Production-Scale Recommendation System

Complete System Flow & Architecture

Generated: February 11, 2026

Table of Contents

1. System Overview

This is a **production-grade recommendation system** designed to handle billions of requests daily, similar to systems at Roblox, Meta, or Google. The architecture balances **user experience** (relevant recommendations) with **revenue optimization** (effective monetization).

Key Capabilities:

Component	Description	Scale
Throughput	Handles 10K+ queries per second	Billions of requests/day
Latency	Sub-100ms p99 latency	< 50ms typical
Catalog Size	Millions of items	10M+ items indexed
Users	Millions of active users	100M+ daily active
Model Complexity	Transformer-based CTR prediction	BERT + LightGBM ensemble
Training Data	100TB+ interaction data	Daily retraining

2. Data Pipeline Flow

The data pipeline processes raw user-item interactions from various sources (web logs, mobile apps, streaming events) and transforms them into clean, validated datasets ready for model training.

2.1 Data Sources

Source	Type	Volume	Update Frequency
User Interactions	Event logs (S3/Parquet)	Billions/day	Real-time streaming
User Profiles	Database snapshot	100M records	Daily batch
Item Metadata	Database + CMS	10M items	Hourly batch
Context Data	Real-time API	N/A	Per request

2.2 Data Validation Steps

Step 1: Schema Enforcement
- Validate data types (user_id: string, timestamp: datetime, etc.)
- Ensure required fields are present
- Reject malformed records

Step 2: Quality Checks
- Remove null values in critical fields (user_id, item_id, timestamp)
- Deduplicate identical interactions
- Filter invalid timestamps (future dates, too old)
- Validate event types (view, click, purchase, etc.)

Step 3: Anomaly Detection
- Detect spikes in daily event volume (potential data pipeline issues)
- Identify bot-like behavior (users with >1000 interactions/day)
- Flag statistical outliers

Step 4: Data Quality Metrics
- Track data quality rate: clean_records / total_records
- Alert if quality rate < 95%
- Log validation reports for monitoring

2.3 Train/Test Split Strategy

Time-based split (critical for recommendation systems):

• **Why time-based?**
  - Prevents data leakage (no future information in training)
  - Simulates production scenario (predict future from past)
  - Accounts for temporal patterns and seasonality

• **Split strategy:**
  - Training: All data up to T-14 days
  - Validation: T-14 to T-7 days
  - Test: Last 7 days

- **Why this matters:**
  - Random splits can inflate metrics by 10-20%
  - Time-based split gives realistic performance estimates

3. Feature Engineering Pipeline

Feature engineering transforms raw data into meaningful signals for machine learning models. This is arguably **the most important step** in building effective recommendation systems.

3.1 User Features

Feature Category	Examples	Purpose
Demographics	Age, gender, location, language	Broad personalization
Behavior Stats	Total interactions, avg session time, CTR	Engagement level
Preferences	Favorite categories, brands, price range	Content affinity
Recency	Days since last visit, last purchase	User lifecycle stage
Purchase History	Conversion rate, avg order value, LTV	Revenue optimization
Sequential	Last 50 items interacted with	Temporal patterns

3.2 Item Features

Feature Category	Examples	Purpose
Content	Title, description, category, tags	Content-based filtering
Popularity	Total views, CTR, conversion rate	Trending items
Quality	Average rating, number of reviews	Quality filtering
Temporal	Days since creation, trending score	Freshness boost
Text Embeddings	BERT/sentence-transformers vectors	Semantic similarity
Metadata	Price, brand, availability	Business rules

3.3 Contextual Features

Context features capture the **situation** in which recommendations are requested:

• **Temporal:** Hour of day, day of week, is_weekend, holiday season
• **Device:** Mobile vs desktop, OS, screen size
• **Location:** Country, timezone, language
• **Session:** Pages visited, time on site, search queries
• **Placement:** Homepage, category page, search results

Why cyclical encoding?
For temporal features like hour_of_day, we use sin/cos encoding to handle continuity (23:00 is close to 00:00, not far away):

```
hour_sin = sin(2π × hour / 24) hour_cos = cos(2π × hour / 24)
```

4. Embedding Generation

Embeddings are dense vector representations that capture similarity in a low-dimensional space. They are fundamental to modern recommendation systems.

