python -m uvicorn serving.app:app --host 0.0.0.0 --port 8000
```

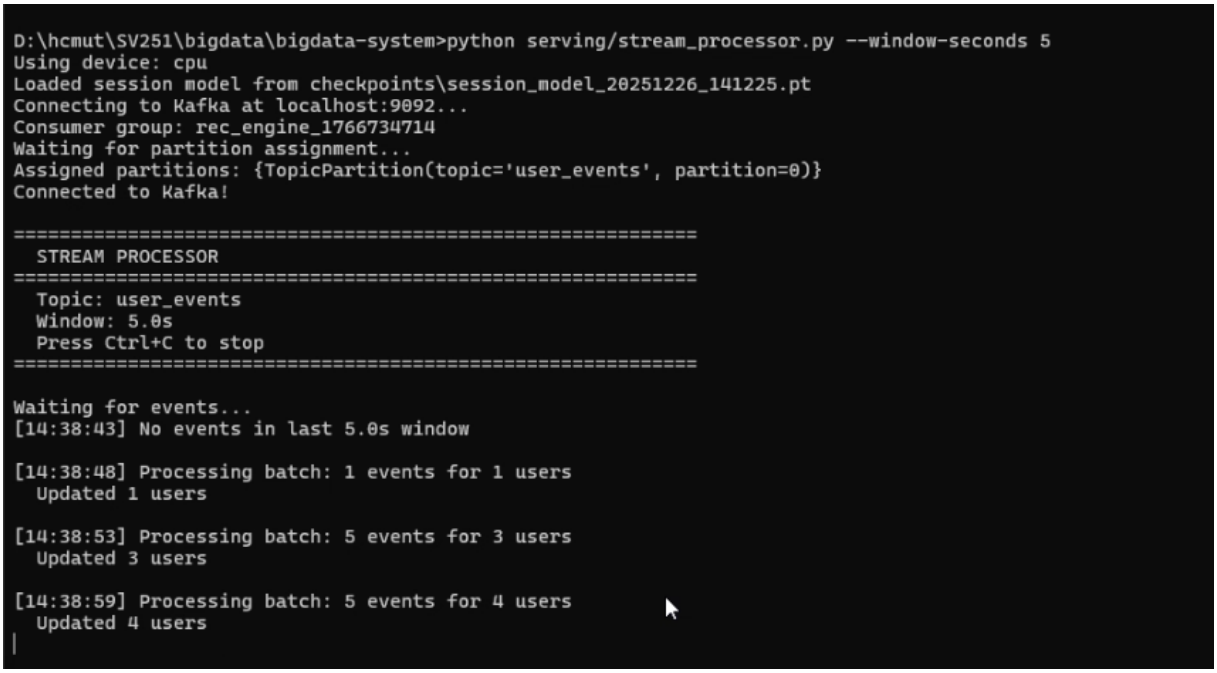

Figure 6.3: Terminal running Stream processor - a Kafka consumer

Stream Processor The Stream Processor consumes user behavior events from Apache Kafka and maintains real-time session state. It implements time-window batching to balance latency with processing efficiency: events are accumulated over a configurable window (default 5 seconds), then processed in batch to update user sessions in Redis. For each batch, the processor groups events by user, appends new reading events to the user's

session history, computes updated session embeddings using the Session Model, and invalidates cached recommendations to ensure freshness. This component enables the system to respond to changing user interests within seconds rather than waiting for batch reprocessing.

