System Design Report: Personalized Video Feeds

Haythem Ben Abdallah

February 1, 2026

Abstract

This document presents the system design for a server-side, rule-based personalization layer for an existing feed platform. The Feed API fetches cached tenant candidate lists and user signals, applies deterministic weighted ranking with diversity optimization, and returns top-N results within strict latency budgets. Content managers retain editorial control via CMS-driven boosts and filters. Inputs are cached aggressively (tenant configs, candidate lists), signals are ingested asynchronously ($\leq$5 min lag), and a robust kill-switch/fallback mechanism serves default trending feeds if failures occur. The design prioritizes latency, explainability, and safe rollouts while maintaining clear, reversible paths to add ML capabilities, real-time signals, and precomputation for higher scale.

Contents

1 Problem Statement & Goals

1.1 Problem Statement

Currently, our platform's video feeds are static and "broadcast-style," delivering the same content order to all users within a given context. This "one-size-fits-all" approach fails to leverage available user interaction data (watch history, engagement) and demographic context, resulting in suboptimal user engagement, lower retention, and a lack of relevance for diverse user bases across our 120+ tenants. We need to transition from a static delivery model to a personalized experience to increase value for end-users and tenants.

1.2 Goals

The primary goal is to build a scalable, multi-tenant backend system that delivers personalized video feeds based on user signals and tenant-specific rules.

1.2.1 What We Are Building

- **Weighted Rule-Based Ranking Engine:** A backend service that re-ranks candidate video lists based on user watch history, explicit engagement (likes, shares), and tenant-configured weighting rules (e.g., editorial boosts, recency).
- **User Signal Pipeline:** A mechanism to ingest and store user events (hashed IDs) with up to 5 minutes of acceptable lag, respecting strict privacy and retention policies (90 days, no PII).
- **Feed API:** A high-performance endpoint (`GET /v1/feed`) serving ranked content with a p95 latency of <250ms at 3k RPS.
- **Tenant Configuration:** Extensions to the existing CMS to allow tenants to override global ranking weights and apply filters.
- **Safety Mechanisms:** A robust feature-flagging system with an immediate "kill switch" to revert to the legacy non-personalized feed in case of failure or performance degradation.

1.2.2 What We Are Not Building (Non-Goals)

- **Cold-Start Machine Learning:** We are explicitly not building complex ML models (collaborative filtering, deep learning recommendation models) for this MVP. Stick to deterministic, rule-based heuristics.
- **Real-time Session Personalization:** While we value freshness, sub-second reaction to user actions within the same session is not a requirement (5-minute signal lag is acceptable).
- **Client-side Ranking:** All ranking logic will reside server-side to ensure consistency and protect proprietary ranking logic.
- **PII Management:** We will not build new PII storage; the system will strictly operate on anonymized/hashed user identifiers provided by the SDK.

1.3 Success Criteria

1.3.1 Performance

- **API Latency:** p95 < 250ms, p99 < 600ms for a standard 20-item feed.
- **Throughput:** Capable of handling 3k RPS peak load.

1.3.2 Data Freshness

- New uploaded content is eligible for feeds within 60 seconds.
- User signals influence the feed within 5 minutes of the event.

1.3.3 Resiliency

- 100% success rate in falling back to the non-personalized feed if the ranking engine times out or fails (Kill Switch verification).

1.3.4 Scalability

- System successfully handles unique ranking configurations for all 120 tenants without crosstalk or performance degradation.

