ChatGPT

Instagram Reels-Style Recommendation System Design

High-Level Architecture Overview

The system is typically **two-tiered**: an **offline (batch)** side and an **online (real-time)** side. In offline mode, raw data is ingested into a data lake (e.g. Amazon S3) and processed (using Spark/EMR or AWS Glue) to build feature tables and train models. In online mode, a streaming pipeline collects user events (clicks, views, likes) and updates an **online feature store** for fast lookup. An inference service then uses these features to generate recommendations. Modern designs use microservices and event-driven components (API Gateway, Kinesis/Kafka, Lambda) for modularity and scalability. A multi-stage pipeline is common: for example, Amazon's reference design uses a **collaborative-filtering model** (e.g. matrix factorization) to propose candidates, and a separate **ranking model** to re-rank them by user context [1] [2]. The overall data flow is: user event → streaming processor → (cache/feature store) → model inference → response. This separation also supports offline analytics and A/B tests versus the live system.

Data Pipeline Design

Figure: Example data pipeline for streaming user events into feature stores and batch training.

- **Data Sources:** Collect clickstream and engagement logs (video plays, skips, likes, comments), content metadata (video duration, creator, text tags), social signals (follows, shared links), and external trend signals. For example, Instagram-style data includes which Reels a user watches (and for how long), the video's audio track ID, etc.
- **Ingestion:** Use a scalable streaming service (e.g. Amazon Kinesis Data Streams or Kafka) to ingest real-time events from clients. Events are sent to a **stream processing** layer (e.g. Kinesis Data Analytics or Apache Flink) for aggregation. Batch data (e.g. historical logs) are periodically loaded into data lakes (S3, Redshift). For instance, AWS uses Kinesis → Kinesis Data Analytics (windowed SQL) → AWS Lambda to push aggregated clickstream features into a SageMaker Feature Store [3].
- **Preprocessing:** In each pipeline, filter out bot/spam, dedupe events, sessionize user activity, and join with user/video metadata. For example, join click events with user profiles (demographics) and video attributes (category, trending score). Aggregate features (counts, recency) and encode categorical variables (one-hot or learned embeddings). Maintain user histories (sequence of watched video IDs) and compute session-level features (e.g. watch counts in last hour).
- **Feature Engineering:** Construct features for both modeling stages. Candidate models typically use *dense embeddings*: map users and videos into a common vector space (via IDs and content features) [4] [5]. For example, a two-tower network could embed user ID + recent history in one tower, and video ID + CNN-derived visual features or text embeddings in another [5]. Also generate static features (user age, location, video language, length) and dynamic features (current time, device, session index). Content features may include audio/text embeddings (e.g. BERT on subtitles, ResNet on thumbnails).
- **Feature Store:** Store feature vectors in a feature store with separate **offline** and **online** stores [6] [7]. The offline store (append-only, e.g. S3-backed) is used for model training/batch scoring, while the online store (low-latency NoSQL or cache) serves real-time inference. AWS SageMaker Feature Store or a Redis/ElastiCache store are typical: these support sub-millisecond lookups of user/video features [7] [8]. For

example, Amazon Music's feature store uses ElastiCache to provide millions of queries/second at single-digit millisecond latency [8].

ML Modeling (Candidate Generation & Ranking)