Listing 6.2: Starting the Stream Processor

```
python serving/stream_processor.py --window-seconds 5
```

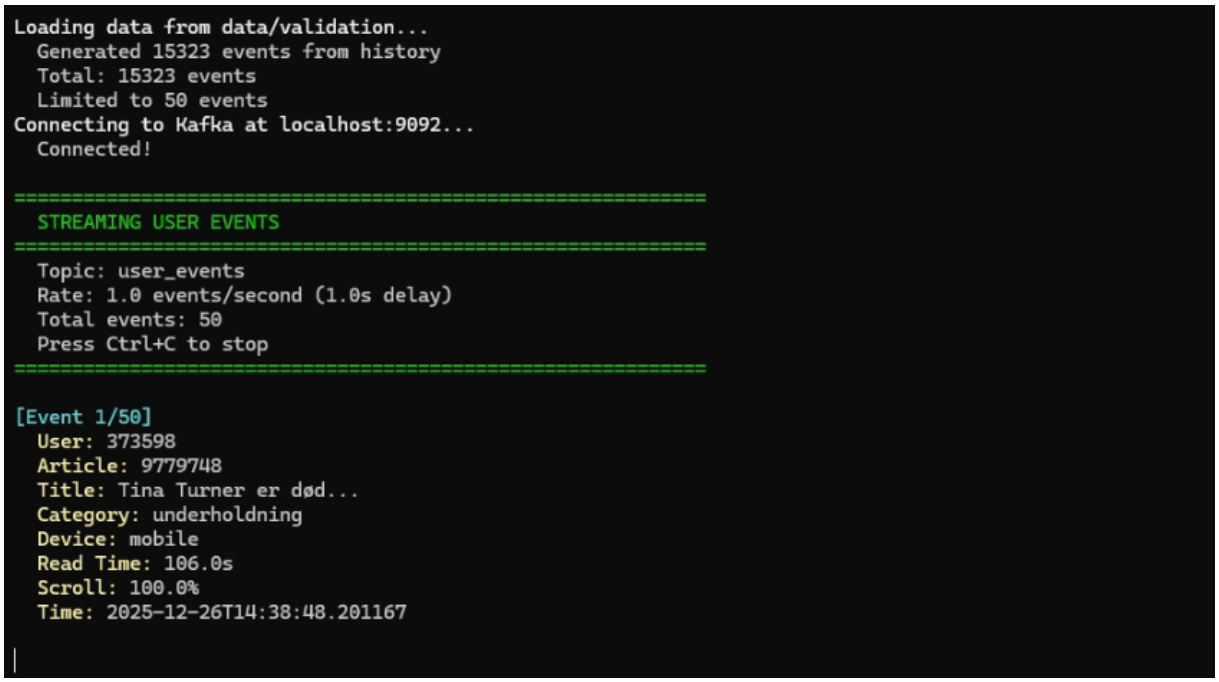

Figure 6.4: Terminal simulating event stream - a Kafka producer

Event Simulator (Terminal 3) For demonstration purposes, the Event Simulator replays historical user behavior from the EB-NeRD dataset through the Kafka event stream. It reads from the validation set's history and behaviors files, extracting article reads with associated engagement metrics (read time, scroll percentage). Events are published to Kafka at a configurable rate, simulating real-time user activity. Each event contains the user ID, article ID, reading duration, and scroll depth, mimicking the telemetry that would be collected from a production news website.

Listing 6.3: Starting the Event Simulator

```
python demo/simulate_stream.py --rate 1 --limit 100
```

Figure 6.5: Terminal acted as client, sending recommendation request

Recommendation Client (Terminal 4) The client terminal demonstrates how applications would interact with the recommendation API. After events have been processed, users can query the system to retrieve active user sessions and request personalized recommendations. The demo endpoint returns both the user's recent reading history (enriched with article titles and categories) and the top-10 recommended articles with relevance scores, providing visibility into how the system interprets user behavior and generates recommendations.

Listing 6.4: Querying Recommendations

```
# List users with active sessions
curl -s http://localhost:8000/users/active | python -m json.tool

# Get recommendations for a specific user
curl -s http://localhost:8000/demo/USER_ID | python -m json.tool
```

6.1.2 End-to-End Data Flow

When a user reads an article, the following sequence occurs:

1. The Event Simulator publishes a reading event to the `user_events` Kafka topic, containing the user ID, article ID, read time, and scroll percentage.

2. The Stream Processor consumes the event and buffers it until the time window closes. When the window expires, it processes all accumulated events as a batch.

3. For each affected user, the processor retrieves their session history from Redis, appends the new events, and computes an updated session embedding using the Session Model's GRU encoder.

