Contents

ML Recommendation System Design

Overview

This document outlines the architecture and data flow of a real-time machine learning recommendation system similar to those used by large social media platforms such as Facebook. The system ingests user behavioral events, generates candidate content, ranks it using machine learning models, and delivers a personalized feed to the user.

Components and Flow

1. User Event Ingestion
    - User actions captured: clicks, views, likes, scrolls, watch time.
    - Transport: Kafka pipelines events in real time.
2. Real-Time Stream Processing
    - Flink consumes from Kafka and transforms events for:
        - Feature computation
        - Candidate filtering
        - Metadata enrichment

Data Sources

Feature Store

- Includes data on:
  - Users: age, interests, avg_dwell_time, embeddings
  - Items: category, freshness, engagement_score
  - Context: device_type, time_of_day, language
- Storage: S3 for batch storage, Redis for hot data (e.g., top-ranked candidates)

Metadata Store

- Backed by RDBMS or RocksDB
- Stores social and graph data:
  - Friend relationships (Edge)
  - Group memberships (Edge)
  - NSFW, region, language flags (Node)
  - Follow/block/visibility rules (Edge)

ANN Index (FAISS)

- Built offline using Spark
- Persisted in S3/blob store
- Loaded into in-memory or memory-mapped sharded servers for fast vector similarity search

The ANN Index is functionally similar to a Vector Database (Vector DB). Both store high-dimensional embeddings (such as item or user vectors) and support fast nearest-neighbor search using approximate methods like HNSW, IVF, or PQ. The primary difference is that a Vector DB adds infrastructure and usability on top of ANN indexing: it provides APIs, metadata filtering, persistence, and horizontal scalability. FAISS, in this case, is used as the core ANN engine, loaded into memory for low-latency querying, while a production-grade Vector DB (like Pinecone or Milvus) might be used if richer querying and operational needs arise.

Candidate Generation

Inputs: Feature Store, Metadata, ANN Index, Redis Cache

Responsibilities:
- Pull content based on user signals
- Query FAISS for nearest-neighbor matches

- Merge candidates from multiple sources
- Remove ineligible/blocked items
- Enforce source quotas and diversity rules
- Limit final output to Top-K (e.g., 500–1000 candidates)

Output: Feature-enriched candidate list to Ranking Service

Example features per candidate:
- friend_post, ann_similarity, item_category, item_freshness_hours
- engagement_score, user_dwell_time, device, language, time_of_day

Example Vector Payload to Ranking Service

```json
{
  "user_id": "u_12345",
  "request_context": {
    "device": "iOS",
    "time_of_day": "evening",
    "region": "US",
    "language": "en"
  },
  "candidates": [
    {
      "item_id": "post_001",
      "features": {
        "user_embedding": [0.12, 0.87, -0.33],
        "item_embedding": [0.49, 0.21, 0.73],
        "friend_post": false,
        "same_group": false,
        "ann_similarity": 0.91,
        "item_category": "sports",
        "item_freshness_hours": 1.2,
        "item_engagement_score": 0.78,
        "user_avg_dwell_time": 5.4,
        "device_type": "iOS",
        "time_of_day": "evening",
        "is_hot_item": false
      }
    }
  ]
}
```

Understanding and Training with Embeddings and Features

Modern recommendation systems rely heavily on dense feature vectors that include both raw attributes and learned embeddings. This section explains what embeddings are, how to interpret them, and how to use them for model training.

What Are Embeddings?

Embeddings such as `user_embedding` and `item_embedding` are dense vectors representing users and items in a learned latent space. Each value (e.g., `0.12`) in these vectors corresponds to a coordinate in that space, but **the individual numbers are not human-interpretable**.

Key Points:

- Embeddings are learned during model training to capture behavioral or semantic similarity.
- A value like `0.12` is only meaningful **in relation to other vectors** — their absolute meaning is not interpretable.
- The **position** of a user or item in the embedding space determines its similarity to others.
- Similar vectors imply similar behavior or content preference.