- **Candidate Generation:** Use a **retrieval model** to narrow billions of videos to a few hundred candidates per user. Common approaches include collaborative filtering (matrix factorization or two-tower networks) and content-based filtering. The two-tower model learns a user embedding from user features and an item (video) embedding from video features [9] [10], and retrieves nearest-neighbor videos by dot-product or cosine similarity. Inference is made efficient by precomputing all video embeddings offline and using an approximate nearest-neighbor index (e.g. Faiss, ScaNN) to find top-K videos for a given user vector [11] [12]. In practice, multiple strategies may run in parallel: popularity-based or trending lists, category-specific recommenders, and ANN from embedding retrieval. As one source notes, "the recommender has a few hundred candidates retrieved from candidate generation (e.g. matrix factorization or neural models) and applies a large-capacity model to rank and sort" [2].
- **Ranking Model:** A *deep neural network* re-ranks candidate videos by modeling fine-grained preferences. Modern rankers often use **multi-task learning** (MTL) to predict multiple user signals (e.g. click/no-click, watch completion fraction, likes, shares) simultaneously. For example, YouTube's research describes a ranking architecture with a Multi-gate Mixture-of-Experts (MMoE) that branches into heads for clicks, watch time, likes, and dismissals [13] [14]. A popular variant is a combined Wide & Deep model extended with an MMoE or multi-head tower. The **loss function** is typically a weighted sum of per-task losses (e.g. binary cross-entropy for clicks, regression loss for watch time). Negative sampling is used for training (pairs of (user,non-clicked video) vs (user,clicked video)). In practice, pointwise losses (logistic regression on each instance) are often chosen for scalability; one study notes that a pointwise logistic loss "runs 10–60× faster than pairwise or listwise" and performs comparably [15]. The model may include a "shallow tower" component to correct selection bias (e.g. position bias), as done in YouTube's system [16]. Overall, the ranker outputs a score for each candidate and returns a personalized top-N sorted list.

Real-Time Serving Architecture

- **Inference Service:** User requests (page loads, scroll events) hit an inference API (e.g. via Amazon API Gateway → AWS Lambda or microservice). The service retrieves the user's latest features (from the online Feature Store or cache like Redis [8]) and, if needed, computes the user embedding. It then queries the candidate generation service (ANN index) to get ~100–1000 video IDs. Next, the ranking model scores these candidates on the fly (typically a lightweight forward pass) and returns the top recommendations. All of this must happen at low latency (e.g. <100ms p95). **Caching** is important: popular video features or frequently queried user embeddings can be cached in-memory to avoid recomputation. ElastiCache/Redis is a common choice for sub-millisecond feature lookups and caching heavy results [8].
- **Infrastructure:** In practice, the model can be deployed on containerized endpoints (Amazon EKS/ECS) behind a load balancer. AWS recommends hosting ML inference on ECS with a Network Load Balancer to achieve millisecond latency at high throughput [17]. GPU or CPU-based instances may serve the model, depending on compute needs. SageMaker multi-model endpoints or custom serving (e.g. Triton Inference Server) can also be used. The system should auto-scale based on traffic. High availability is achieved via multi-AZ deployments.
- **A/B Testing and Experimentation:** The serving layer typically includes an experimentation framework. Traffic can be routed to different models or model versions (bucketed by user or time). For example, one can use weighted rollout (e.g. 5% of users see a new ranker) and

compare metrics. Amazon's recommendation solutions often use A/B testing to validate lifts in CTR or engagement. Real-time metrics (impressions, clicks, watch-time, shares) are logged (e.g. to Kinesis Data Firehose → Redshift or CloudWatch) for analysis. Care is taken to capture randomization seeds and ensure statistically sound experiments.

Feedback Loop and Model Retraining

- **Implicit Signals:** User interactions serve as labels for retraining. For example, if a user watches a Reel > 50% or likes it, that interaction becomes a positive signal; skipping or short watches are negatives. These signals (views, likes, shares, dwell time) are streamed back to the data store. A **labeling service** may assign binary or graded targets (e.g. multi-level watch-duration buckets).
- **Retraining Frequency:** Models are retrained regularly to capture changing trends. Short-video feeds often retrain daily or even multiple times per day. A pipeline takes recent interaction logs (from the offline store), merges with historical data, and re-trains the embedding and ranking models. Online feature-store (or event logs) are used to prepare fresh training examples. Systems like Amazon Personalize automate retraining on schedule; custom pipelines might use AWS SageMaker Pipelines or Step Functions to orchestrate nightly retraining jobs.
- **Drift Detection:** The system must detect when the model's performance degrades. This can use shadow testing (run the current model offline on recent data) or monitors on online metrics (e.g. sudden CTR drop). Data drift (changes in feature distributions, e.g. new trending topics) can be monitored with tools like Evidently or CloudWatch alarms. When drift is detected, a new training run is triggered.
- **Exploration vs. Exploitation:** To avoid feedback loops and cold-start bias, some exploration is necessary. The system might occasionally insert some random or novel videos into the feed to collect unbiased signals (multi-armed bandit techniques). These interactions enrich the data for retraining. Logging should capture not only positive actions but also exposures (impressions) to enable unbiased evaluation.