4. The new session embedding is stored in Redis, and any cached candidate lists for that user are invalidated.

5. When a recommendation request arrives, the API server retrieves the session embedding from Redis and performs KNN search in Elasticsearch to find the top-100 candidate articles.

6. The Reranker scores each candidate using the session context and user demographics, producing the final ranked list.

7. The top-10 articles are returned to the client with their titles, categories, and relevance scores.

This architecture ensures that recommendations reflect user behavior within seconds of interaction, while the two-stage pipeline maintains computational efficiency suitable for production workloads.

6.2 News Website Interface

To validate the proposed architecture and demonstrate the real-time capabilities of the recommender system, we developed a fully functional web application titled "The Daily Chronicle." The frontend is designed as a minimalist Single Page Application (SPA) to ensure a seamless reading experience while efficiently capturing user behavioral signals (clicks, dwell time, and scroll depth) required for the Session Model.

The following subsections detail the key interfaces and their role in the recommendation pipeline.

6.2.1 Authentication and Onboarding

The entry point to the system is the authentication interface (Figure 6.6). To facilitate immediate evaluation of the system without requiring complex registration steps, the interface includes a "Use Demo Account" feature.

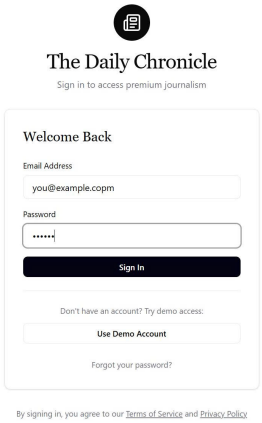

Figure 6.6: The user login interface featuring a demo access option.

Upon successful authentication, the frontend retrieves the unique `user_id`. This identifier is critical as it tags all subsequent Kafka event streams and allows the backend to fetch the specific user embedding from the Redis cache during recommendation requests.

6.2.2 Homepage and Navigation

Figure 6.7 displays the homepage and the global navigation state. The design prioritizes content visibility to reduce cognitive load, ensuring that user interactions are driven by genuine interest rather than UI clutter.

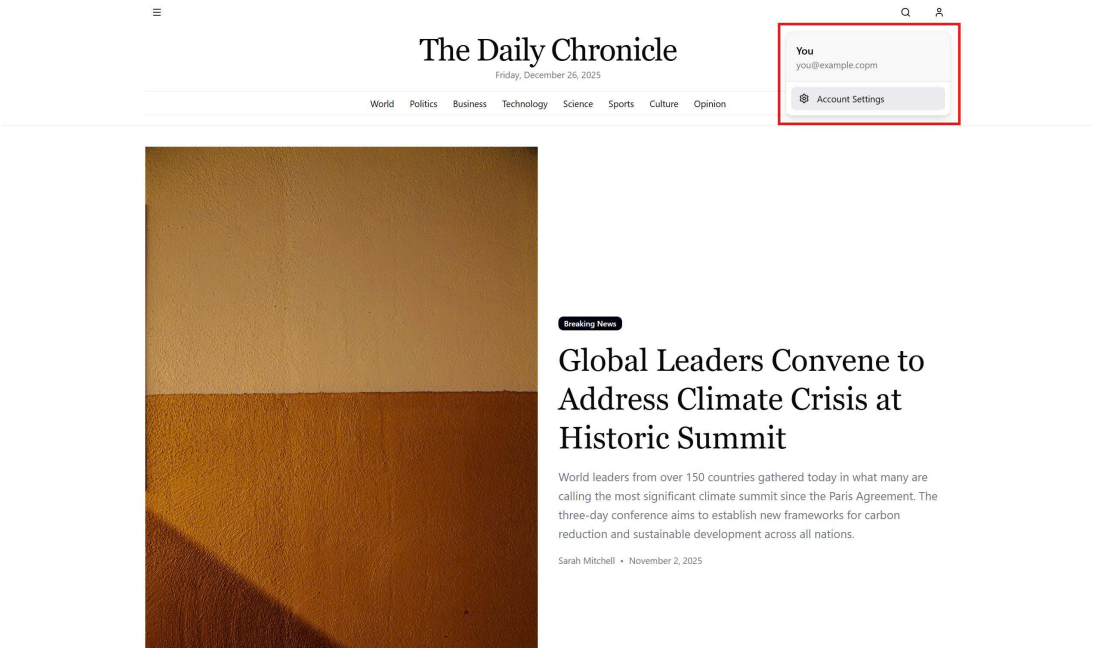

Figure 6.7: Homepage view showing the logged-in user state and account dropdown.