Example:

```
import numpy as np

user_embedding = np.array([0.12, 0.87, -0.33])
item_embedding = np.array([0.49, 0.21, 0.73])

cos_sim = np.dot(user_embedding, item_embedding) / (
    np.linalg.norm(user_embedding) * np.linalg.norm(item_embedding)
)
```

Here, `cos_sim` is the similarity score between the user and item, which can be used directly as a feature in the ranking model.

How to Train the Model with Features

Training a recommendation model requires assembling many labeled data points where each includes:

- A **feature vector** representing the user-item interaction at a specific time.
- A **label** (e.g., `clicked = 1` or `clicked = 0`) that reflects the user's behavior.

1. Constructing the Training Dataset Each row in the training dataset is built by joining:

- **Impression logs** (with `user_id`, `item_id`, `event_time`, and `click` label).
- **User features** as of the impression time (e.g., `user_avg_dwell_time`, `user_embedding`).
- **Item features** as of the impression time (e.g., `engagement_score`, `item_embedding`).
- **Context features** (e.g., `device_type`, `region`, `time_of_day`).

Point-in-time correctness must be enforced during joins to avoid data leakage.

Example Training Row

```
{
  "user_id": "u_123",
  "item_id": "post_456",
  "event_time": "2025-06-15T12:30:00Z",
  "label": 1,
  "features": {
    "user_avg_dwell_time": 5.4,
    "item_engagement_score": 0.78,
    "item_freshness_hours": 1.2,
    "ann_similarity": 0.91,
    "device_type": "iOS",
    "time_of_day": "evening",
    "user_embedding": [0.12, 0.87, -0.33],
    "item_embedding": [0.49, 0.21, 0.73]
  }
}
```

2. Preprocessing

- Normalize numerical features (e.g., z-score or min-max).
- Encode categorical features (e.g., one-hot, embeddings).
- Optionally bucket or clip values to reduce outliers.

3. Model Training Feed the features and labels into a machine learning model:

```
X = [vectorized features]
y = [click label]

# Example with XGBoost
import xgboost as xgb
model = xgb.XGBClassifier()
model.fit(X, y)
```

Other options: - Logistic regression (fast, interpretable) - Deep neural networks (capture nonlinearities and embeddings) - Two-Tower models (separately embed users and items and train jointly)

4. Model Evaluation

- Metrics: AUC, precision@k, recall@k, NDCG
- Use holdout sets or temporal validation (e.g., train on week N, test on N+1)
- Perform A/B testing before full deployment

5. Deployment

- Export the trained model (e.g., as ONNX or Pickle)
- Load it into the Ranking Service
- At inference time, serve feature vectors and apply the model to score candidates

Ranking Service (ML Models)

Pipeline:
- Pre-Ranker:

- Filters 500 → 200 candidates
- Uses fast ML (logistic regression or tiny GBDTs)
- Latency: ~5 ms

- Main Ranker:
  - Scores ~200 candidates
  - Uses heavy ML (XGBoost, DNNs, Two-Tower Models)
  - Latency: ~10–20 ms

Output: Ranked candidate list with scores

Post-Ranking Logic

- Final filtering and reordering:
  - Top-N selection (e.g., 50)
  - Diversity enforcement
  - Business rules (e.g., NSFW demotion, engagement balancing)

Feed Assembly

- Ranked items are passed via Kafka to Feed Assembly Service
- Creative Renderer hydrates content with:
  - Images, video URLs, text
  - User profile metadata
  - CDN asset links

User Delivery

- Final feed sent to mobile or web client
- Client renders using app-native frameworks
- User actions feed back into Kafka for the next cycle