Personalization & Context-Aware Modeling

- **User Embeddings:** We learn a **user embedding** vector from their history. This uses user-specific inputs (ID, demographics, historical video IDs, watch frequencies) through a neural network to produce a dense representation of preferences [9]. Sequence models (RNNs or Transformers) may process the recent watch sequence to capture short-term interests.
- **Content Embeddings:** Similarly, each video has an **item embedding** from its features (ID, category, tags, visual/audio embeddings) [10]. By placing users and items in the same latent space, we personalize via similarity. At serving time, the user's embedding is matched against precomputed video embeddings with a fast nearest-neighbor lookup [11].
- **Context Features:** Context (time of day, device type, location, session depth) is also fed into the model. For instance, separate embedding tables can represent time bins or device categories, or the context can modulate the ranking scores. Contextual bandit techniques can adapt recommendations based on real-time context.
- **Multi-Task & Personalization:** In a multi-task ranker, personalization arises naturally: the user and content features feed into shared layers. If using MMoE, each user and item vector is multiplied with expert networks, allowing specialization (e.g. certain experts focus on comedy videos, others on fast cuts). This yields a fine-grained personalization where, for example, users who binge-watch dance videos get different expert weights than those who prefer DIY crafts.

Evaluation Metrics & Validation

- **Offline Metrics:** Before deployment, models are evaluated on hold-out logs. Common ranking metrics include **Precision@K, Recall@K, MAP, and NDCG** (Normalized Discounted Cumulative Gain) at various cutoffs [18] . These measure how well the model ranks true positives near the top. For CTR prediction tasks, **ROC-AUC** is used to assess classification accuracy (research shows even a 1% AUC gain can yield significant CTR lifts online [19]). Offline evaluation must be done carefully: time-based splits (training on past, testing on future) avoid leakage. Counterfactual or IPS correction techniques can be applied to mitigate exposure bias in evaluation logs. Also measure beyond accuracy: **novelty, diversity, and serendipity** of recommendations are important secondary metrics [18] .
- **Online Metrics:** Ultimately we judge by user engagement. A/B tests compare new models versus control on business metrics: **Click-Through Rate (CTR) or Watch-Through Rate (WTR)** of Reels, average watch duration per session, number of shares/comments per user, and user retention/DAU. For example, one may measure "CTR uplift" in the experimental bucket. Secondary metrics include dwell time in the app, number of videos viewed per session, and any monetization signals. The shaped.ai blog notes that offline NDCG or AUC can be promising, but final judgement comes from live CTR/lift [20] . Statistical significance and ramp-up (gradual rollout) are used to validate findings. All online experiments are tracked (e.g. via Redshift dashboards) to ensure improvements hold at scale.

