Workshop 2

Data Architecture for a Music Streaming Platform based on Spotify

Brayan Stiven Yate Prada – Student ID: 20192020151
Holman Andres Alvarado Diaz – Student ID: 20201020032
Universidad Distrital Francisco José de Caldas

Faculty of Engineering
Course: Databases II
Professor: Carlos Andrés Sierra Virgüez

May 29, 2025

Contents

1 Data System Architecture

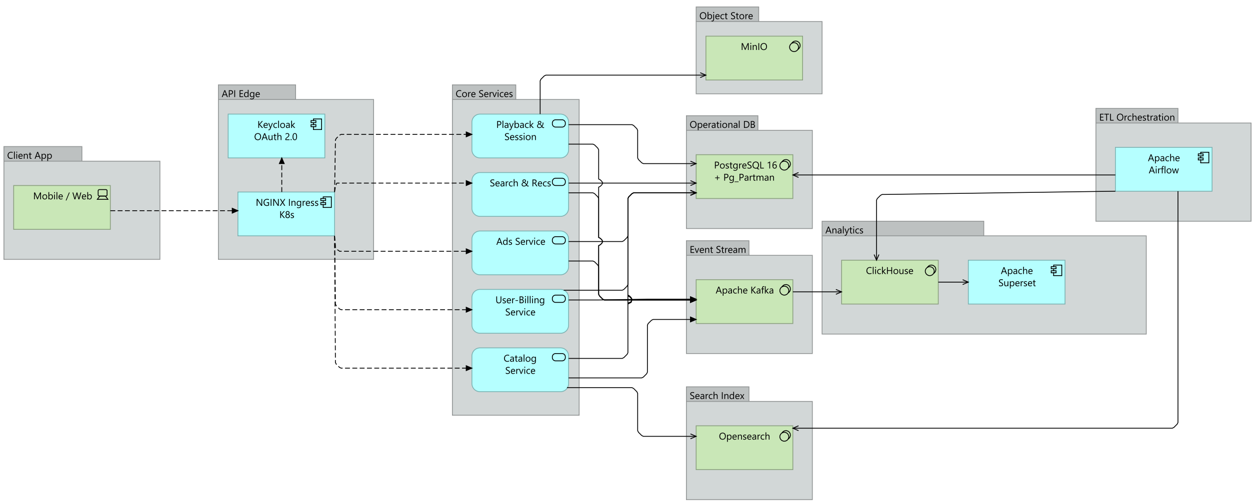

Figure 1: High-level Architecture Diagram

1.1 Justification for Each Technology

Component	Technology	Reason	Primary Data Stored
API Edge	**NGINX Ingress Controller**	Ships with every Kubernetes distribution; trivial TLS termination; rich ecosystem.	Proxies all HTTPS traffic.
Authentication	**Keycloak**	OAuth2/OIDC server; social log-ins, multi-factor authentication.	User credentials and tokens (hashed / JWT).
Operational SQL	**PostgreSQL 16 + pg_partman**	Logical/physical replication; partitioning and native JSONB; easy to query.	Users, subscriptions, catalog (artists, albums, tracks), playlists, payments, ads, social graph.
Search / Auto-complete	**OpenSearch**	Apache-licensed Elasticsearch fork; scales from single node to multi-node; SQL plug-in for analysts.	Denormalised documents for tracks, artists, playlists.
Object Storage	**MinIO**	Lightweight S3 clone; single binary; erasure coding; deployable on-prem or on any cloud.	Audio files, cover art, long-term raw logs, GDPR exports.

Event Streaming	**Apache Kafka**	Industry-standard durable, ordered, replayable event log; hundreds of client SDKs.	Play/skip events, ad impressions, change-data capture (CDC) from PostgreSQL via Debezium.
Real-Time / Batch Analytics	**ClickHouse**	Blazing fast, column-oriented, SQL-native; licence-free; handles $> 100\,000$ inserts/s and sub-second dashboards.	Aggregated play counts, creator dashboards, advertising metrics, Wrapped-style reports.
Orchestration / ETL	**Apache Airflow**	Familiar DAG UI; Python-first; vast provider library (Debezium → ClickHouse, backups, ML).	Schedules nightly royalty jobs, table vacuums, data-quality checks.
Business Intelligence	**Apache Superset**	Modern OSS BI; drop-in ClickHouse driver; RBAC and SSO via Keycloak.	Front-end only – reads ClickHouse for charts.

1.2 End-to-End Data Flow (grounded in the schema)

Creator Upload

1. The *Catalog Service* receives the audio file and its metadata.
2. Binary data are stored in a MinIO bucket `/audio/`.
3. Rows for artist, album and track are inserted into PostgreSQL (ER diagram tables).
4. A Debezium connector streams these row-level changes to Kafka.
5. A lightweight Flink job flattens the JSON and pushes it to OpenSearch, allowing the track to become searchable within seconds.