4.1 Embedding Strategies

Strategy	When to Use	Pros	Cons
Matrix Factorization	Simple baseline, cold start	Fast, interpretable	Limited to user-item
Two-Tower Neural	Production systems	Scalable, fast serving	Requires more data
Sequential (Transformer)	Session-based	Captures temporal	Higher latency
Multi-modal	Rich content (text+image)	Best accuracy	Most expensive

4.2 Two-Tower Architecture (Industry Standard)

Why Two-Tower?
- Separate user and item encoders enable **independent caching**
- Item embeddings computed offline (static, updated daily)
- User embeddings computed online (dynamic, based on recent behavior)
- Enables **fast ANN search** for candidate generation

Architecture:

User Tower:
User Features → Dense(256) → ReLU → BatchNorm → Dense(128) → ReLU → Dense(128) → L2 Normalize

Item Tower:
Item Features → Dense(256) → ReLU → BatchNorm → Dense(128) → ReLU → Dense(128) → L2 Normalize

Similarity:
score = user_embedding · item_embedding (dot product of normalized vectors = cosine similarity)

4.3 ANN Search with FAISS

Challenge: Computing similarity for 10M items in real-time is too slow (10+ seconds)

Solution: Approximate Nearest Neighbor (ANN) search using FAISS

Index Types:
• **Flat:** Exact search, O(n) - Use for <100K items
• **IVF (Inverted File):** Cluster-based, O(k) - Production standard
• **HNSW:** Graph-based, best recall/speed - Premium choice

Production Setup:
• Index type: IVF with 1000 clusters
• Search nprobe: 10 clusters (1% of total)
• Latency: ~20ms for 10M items → 500 candidates
• Recall@500: 95%+ (vs 100% for brute force)

5. Two-Stage Retrieval Architecture

The Critical Design Decision: Why we can't run complex models on millions of items in real-time.

5.1 Stage 1: Candidate Generation (Fast)

Goal: Quickly narrow down 10M items → 500 candidates
Latency Budget: 20-30ms
Method: Embedding similarity + ANN search

Flow:
1. Fetch user embedding from cache (Redis, <5ms)
   → If cache miss: compute from features (<10ms)
2. Normalize user embedding (L2 norm)
3. FAISS ANN search on item index (~20ms)
   → Returns top 500 items by cosine similarity
4. Apply basic filters (in-stock, region-allowed)

Key Insight: We trade some accuracy (ANN vs exact) for massive speed gain (20ms vs 10s)

5.2 Stage 2: Ranking (Precise)

Goal: Accurately score 500 candidates → top 50 items
Latency Budget: 15-30ms
Method: Complex ML model (LightGBM or Neural Network)

Flow:
1. Fetch detailed features for all 500 user-item pairs
   → Parallelize: user features, item features, context (ThreadPool)
   → Time: ~10ms
2. Create feature vectors (100-200 features)
   → User stats, item metadata, interaction features, cross-features
3. Model inference (batch prediction)
   → LightGBM: ~10ms for 500 items
   → Neural network: ~20ms (GPU batching)
4. Sort by predicted score

Model Choice:
• **LightGBM:** Production standard (fast, accurate)
• **DeepFM/DCN:** For complex interactions (higher latency)
• **Hybrid:** LightGBM for top-500, neural for final top-50

6. Model Training & Evaluation

6.1 Training Pipeline

Daily Retraining Schedule:

1. Data Collection (00:00 - 02:00 UTC)
• Aggregate last 7 days of interaction data
• Join with user profiles and item metadata
• Run validation and quality checks

2. Feature Engineering (02:00 - 04:00 UTC)
• Compute user statistics and preferences
• Generate item popularity metrics
• Create sequential features and embeddings

3. Model Training (04:00 - 10:00 UTC)
• **Embedding Models:** 2-4 hours on 4 GPUs
• **Ranking Models:** 1-2 hours on 16 CPUs
• Hyperparameter tuning with Optuna
• Cross-validation for model selection

4. Evaluation (10:00 - 11:00 UTC)
• Offline metrics: AUC, NDCG, Log Loss
• Compare with baseline and previous model
• Generate evaluation report

5. Deployment (11:00 - 12:00 UTC)
• A/B test on 5% traffic
• Monitor online metrics (CTR, latency)
• Gradual rollout if successful

6.2 Evaluation Metrics

Metric	Purpose	Target	Why It Matters
AUC-ROC	Binary classification	>0.75	Overall model quality
Log Loss	Calibration	<0.35	Probability accuracy
NDCG@10	Ranking quality	>0.80	Position matters
MAP@10	Precision	>0.60	Relevant items at top
Coverage	Catalog diversity	>30%	Don't ignore long tail
Novelty	Surprise factor	Balanced	Avoid filter bubbles

Critical Insight: Offline metrics ≠ Online metrics

• **Offline:** AUC = 0.80 (test set)
• **Online:** CTR = 3.5% (real users)

Why the gap?
• Distribution shift (training data is old)
• Position bias (users click top results)

• Selection bias (shown items ≠ all items)
• User behavior changes

Solution: Always A/B test before full deployment!