Logging and Feedback

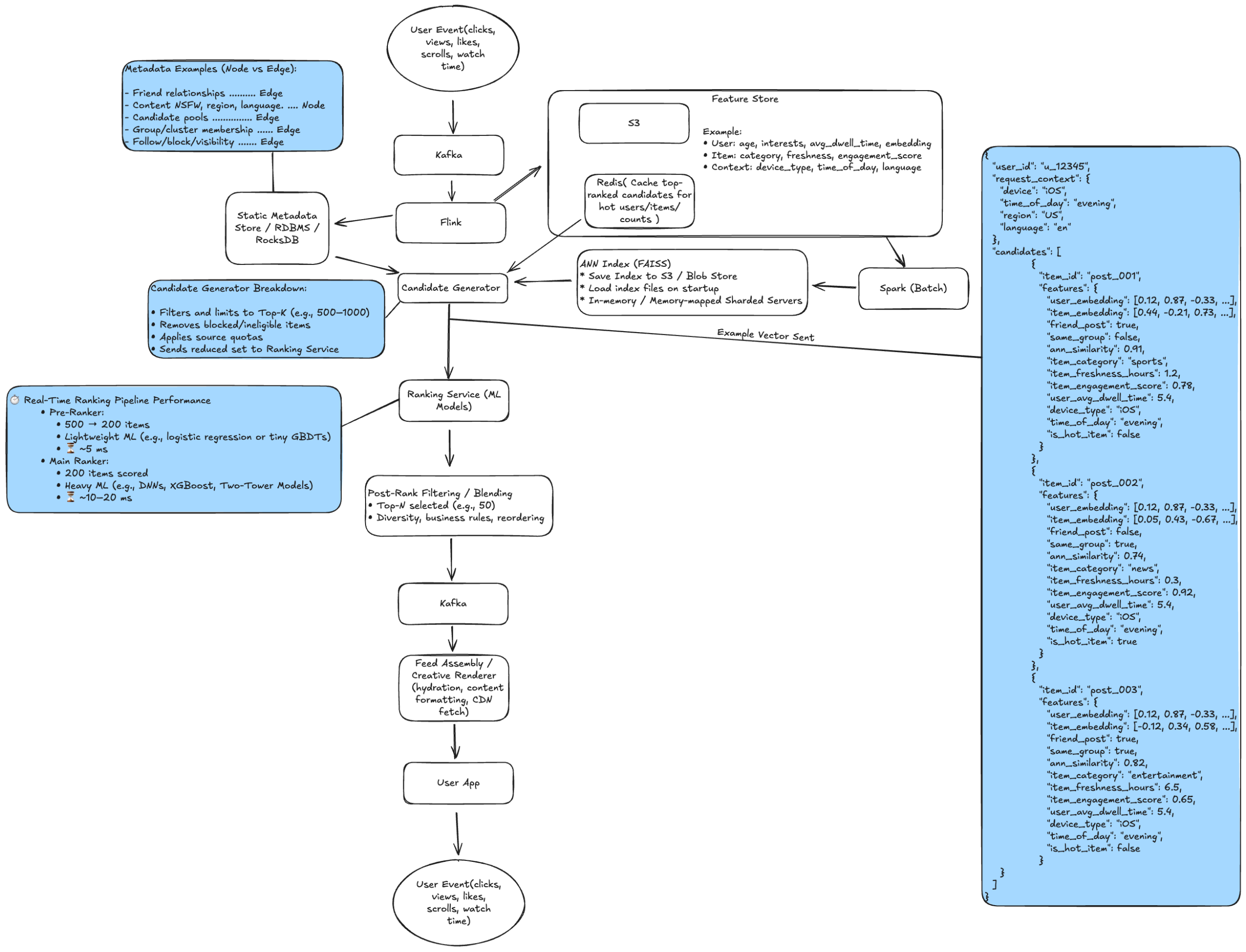
- All impressions and clicks are logged
- Used for:
  - Offline model training
  - Real-time metrics and observability
  - A/B testing and experimentation

Performance Summary

- Candidate Generator: ~50 ms
- Pre-Ranker: ~5 ms
- Main Ranker: ~10–20 ms
- Total ranking latency: ~20–30 ms
- End-to-end (including rendering): ~50–150 ms

Architecture Diagram

You can edit this diagram by uploading the PNG to Excalidraw.