Listener Playback

1. The *Playback Service* retrieves `track_id`, rights and bitrate information from PostgreSQL, signs a presigned MinIO URL and returns it to the client.
2. The client begins streaming; every 5 s it emits a `play-tick` JSON message to Kafka.

Real-Time Aggregation ClickHouse materialised views ingest the `play-tick` topic directly—no Spark or Flink cluster is required for the minimum viable product (MVP). Dashboards in Superset refresh automatically for artists and advertising operations.

Nightly Jobs *(Airflow)*

- `royalty_etl.py`: SQL executed in ClickHouse summarises plays by `track_id` and writes the payout CSV to the PostgreSQL `payouts` table.
- `search_reindex.py`: Performs a bulk re-sync of OpenSearch from PostgreSQL to catch edge cases.
- **Back-ups**: pg_dump streams are stored as compressed objects in MinIO.

Social Graph and Recommendations The `user_follow` bridge table lives in PostgreSQL; for the MVP we run simple collaborative-filtering in Python inside Airflow once per day and write top-N recommendations into `user_recs`. When sub-50 ms cold-start latencies become necessary these features can migrate to Redis, but not in version 1.

1.3 Why This Trimmed Stack Meets Our Goals

- **Feasible for a small team:** only six stateful services (PostgreSQL, MinIO, Kafka, OpenSearch, ClickHouse, Keycloak). All ship Helm charts and run on three modest VMs or a single Kubernetes cluster.
- **Single source of truth:** PostgreSQL hosts the entire ER diagram with just two bridge tables (`playlist_track`, `user_follow`) and one fact table (`ad_impression`). OLTP rows are partitioned by tenant (`user_id mod N`) enabling future shard-out.
- **Analytics without Hadoop:** ClickHouse reads Kafka directly, avoiding Spark/HDFS.
- **Search within seconds:** Debezium + Kafka keeps OpenSearch in near-real-time sync.
- **Fully open-source:** no licences, no managed-only products.
- **Clear upgrade path:** if traffic explodes we can add Postgres replicas, expand Kafka, deploy more ClickHouse shards or adopt Citus; none require a rewrite.

2 Information Requirements

#	Information Type	Description	Storage	Business Link	Key User Stories
1	**Catalog Metadata**	Track, album, artist info. Enables search & legal playback.	PostgreSQL (`tracks`, `albums`, `artists`); OpenSearch cache.	Global audio library; Key Partners.	Search; Playback; Upload Audio.
2	**Audio References**	Presigned URLs + bitrate list for playback via CDN or MinIO.	Generated by Playback Service from object metadata.	Seamless streaming.	Playback; Device Integration.
3	**User Profile**	Display name, avatar, language, tier, parental controls. UI personalization + policy enforcement.	PostgreSQL (`users`).	Personalisation; Segmentation.	Profile Management; Register; Parental Control.
4	**Subscription & Billing**	Plan, renewal, token, grace state. Controls privileges and ad access.	PostgreSQL (`subscriptions`, `payment_method`).	Recurring revenue.	Subscription Management; Premium Access.

5	**Playback State**	Position, device ID, network, heartbeat. Enables adaptive bitrate + resume.	Kafka → ClickHouse or in-memory cache.	Playback continuity.	Adaptive Streaming; Resume Playback.
6	**Suggestion**	Ranked track IDs + labels (e.g., repeat, trending).	PostgreSQL JSONB (`user_recs`).	Discovery; ML-driven value.	Discover; Playlist Creation.
7	**Search Suggestions**	Tokens, fuzzy matches (tracks, artists, playlists).	OpenSearch.	Search efficiency.	Content Search.
8	**Social Graph**	Follows, playlist rights, friend activity.	PostgreSQL (`user_follow`, `playlist_track`); Kafka.	Community features.	Follow Users; Collaborative Playlists.
9	**Ads & Creatives**	Targeting, CPM, assets; user context.	PostgreSQL (`ad_campaign`, `ad_creative`) + in-memory rules.	Ad revenue.	Launch Campaigns; Real-Time Ads.
10	**Creator Analytics**	Play counts, geo maps, promo-lift.	ClickHouse views via Superset.	Creator tools; Partner support.	View Stats; Track Promotion Impact.
11	**Royalty Reports**	Stream counts, payouts per rightsholder.	Airflow ETL → PostgreSQL (`payouts`); MinIO.	Royalty costs.	(Internal workflows).
12	**Compliance Logs**	Admin actions, GDPR links, token audits.	Append-only Postgres + WORM MinIO.	Trust; Legal compliance.	Data Requests; Audit Trails.