2 High-Level Architecture

2.1 Architecture Diagram

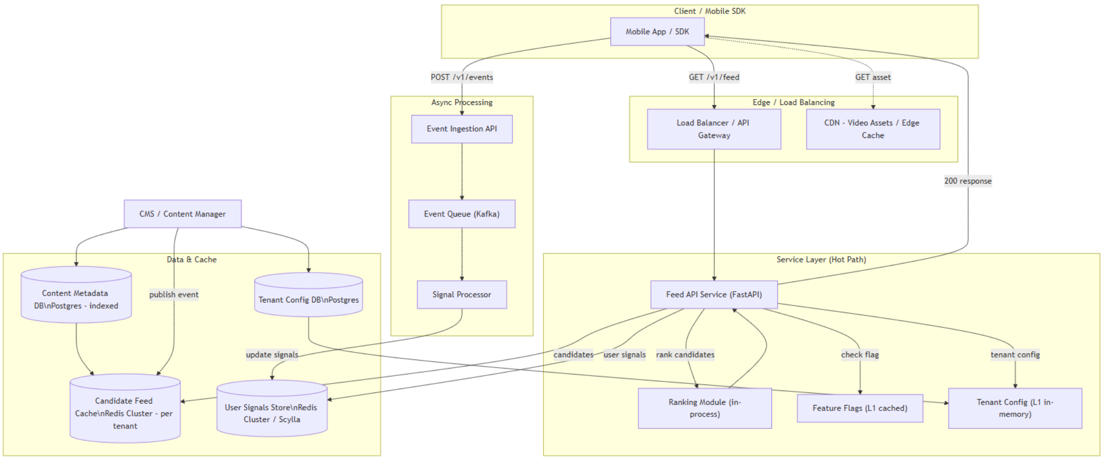

Figure 1: High-Level Architecture

2.2 Core Flows

2.2.1 Request-Time Path (Hot Path)

This path must execute within the 250ms p95 budget.

1. **Request:** SDK calls `GET /v1/feed`.
2. **Safety Check:** Feed API checks Feature Flags. If the "Personalization" flag is OFF (or Kill Switch ON), it immediately fetches and returns a generic "Trending" list from Candidate Feed Cache. This allows 0-latency fallback.
3. **Data Fetching:**
   - **User Signals:** Parallel fetch of the user's recent interactions (last 50 watched IDs, category affinities) from the User Signals Store.
   - **Tenant Rules:** Fetch ranking weights (e.g., "boost sport by 1.5x") from local in-memory Tenant Config Cache.
   - **Candidates:** Fetch the active video pool (metadata) for this tenant from Candidate Feed Cache.
4. **Ranking:** The Ranking Module (in-process) scores the candidates:

$$\text{Final Score} = \text{Base Score (Recency/Popularity)} \times \text{Tenant Weights} \times \text{User Affinity} \tag{1}$$

   Filters out previously watched videos (if rule active).
5. **Response:** Top N videos are returned.

2.2.2 Async Ingestion Path (Offline Path)

1. **Event:** User finishes a video on the SDK. Use fire-and-forget mechanism.
2. **Ingest:** SDK sends batched events (e.g., every 30s or on app background) to Event Ingestion API.
3. **Queue:** API validates format and pushes to Event Queue (Kafka/SQS) to decouple write load.
4. **Process:** Signal Processor consumers pull events, map them to our internal hashing schema, and update the User Signals Store.
5. **Retention:** Old signals automatically expire via TTL (90 days).

2.2.3 CMS Publish Flow

1. **Publish:** Content Manager uploads a video or changes a ranking rule via CMS.
2. **Persist:** Metadata saved to Content Metadata DB (Postgres).
3. **Cache Invalidation:** The CMS publishes an event to clear/refresh the Candidate Feed Cache for that tenant, ensuring the "60s freshness" goal is met.

2.3 Key Architectural Decisions & Trade-offs

2.3.1 Where Does Ranking Happen?

Decision: Server-side, In-Process within Feed API.
Reasoning:

- **Latency:** Calling a separate "Ranking Microservice" adds network hops (serialization/deserialization + wire time). For a p95 of 250ms, we want to minimize internal IO.
- **Data Locality:** By caching candidates and signals heavily, the ranking logic (simple multiplication/sorting) is CPU-bound and fast enough to run on the API nodes.

2.3.2 Caching Strategy & TTLs

Cache Layer	Purpose	Technology	TTL
Tenant Config L1	Usage weights, boost rules	In-Memory (Guava/Caffeine)	1-5 mi
Fallback Feed L1	Top 20 "Trending" items (Safety Cache)	In-Memory (Guava/Caffeine)	1 min
Candidate Feed L2	List of all active video IDs/Metadata per tenant	Redis (Cluster)	5 min
User Signals	Watch history / Preferences	Redis	Persist

Table 1: Caching Strategy

Note: To meet the "60s Freshness" goal, the CMS actively invalidates the Candidate Feed cache immediately upon video publication. The 5-minute TTL is a safety net (fallback) in case the invalidation event is lost.

2.3.3 Why NOT Cache the Final Personalized Feed?

The final personalized feed response is not cached because it is unique to the user and time.

- **The "Spam" Defense:** We prevent abuse via Rate Limiting (HTTP 429) at the API Gateway level (e.g., max 2 refreshes per second), not caching.
- **The UX Argument:** In vertical feeds (TikTok-style), "Pull-to-Refresh" is an explicit user intent asking for fresh content. If we cache for 30s, the user pulls, sees the exact same stale list, gets frustrated, and pulls again.