Explainability and Fairness Considerations

- **Transparency:** Deep neural rankers are largely black boxes, but we can provide some transparency. For example, log features contributing to a recommendation ("Because you watched X, we show you Y"). Attention mechanisms or feature attribution (e.g. SHAP) can identify which features most influenced a score. Some systems include an "explainable" module (e.g. highlighting words or tags that match user interests) to improve trust. Encouragingly, explainable recommendation is an active research area [21] .
- **Fairness and Bias Mitigation:** Recommenders can inadvertently amplify biases (e.g. popularity bias or demographic bias). Studies identify issues like over-recommending popular creators or under-recommending niche content [22] . Mitigation strategies include: (a) *Pre-processing* – balancing training data so minority content/groups aren't underrepresented [23] ; (b) *In-processing* – adding regularization or constraints to the loss (e.g. an exposure regularizer that penalizes popularity skew) [23] ; and (c) *Post-processing* – re-ranking results to enforce diversity or exposure parity (for instance, ensuring videos from all creators get some exposure) [23] [24] . We may define group fairness metrics (e.g. exposure fairness across video categories) and monitor them. For user fairness, ensure different demographic groups receive equally relevant content. In Amazon's context, systems can use multi-stakeholder fairness: treat both users and content creators as stakeholders and ensure equitable treatment.
- **User Control:** To empower users, the UI may allow feedback ("not interested in this topic") that is fed back into the model. For instance, if a user rejects or hides a recommendation, that signal is treated as negative feedback in retraining. Providing controls (e.g. allow user to select interests or filter out content) further enhances fairness and user satisfaction.
- **Example:** In practice, a sophisticated ranker might include a small network to correct position bias; indeed, YouTube's model adds a "shallow tower" specifically to model and debias for how items were presented [16] . Such techniques help ensure the model learns true preferences, not just historical click biases.

Scalability Considerations

- **Data Scale:** The system must handle *millions of users* and *billions of videos*. All components must scale horizontally. Raw logs may reach terabytes per day; using big-data storage (S3, HDFS) and distributed processing (Spark on EMR or Databricks) is essential.
- **Model Scale:** Training large models on this data requires distributed training (e.g. SageMaker on multiple GPU instances or using Horovod). Candidate retrieval must work on billions of embeddings – hence FAISS or similar approximate NN libraries that compress vectors are used [12]. The two-tower approach is elegant here: item vectors (billions) are precomputed offline and stored, and only the user tower needs an online forward pass. Approximate search then yields candidates in milliseconds [11].
- **Serving Scale:** The online service must support high QPS (requests/sec). Caches (Redis, DynamoDB Accelerator) help by storing hot features and results. Auto-scaling groups (EKS/ECS tasks) replicate the inference service across servers. For example, using AWS ElastiCache, Amazon Music serves terabytes of feature data in-memory with sub-ms latency [8]. Traffic routing (multi-region deployment) and disaster recovery (backup endpoints) ensure availability.
- **Throttling & Sharding:** To prevent overload, rate-limiting and user-sharding techniques are applied. Popular video embeddings can be sharded by ID, and user requests can be partitioned by hashed user ID. Message queues (Kinesis shards or Kafka partitions) can scale out ingestion. In Amazon's architecture guidance, using load balancers and auto-scaling is key to support millions of requests per second [17].

Tools and Frameworks

- **AWS Services:** A system at Amazon would leverage managed services:
- *Data Ingestion:* Amazon Kinesis Data Streams (or MSK/Kafka) for event collection, AWS Lambda for lightweight processing, and Amazon API Gateway for secure APIs [25] [26].
- *Storage:* Amazon S3 as the data lake, DynamoDB or Amazon ElastiCache (Redis) for low-latency feature storage [8]. AWS Glue or EMR for ETL jobs, and Redshift/EMR/Snowflake for analytics.
- *Feature Store:* Amazon SageMaker Feature Store (online and offline) to manage features [7] [27]. Alternatively, an in-memory Redis cache (Amazon ElastiCache) can serve as an ultra-low-latency store [8].
- *Training:* Amazon SageMaker (notebooks, training jobs) supports XGBoost, TensorFlow, PyTorch training and hyperparameter tuning [28]. For example, one AWS sample instantiates a SageMaker XGBoost estimator and fits it on S3 data [29]. AWS Batch or EKS could run custom TensorFlow/PyTorch jobs.
- *Serving:* SageMaker Endpoints, Amazon ECS/EKS with containers (TensorFlow Serving, TorchServe), or AWS Lambda for light models. Load balancing via Amazon API Gateway or NLB ensures high throughput [17].
- *Streaming Analytics:* Amazon Kinesis Data Analytics or AWS Lambda for streaming ETL; Amazon Personalize for turn-key recommendations (though custom models give more control) [25] [26].
- *Monitoring & A/B:* Amazon CloudWatch for logs/metrics, Amazon QuickSight or CloudWatch dashboards for KPI tracking, and Amazon EMR or SageMaker Model Monitor for data drift. Experiments can be managed via SageMaker Experiments or custom A/B tooling.
- **Open-Source Tools:** The models themselves use ML libraries (TensorFlow, PyTorch, XGBoost, Scikit-learn). For candidate retrieval, **FAISS** (Facebook) is widely used for billion-scale ANN [12]. Apache Spark/Flink for batch/stream processing, and Redis or Memcached for caching. Workflow tools like Apache Airflow or Kubeflow Pipelines may orchestrate complex pipelines. Feature store alternatives like **Feast** (open-source) can also be adopted. For indexing, Google's **ScaNN** or Nvidia's **FAISS** GPU versions might be used.