7. Production Serving

7.1 Serving Architecture

Infrastructure Stack:

• **API Layer:** FastAPI (async, high-performance)
• **Model Serving:** NVIDIA Triton (GPU inference) or TorchServe
• **Feature Store:** Feast + Redis (online) + S3 (offline)
• **Cache:** Redis for user embeddings, popular items
• **ANN Search:** FAISS on GPU
• **Load Balancer:** Nginx or AWS ALB
• **Orchestration:** Kubernetes (100+ pods)

Deployment Strategy:
• Horizontal scaling: 100+ replicas
• Auto-scaling based on QPS and latency
• Blue-green deployment for zero downtime
• Canary releases for new models

7.2 Caching Strategy

Cache Type	Data	TTL	Benefit
User Embeddings	128-dim vectors	1 hour	Skip feature fetch + encoding
Popular Items	Top 1000 items	15 min	Cold start fallback
Item Metadata	Category, price, etc.	1 day	Reduce DB queries
Feature Vectors	Pre-computed features	6 hours	Faster ranking
Model Predictions	User-item scores	30 min	Repeat requests (rare)

7.3 Business Logic Layer

Post-processing rules applied after model scoring:

1. Diversity
• Max 3 items per category in top-10
• Ensures variety for better user experience
• Prevents over-concentration on popular categories

2. Freshness Boost
• Boost recently added items (exponential decay)
• Helps new content get initial exposure
• Formula: score × (1 + 0.1 × e^(-age/30))

3. Deduplication
• Remove items user recently viewed/purchased
• Fetch from Redis (recent_items:user_id)
• TTL: 7 days

4. Business Filters
• Out-of-stock items

- Region restrictions
- Age-appropriate content
- Brand safety rules

8. Monitoring & Observability

8.1 Key Metrics to Monitor

Category	Metrics	Alert Threshold	Action
Latency	p50, p95, p99	p99 > 100ms	Scale up replicas
Throughput	QPS, RPS	Drop > 20%	Check upstream
Model Quality	CTR, CVR, NDCG	Drop > 5%	Investigate drift
Data Drift	PSI, KL divergence	PSI > 0.2	Retrain model
System Health	CPU, Memory, GPU	Usage > 80%	Scale resources
Error Rate	4xx, 5xx errors	Rate > 0.1%	Rollback if needed

8.2 Data Drift Detection

Population Stability Index (PSI): Industry standard for drift detection

How it works:
1. Bin training data distribution into deciles
2. Compare production data to same bins
3. $PSI = \Sigma (prod\% - train\%) \times \ln(prod\% / train\%)$

Interpretation:
• $PSI < 0.1$: No significant drift ✓
• $0.1 < PSI < 0.2$: Moderate drift - monitor closely ■
• $PSI > 0.2$: Significant drift - retrain immediately ■

What to monitor:
• All numerical features (age, price, engagement metrics)
• Categorical distributions (category mix, device types)
• Target variable (CTR, conversion rate)

8.3 A/B Testing Framework

Experimental Rigor:

Before Launch:
• Calculate required sample size (power analysis)
• For 5% MDE at 80% power: typically 50K-100K users per variant
• Define success metrics and guardrails

During Experiment:
• Random assignment to control/treatment
• Monitor guardrail metrics (no degradation)
• Check for novelty effects (day 1 vs day 7)

Analysis:
• Statistical significance test (z-test for proportions)
• Confidence intervals for lift estimation
• Multiple testing correction if running many experiments

Decision Criteria:
• $p < 0.05$ AND relative lift > 2% → Ship ✓
• $p > 0.05$ OR neutral/negative → Don't ship ✗

9. End-to-End Request Flow

Complete flow from API request to response:

Step 1: Request Received (0ms)
→ User makes request via API: GET /recommend?user_id=12345#_items=20
→ Load balancer routes to available service replica

Step 2: User Embedding Fetch (5ms)
→ Check Redis cache: user_emb:12345
→ If cache hit: return 128-dim vector
→ If cache miss: fetch features from Feast → encode with user tower → cache

Step 3: Candidate Generation (20ms)
→ Normalize user embedding (L2 norm)
→ FAISS IVF search on 10M item embeddings
→ Retrieve top 500 candidates by cosine similarity
→ Apply basic filters (in-stock, region-allowed)

Step 4: Feature Fetching (10ms)
→ Parallel fetch with ThreadPoolExecutor:
• User features (demographics, behavior stats)
• Item features (metadata, popularity, quality)
• Context features (time, device, location)
→ Construct 500 feature vectors (one per candidate)

Step 5: Ranking Model Inference (15ms)
→ LightGBM batch prediction on 500 items
→ Output: predicted CTR/engagement score per item
→ Sort by score descending