2.3.4 Hot Path vs. Offline Path

- **Hot Path:** Feed API, Feature Flags, User Signals Read, Candidate Cache. Crucial for UX.
- **Offline Path:** Ingest API, Queue, Workers, Content DB Writes. Crucial for Data Integrity, but can lag.

3 Data Model

3.1 Content Items Table

```sql
CREATE TABLE content_items (
  content_id uuid PRIMARY KEY,
  tenant_id text NOT NULL,
  title text,
  tags text[],
  creator_id text,
  publish_ts timestamptz,
  editorial_boost numeric,
  maturity text,
  metadata jsonb,
  created_at timestamptz DEFAULT now()
);

CREATE INDEX ON content_items (tenant_id, publish_ts DESC);
CREATE INDEX ON content_items USING GIN (tags);
```

Listing 1: content_items table schema

- **Retention:** Permanent unless business TTL required; candidate lists generated from recent `publish_ts`.
- **Purpose:** Authoritative source for eligibility and editorial metadata.

3.2 Candidate Cache (Redis — Per-Tenant)

- **Key:** `candidates:{tenant_id}:{locale?}`
- **Value:** List of candidate objects (id, tags, publish_ts, editorial_boost, global_ctr)
- **TTL:** 30s (aggressive freshness). CMS pushes invalidation events on publish.

3.3 User Signals (Redis Cluster / Scylla for Scale)

Key pattern: `signals:{tenant_id}:{user_hash}`
Value (compact):

```json
{
  "watched_bloom": "<base64_bloom>",
  "recent_watched": ["vid_123","vid_456"],
  "affinities": {"sports":0.8,"music":0.2},
  "engagement_score": 0.72,
  "last_active_ts": 1670000000
}
```

Listing 2: User signals data structure

- **Retention:** TTL / rolling retention 90 days (expire keys older than 90d).
- **Size target:** Compact (<1KB) to keep Redis memory efficient.

3.4 Tenant Configuration (Postgres + L1 Cache)

```sql
CREATE TABLE tenant_configs (
  tenant_id text PRIMARY KEY,
  ranking_weights jsonb,
  editorial_boosts jsonb,
  filters jsonb,
  rollout_pct int DEFAULT 0,
  personalization_enabled boolean DEFAULT false,
  updated_at timestamptz DEFAULT now()
);
```

Listing 3: tenant_configs table schema

- **Cache:** Every Feed API node keeps in-memory L1 copy; refresh on change via pub/sub or poll 30s.

4 API Contract (SDK → Backend)

4.1 Endpoint: GET /v1/feed

Purpose: Return ordered feed for user

4.1.1 Request Example

```
GET /v1/feed?limit=20
Host: api.example.com
Authorization: Bearer <sdk-token>
X-Tenant-Id: tenant_abc
X-User-Hash: sha256:a1b2c3...
If-None-Match: "W/\"feed-v1-tenant_abc-user_a1b2-v42\""
```

Listing 4: GET /v1/feed request

4.1.2 Query Parameters

- `limit` (int, default 20)
- `cursor` (opaque string) — optional; for pagination

4.1.3 Response (200 OK - Personalized)

Headers:

```
Cache-Control: private, max-age=30
ETag: "W/\"feed-v1-tenant_abc-user_a1b2-v42\""
X-Feed-Type: personalized
X-Request-ID: req_abc123
```

Body:

```
{
  "items": [
    {
      "id": "vid_123",
      "title": "Amazing Goal",
      "thumbnail_url": "...",
      "duration_seconds": 30,
      "creator": {"id":"c_1","name":"Name"},
      "tracking_token":"tok_abc",
      "score": 0.72
    }
  ],
  "next_cursor": "eyJzY29yZSI6ODAsImxhc3RfaWQiOiJ2aWRfMTIzIn0=",
  "has_more": true,
  "degraded": false,
  "request_id": "req_abc123"
}
```

Listing 5: Personalized feed response

4.1.4 304 Not Modified Response

If `If-None-Match` matches server ETag:

```
HTTP/1.1 304 Not Modified
Cache-Control: private, max-age=30
```

No body (bandwidth optimized).

4.1.5 Fallback / Degraded Response

On ranking/Redis error return 200 with `degraded: true` and `feed_type: default` (not a 500).

Example header:

```
Cache-Control: public, max-age=30, stale-while-revalidate=15
X-Feed-Type: default
X-Degraded: true
```

4.1.6 Error Semantics

- **400** — Bad request (invalid cursor)
- **401/403** — Auth/tenant errors
- **429** — Rate limit (e.g. >2 refreshes/sec)
- **500** — True server meltdown (rare). Prefer returning fallback with 200 when possible.