The navigation bar provides access to the user context menu. As shown in the top-right corner of Figure 6.7, the user is authenticated (e.g., *you@example.com*). This persistent session allows the system to track the user's trajectory across different sessions, which is essential for the *history.parquet* data aggregation described in Section 3.4.

6.2.3 Article Reading and Recommendations

The core of the demonstration lies in the article reading view. When a user enters an article (Figure 6.8), the system begins a timer to calculate `read_time` and monitors the viewport to calculate `scroll_percentage`. These metrics are streamed to the Kafka topic `user_events` upon page exit or periodically.

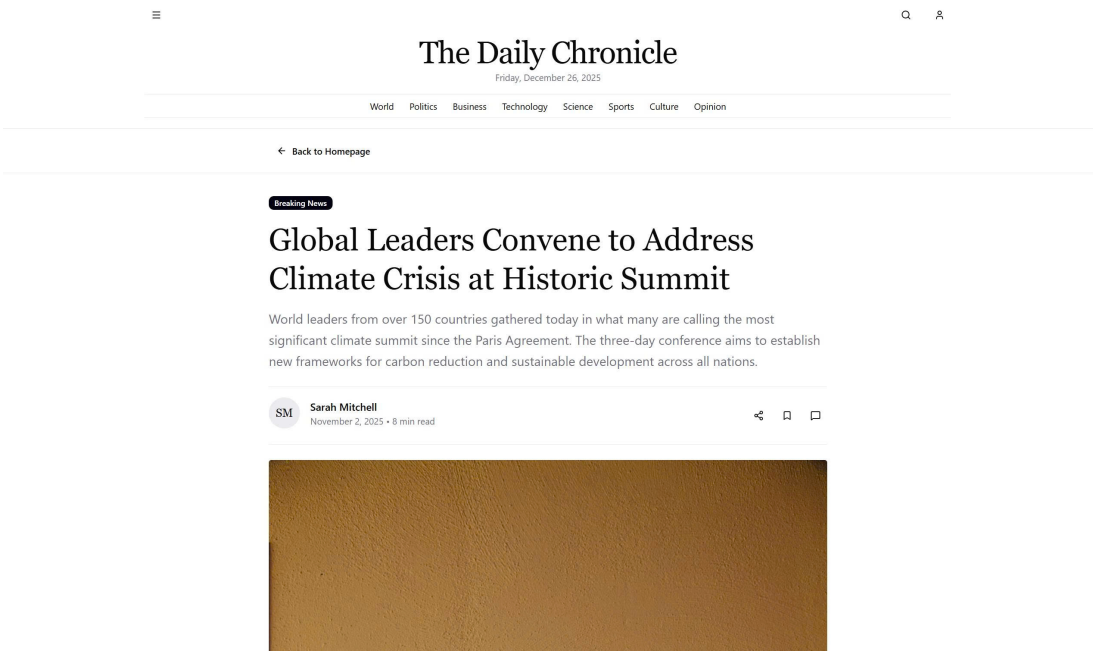

Figure 6.8: The article reading interface.

At the bottom of the article (Figure 6.9), the "Related Stories" section displays the output of our Two-Stage Recommendation Pipeline.