Step 6: Business Logic (5ms)
→ Apply diversity constraints (max 3 per category)
→ Apply freshness boost to new items
→ Deduplicate against recent views
→ Re-rank after adjustments

Step 7: Response Construction (2ms)
→ Take top 20 items
→ Fetch display metadata (title, image URL, price)
→ Format JSON response

Step 8: Response Sent (57ms total)
→ Return ranked items with scores and metadata
→ Log request for monitoring and offline learning
→ Well under 100ms p99 SLA ✓

Stage	Latency	Critical Path?	Optimization
User Embedding	5ms	Yes	Redis caching (99% hit rate)
Candidate Gen	20ms	Yes	FAISS GPU, IVF indexing
Feature Fetch	10ms	Yes	Parallel ThreadPool, Feast
Ranking	15ms	Yes	LightGBM batching, CPU

Business Logic	5ms	No	In-memory processing
Response Format	2ms	No	JSON serialization
Total	**57ms**	**-**	**p99 < 100ms SLA**

10. Key Design Decisions

10.1 Why Two-Stage Architecture?

Problem: Can't run complex models on millions of items in real-time

Naive approach: Score all 10M items with ranking model
→ Latency: 10M × 0.01ms = 100 seconds ■

Two-stage approach:
→ Stage 1 (Fast): 10M → 500 candidates in 20ms using embeddings
→ Stage 2 (Precise): 500 → 50 final items in 15ms using complex model
→ Total: 35ms ✓

Trade-off:
• Some accuracy lost in Stage 1 (ANN vs exact search)
• Massive speed gain enables real-time serving
• Accuracy recovered in Stage 2 with rich features

Industry adoption: YouTube, Google, Meta, Pinterest, TikTok all use this pattern

10.2 LightGBM vs Deep Learning for Ranking?

LightGBM Advantages:
• Fast inference: 10ms for 500 items
• Handles mixed data types naturally (numeric + categorical)
• Built-in feature interactions
• Interpretable (feature importance)
• Less training data needed

Deep Learning Advantages:
• Better for unstructured data (text, images)
• Learns complex non-linear interactions
• Transfer learning (pre-trained models)

Our Choice: Hybrid Approach
• Use transformers for *embeddings* (Stage 1)
• Use LightGBM for *ranking* (Stage 2)
• Best of both worlds: semantic understanding + fast serving

10.3 Feature Store: Why Essential?

Problem: Training/Serving Skew

Without feature store:
• Training: Compute features with Spark SQL on S3 data
• Serving: Compute features with Python code on PostgreSQL
→ Different logic → Different results → Model performs poorly in production ■

With feature store (Feast):
• **Single source of truth** for feature definitions
• Same features in training (offline) and serving (online)
• Point-in-time correctness prevents data leakage
• Feature versioning for reproducibility

Architecture:
• Offline: S3/Parquet for training (batch)
• Online: Redis for serving (low-latency)
• Sync: Daily materialization job

Impact: Feature stores typically improve model accuracy by 5-10% by eliminating skew

10.4 Why Daily Retraining?

User behavior and item catalog change constantly:

• New items added daily → need fresh embeddings
• User preferences evolve → need updated user models
• Seasonal trends (holidays, events) → need adaptive weights
• Competitors launch campaigns → market dynamics shift

Retraining Frequency Trade-offs:

Weekly: Cheaper, but stale (up to 7 days old data)
Daily: Good balance for most systems ✓
Hourly: Fresh but expensive (high compute cost)
Real-time: Online learning (complex, can be unstable)

Our approach: Daily batch + hourly embedding updates for new items

Appendix: Quick Reference

System Components Summary

Component	Technology	Purpose
Data Pipeline	PySpark	ETL and data validation
Feature Store	Feast + Redis + S3	Online/offline features
Embedding Models	PyTorch Two-Tower	User/item vectors
ANN Search	FAISS (IVF)	Fast candidate retrieval
Ranking Model	LightGBM	Precise scoring
API Server	FastAPI	REST endpoints
Caching	Redis	User embeddings, metadata
Orchestration	Kubernetes	Scaling and deployment
Monitoring	Prometheus + Grafana	Metrics and alerts
Experiment	Custom A/B framework	Statistical testing
ML Tracking	MLflow	Model versioning

Performance Targets

Metric	Target	Current
Latency (p99)	< 100ms	~50ms
Throughput	> 10K QPS	12K QPS
CTR	> 3%	4.2%
Model AUC	> 0.75	0.78
NDCG@10	> 0.80	0.82
Uptime	> 99.9%	99.95%

This document provides a comprehensive overview of a production-scale recommendation system designed for platforms serving billions of users. The architecture emphasizes scalability, low latency, and maintainability while balancing user experience with business objectives.