5 CMS Configuration

5.1 CMS Capabilities

- Upload / edit content metadata (title, tags, maturity)
- Create campaigns / editorial boosts (priority slots or boost weights)
- Per-tenant ranking weight overrides (recency vs affinity vs popularity)
- Maturity / safety filters per tenant
- Tenant-level enable/disable personalization + rollout %

5.2 How It's Stored

- CMS writes to `content_items` and `tenant_configs` in Postgres.
- CMS emits `REFRESH_CANDIDATES:{tenant}` and `CONFIG_UPDATE:{tenant}` events that Feed API nodes listen to (pub/sub).

5.3 UI/UX for Content Managers

- **Editor view:** Pick content → set editorial boost (0.0–3.0)
- **Tenant config screen:** Set `rollout_pct`, toggle personalization, set default weights, set maturity policy
- **Preview:** Show "preview feed" for tenant with simulation toggles (shadow mode)

6 Trade-offs & Decisions

Decision	Optimized for	Deferred	Reversibility
Simple weighted scoring vs ML	Latency, explainability, ship speed	ML bandits / trained models	High — ranking pluggable
Redis caching (inputs) vs full precomputation	Freshness (60s) & latency	Full batch precomputed per-user	Medium — can add precompute later
In-memory flag eval (L1) vs external FF service	Zero-latency kill switch	External flag integration	High — swap backend
Postgres + indexes as content index	Simplicity & correctness	ES for complex search	Medium — add ES later
Per-request ranking (bounded)	Personalization accuracy	Batch ranking for scale	High — hybrid possible

Table 2: Trade-offs and Decisions

Reversibility: Design intentionally keeps interfaces and modules replaceable (ranking, signal store, candidate source).

7 Rollout & Observability

7.1 Rollout Strategy

- **Phase 0 — Shadow:** Compute personalized feed but return default. Log differences.
- **Phase 1 — Gradual % rollouts per tenant:** Use `hash(user_id) % 100 < rollout_pct`.
- **Phase 2 — Tenant opt-in:** Toggle for specific tenants.
- **Kill Switch:** Global flag to immediately force default/trending feed.

7.2 Observability / Dashboards (Ship P0)

- **Feed API latencies** (p50, p95, p99) — per tenant and global
- **Error / fallback rate** — % requests served default or marked degraded
- **Cache hit rates** — Candidate L2, Signal hits
- **Personalized ratio** — % of requests personalized vs default
- **Cold-start CTR** — CTR for users with <5 signals
- **Feed composition** — age distribution, tag distribution (to detect bubbles)
- **Traffic & RPS per tenant**

P1 metrics: completion rate by feed type, long-term retention delta, A/B lift

7.3 Alerts

- p95 > 250ms
- p99 > 600ms
- Fallback rate > 0.5% (or tenant-specific SLAs)
- Candidate cache invalidation failure rate > threshold

8 Resilience Patterns & Operational Notes

- **Graceful Degradation:** On any critical dependency failure (Redis high latency, ranking timeout) immediately return default trending feed with `degraded:true`.
- **Circuit Breaker:** Wrap Redis and Ranking calls; thresholds trip to fast-fallback.
- **Timeouts & Budgets:** Redis read timeout ∼5-10ms; Candidate fetch ∼10ms; ranking budget ≤100ms.
- **Bulkheads:** Isolate ingestion workers from feed serving resources.
- **Idempotency:** Events are idempotent or deduplicated in SignalProcessor.
- **Backpressure:** If origin overloaded, serve cached/default feeds rather than failing.
- **ETags & Conditional GETs:** Reduce bandwidth and detect identical responses.
- **Monitoring & Chaos:** Run failure injection (Redis slow, Kafka backlog) and verify fallback path.

9 Future Work

What we'd do with more time:

- **ML & Bandits:** Two-tower or GBDT models for ranking; contextual bandits for exploration-exploitation.
- **Real-time Session Signals:** Move user session signals from 5-min lag to sub-10s (Kafka Streams / Flink).
- **Offline Eval & Replayer:** Tool to replay historical traffic to compare candidate ranking variants before rollout.
- **Content Embeddings:** Semantic similarity for cold-start & diversity.
- **Explainability UI:** "Why this video?" for debugging and product trust.
- **A/B & Causal Analytics:** Robust treatment effect measurement for ranking changes.