- **Summary:** In practice, Amazon teams combine managed AWS services (S3, Kinesis, SageMaker, ElastiCache) with open-source frameworks (TensorFlow/PyTorch, Spark, FAISS) to build scalable, maintainable recommendation platforms [25] [8].

Sources: This design synthesizes best practices and published architectures for large-scale video recommendation (e.g. YouTube's two-stage multi-task model [13] [14]) along with AWS reference architectures (real-time pipelines using Kinesis, SageMaker, ElastiCache [26] [8]). All details above are drawn from engineering literature and AWS documentation.

---

[1] [3] [6] [7] [27] [28] [29] Accelerate and improve recommender system training and predictions using Amazon SageMaker Feature Store | Artificial Intelligence
https://aws.amazon.com/blogs/machine-learning/accelerate-and-improve-recommender-system-training-and-predictions-using-amazon-sagemaker-feature-store/

[2] [13] [14] [15] [16] Recommending What Video to Watch Next: A Multitask Ranking System
https://daiwk.github.io/assets/youtube-multitask.pdf

[4] Candidate generation overview | Machine Learning | Google for Developers
https://developers.google.com/machine-learning/recommendation/overview/candidate-generation

[5] [9] [10] [11] The Two-Tower Model for Recommendation Systems: A Deep Dive | Shaped Blog
https://www.shaped.ai/blog/the-two-tower-model-for-recommendation-systems-a-deep-dive

[8] Build an ultra-low latency online feature store for real-time inferencing using Amazon ElastiCache for Redis | AWS Database Blog
https://aws.amazon.com/blogs/database/build-an-ultra-low-latency-online-feature-store-for-real-time-inferencing-using-amazon-elasticache-for-redis/

[12] Faiss: A library for efficient similarity search - Engineering at Meta
https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

[17] Guidance for Low-Latency High-Throughput Model Inference Using Amazon ECS
https://aws.amazon.com/solutions/guidance/low-latency-high-throughput-model-inference-using-amazon-ecs/

[18] 10 metrics to evaluate recommender and ranking systems
https://www.evidentlyai.com/ranking-metrics/evaluating-recommender-systems

[19] Offline evaluation metrics: AUC and Average Impression Log Loss on the... | Download Scientific Diagram
https://www.researchgate.net/figure/Offline-evaluation-metrics-AUC-and-Average-Impression-Log-Loss-on-the-hold-out-data-one_fig2_320891287

[20] Recommender Model Evaluation: Offline vs. Online | Shaped Blog
https://www.shaped.ai/blog/evaluating-recommender-models-offline-vs-online-evaluation

[21] [22] [23] [24] Explainable Fairness in Recommendation
https://arxiv.org/pdf/2204.11159

[25] Implement real-time personalized recommendations using Amazon Personalize | Artificial Intelligence
https://aws.amazon.com/blogs/machine-learning/implement-real-time-personalized-recommendations-using-amazon-personalize/

[26] Architectural Patterns for real-time analytics using Amazon Kinesis Data Streams, Part 2: AI Applications | AWS Big Data Blog
https://aws.amazon.com/blogs/big-data/architectural-patterns-for-real-time-analytics-using-amazon-kinesis-data-streams-part-2-ai-applications/