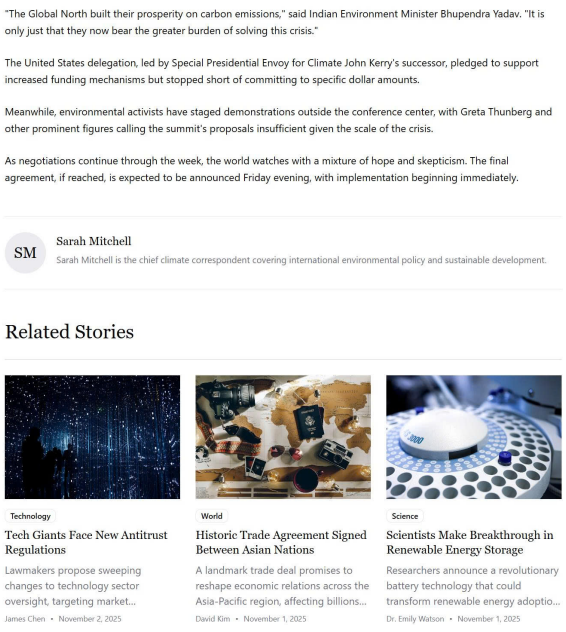

Figure 6.9: Personalized recommendations displayed as "Related Stories" at the end of an article.

Unlike static "latest news" feeds, these three cards are dynamically fetched via the `/recommend` API endpoint. The specific mix of categories shown in Figure 6.9—Technology, World, and Science—reflects the system's ability to cross-reference the current article's context with the user's historical preferences using the Reranker model.

6.2.4 User Profile and Explicit Preferences

Finally, the Account Settings page (Figure 6.10) allows users to manage their explicit preferences.

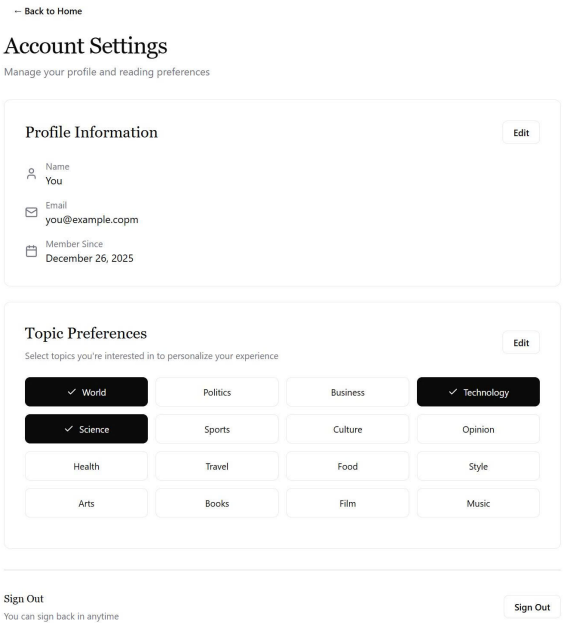

Figure 6.10: Account settings page allowing users to define explicit topic interests.

The "Topic Preferences" section allows users to toggle interests such as World, Science, and Technology. These explicit signals are fed into the **User Profile Encoder** (as detailed in Section 4.1.1) to address the cold-start problem. Even if a user has a limited reading history (short session embedding), the Reranker can utilize these explicit tags to boost relevant articles in the final scoring stage, ensuring immediate personalization.

7 Conclusion

7.1 Summary

This project successfully implemented a personalized news recommendation system using modern big data technologies. At its core, the system employs a two-stage recommendation pipeline that separates candidate generation from personalized ranking, an architectural pattern widely adopted by major technology companies for its balance of efficiency and accuracy at scale.

The first stage utilizes a GRU-based Session Model that captures sequential reading patterns along with behavioral signals such as read time and scroll percentage. This model generates dense session embeddings that enable efficient similarity search across the entire article corpus. The second stage employs a cross-attention Reranker that incorporates user demographics including device type, age, gender, and subscription status to provide deeper personalization beyond what session-based signals alone can offer.

To support real-time recommendations, the system integrates Apache Kafka for event streaming, Elasticsearch for vector similarity search, and Redis for session caching. This infrastructure enables recommendation updates to occur in real-time as users interact with the platform, ensuring that the system remains responsive to changing user interests.

7.2 Role of Big Data Technologies

Elasticsearch serves as the core storage and retrieval engine for our system. Its `dense_vector` field type combined with `script_score` queries enables efficient KNN search over more than 125,000 article embeddings. The distributed architecture of Elasticsearch naturally supports horizontal scaling as the article corpus grows, making it well-suited for production deployments. During training, Elasticsearch enables on-demand embedding retrieval with LRU caching, which allows the system to train on datasets larger than available RAM. Additionally, user profile data is indexed in Elasticsearch for quick lookup during recommendation serving.