User Event(clicks, views, likes, scrolls, watch time)
Kafka
Flink
Metadata Examples (Node vs Edge):
- Friend relationships .... Edge
- Content NSFW, region, language. .... Node
- Candidate pools .... Edge
- Group/cluster membership ...... Edge
- Follow/block/visibility .... Edge
Static Metadata Store / RDBMS / RocksDB
Feature Store
S3
Example:
• User: age, interests, avg_dwell_time, embedding
• Item: category, freshness, engagement_score
• Context: device_type, time_of_day, language
Redis(Cache top-ranked candidates for hot users/items/ counts)
ANN Index (FAISS)
* Save Index to S3 / Blob Store
* Load index files on startup
* In-memory / Memory-mapped Sharded Servers
Spark (Batch)
Candidate Generator
Candidate Generator Breakdown:
• Filters and limits to Top-K (e.g., 500–1000)
• Removes blocked/ineligible items
• Applies source quotas
• Sends reduced set to Ranking Service
Example Vector Sent
Ranking Service (ML Models)
⏱ Real-Time Ranking Pipeline Performance
• Pre-Ranker:
• 500 → 200 items
• Lightweight ML (e.g., logistic regression or tiny GBDTs)
• ⏳ ~5 ms
• Main Ranker:
• 200 items scored
• Heavy ML (e.g., DNNs, XGBoost, Two-Tower Models)
• ⏳ ~10–20 ms
Post-Rank Filtering / Blending
• Top-N selected (e.g., 50)
• Diversity, business rules, reordering
Kafka
Feed Assembly / Creative Renderer (hydration, content formatting, CDN fetch)
User App
User Event(clicks, views, likes, scrolls, watch time)

```
{
  "user_id": "u_12345",
  "request_context": {
    "device": "iOS",
    "time_of_day": "evening",
    "region": "US",
    "language": "en"
  },
  "candidates": [
            {
              "item_id": "post_001",
              "features": {
                "user_embedding": [0.12, 0.87, -0.33, ...],
                "item_embedding": [0.44, -0.21, 0.73, ...],
                "friend_post": true,
                "same_group": false,
                "ann_similarity": 0.91,
                "item_category": "sports",
                "item_freshness_hours": 1.2,
                "item_engagement_score": 0.78,
                "user_avg_dwell_time": 5.4,
                "device_type": "iOS",
                "time_of_day": "evening",
                "is_hot_item": false
              }
            },
            {
              "item_id": "post_002",
              "features": {
                "user_embedding": [0.12, 0.87, -0.33, ...],
                "item_embedding": [0.05, 0.43, -0.67, ...],
                "friend_post": false,
                "same_group": true,
                "ann_similarity": 0.74,
                "item_category": "news",
                "item_freshness_hours": 0.3,
                "item_engagement_score": 0.92,
                "user_avg_dwell_time": 5.4,
                "device_type": "iOS",
                "time_of_day": "evening",
                "is_hot_item": true
              }
            },
            {
              "item_id": "post_003",
              "features": {
                "user_embedding": [0.12, 0.87, -0.33, ...],
                "item_embedding": [-0.12, 0.34, 0.58, ...],
                "friend_post": true,
                "same_group": true,
                "ann_similarity": 0.82,
                "item_category": "entertainment",
                "item_freshness_hours": 6.5,
                "item_engagement_score": 0.65,
                "user_avg_dwell_time": 5.4,
                "device_type": "iOS",
                "time_of_day": "evening",
                "is_hot_item": false
              }
    }
  ]
}
```

Figure 1: ML Recommendation System Architecture