Apache Kafka enables real-time event processing throughout the system. User behavior events such as article reads and scroll depth are streamed in real-time to update session embeddings as users browse. By separating event producers from consumers, Kafka enables independent scaling and deployment of different system components. Events are persisted to disk, ensuring no data loss during service restarts. The stream processor batches events over configurable time windows, striking a balance between latency and processing efficiency.

7.3 Results Summary

The evaluation on the EB-NeRD dataset demonstrates the effectiveness of our approach. The Session Model achieves a Hit Rate@10 of 84.98%, successfully retrieving relevant candidates from over 125,000 articles. The Reranker improves ranking quality significantly, achieving NDCG@10 of 0.4709 and Hit Rate@10 of 87.63% on the validation set—representing a 7.0% relative improvement in NDCG and 9.2% improvement in MRR over the Session Model alone.

The full two-stage pipeline achieves NDCG@10 of 0.4684, Hit Rate@10 of 87.55%, and MRR of 0.3508. With an average end-to-end inference time of 1.92 milliseconds per sample (Session Model: 1.89ms, Reranker: 0.49ms), the system comfortably meets real-time serving requirements. The two-stage architecture proves its value by efficiently reducing the candidate space from over 125,000 articles to 100 candidates before the more computationally expensive reranking step.

7.4 Future Work

Several directions could further improve the system. Cold start handling remains an important challenge, where content-based recommendations could serve new users without reading history. Incorporating time-decay factors would help prioritize recent news articles, addressing the freshness requirements inherent to news recommendation. Diversity-aware reranking could balance relevance with variety to improve user experience and content discovery.

From an infrastructure perspective, online learning would enable continuous model updates based on real-time user feedback. A/B testing infrastructure would be essential for production deployment to measure real-world impact of system changes. Finally, extending support to multiple languages would broaden the applicability of the system to international news platforms.

7.5 Lessons Learned

This project yielded several key insights about building recommendation systems at scale. The two-stage architecture proved essential for large-scale recommendation, as separating candidate generation from ranking is crucial for both computational efficiency and recommendation quality. For short sequences like reading sessions, GRU networks provide comparable quality to Transformers while offering faster inference and simpler incremental updates.

On the infrastructure side, Redis caching of session embeddings and candidates significantly reduces redundant computation, particularly for users who request recommendations multiple times within a session. The streaming architecture powered by Kafka enables immediate response to changing user interests, which is particularly valuable in the fast-moving news domain where user preferences can shift rapidly based on current events.

References

[1] Ekstra Bladet and RecSys Challenge. Eb-nerd: A large-scale dataset for news recommendation. In *Proceedings of the 18th ACM Conference on Recommender Systems*. ACM, 2024.

[2] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 191–198. ACM, 2016.

[3] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. O'Reilly Media, 2015.

[4] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. Session-based recommendations with recurrent neural networks. In *International Conference on Learning Representations*, 2016.

[5] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[6] Johannes Kruse, Kasper Lindskow, Anshuk Uppal, Michael Riis Andersen, Jes Frellsen, Marco Polignano, Claudio Pomo, and Abhishek Srivastava. Recsys challenge 2024: Balancing accuracy and editorial values in news recommendations. In *Proceedings of the 18th ACM Conference on Recommender Systems*. ACM, 2024.

[7] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The Definitive Guide*. O'Reilly Media, 2017.

[8] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. In *arXiv preprint arXiv:1906.00091*, 2019.

[9] Fangzhao Wu, Ying Qiao, Jiun-Hung Chen, Chuhan Wu, Tao Qi, Jianxun Lian, Danyang Liu, Xing Xie, Jianfeng Gao, Winnie Wu, and Ming Zhou. Mind: A large-scale dataset for news recommendation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 3597–3606, 2020.

[10] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. Deep learning based recommender system: A survey and new perspectives. *ACM Computing Surveys*, 52(1):1–38, 2019.