Traceability to Business Model and User Stories

- **Value Proposition** ("global audio library", ad-free Hi-Fi, discovery) requires fast access to items 1, 2, 6 and 5.
- **Revenue Streams** (subscriptions & ads) rely on items 4 and 9.
- **Customer Relationships / Segments** (community, stats) depend on items 3, 8 and 10.
- **Cost Structure / Key Partners** (royalty payouts) require items 1 and 11.
- **Compliance & Trust** are covered by item 12 and secured payment tokens.

3 Query Proposals

3.1 User and Subscription Management (PostgreSQL)

```
-- 1A. Premium users in top-5 high-usage countries
```

```
WITH top_countries AS (
    SELECT country
    FROM   play_events_daily  -- daily roll-up loaded by Airflow
    ORDER  BY total_plays DESC
    LIMIT  5
)
SELECT u.user_id,
       u.display_name,
       s.plan,
       s.country
FROM   users u
JOIN   subscriptions s USING (user_id)
WHERE  s.status   = ’active’
  AND  s.plan     = ’premium’
  AND  s.country  IN (SELECT country FROM top_countries);
```

Purpose: identify premium markets for targeted marketing.

```
-- 2A. Potential account sharing (>=3 countries in 30 days)
SELECT user_id,
       COUNT(DISTINCT ip_country) AS unique_countries_30d
FROM   session_logs PARTITION FOR (CURRENT_DATE - INTERVAL ’30 days’)
GROUP  BY user_id
HAVING COUNT(DISTINCT ip_country) > 3;
```

Purpose: detect family-plan abuse.

3.2 Search and Discovery (OpenSearch)

```
GET /tracks/_search
{
  "query": {
    "function_score": {
      "query": { "match": { "title": "drake" }},
      "field_value_factor": {
        "field":   "play_count",
        "factor":  0.1,
        "modifier":"sqrt",
        "missing": 1
      },
      "boost_mode": "multiply"
    }
  },
  "sort": [
    { "_score": "desc" },
    { "release_date": "desc" }
```

```
  ]
}
```

Purpose: relevancy-boosted autocomplete.

3.3 Playback Behaviour (ClickHouse)

```
-- 3A. Early-churn indicator for new sign-ups (<=30s skips)
SELECT
    user_id,
    COUNT()            AS total_plays,
    sum(duration < 30) AS short_skips
FROM  play_events
WHERE signup_date >= today() - 7
GROUP BY user_id
HAVING total_plays > 5;

-- 3B. Rapid-skip bot detection (last 10 min)
SELECT *
FROM   play_events
PREWHERE user_id = 'u_999'
    AND event_time > now() - INTERVAL 10 MINUTE
WHERE  duration < 15;
```

3.4 Campaign and Cohort Analytics (ClickHouse)

```
-- 4A. Retention by sign-up cohort
WITH cohort AS (
  SELECT user_id,
         toDate(min(event_time)) AS cohort_date
  FROM   play_events
  GROUP  BY user_id
)
SELECT cohort_date,
       activity_date,
       uniqExact(user_id) AS active_users
FROM (
  SELECT user_id,
         toDate(event_time) AS activity_date
  FROM   play_events
) e
JOIN cohort USING user_id
GROUP BY cohort_date, activity_date
ORDER BY cohort_date, activity_date;
```

```sql
-- 4B. Streams attributable to promotional campaigns
SELECT
    t.artist_id,
    t.track_id,
    c.campaign_id,
    count() AS plays_during_campaign
FROM  play_events      AS pe
JOIN  tracks           AS t  ON pe.track_id  = t.track_id
JOIN  ad_campaign      AS c  ON pe.event_time BETWEEN c.start_date AND c.end_date
WHERE c.campaign_type  = 'promotional'
GROUP BY
    t.artist_id, t.track_id, c.campaign_id;
```

3.5 Royalties and Finance (PostgreSQL)

```sql
-- 5A. Royalty calculation weighted by local rates
SELECT
    t.artist_id,
    t.track_id,
    pe.country,
    COUNT(*) * r.rate AS royalty_amount
FROM   play_events_monthly pe  -- ETL roll-up table
JOIN   tracks            t  ON pe.track_id = t.track_id
JOIN   royalty_rates     r  ON r.country   = pe.country
                           AND r.artist_id = t.artist_id
WHERE  pe.month = '2025-05'
GROUP  BY t.artist_id, t.track_id, pe.country, r.rate;

-- 5B. Cross-check: plays vs payments divergence >1 %
WITH plays AS (
    SELECT artist_id,
           COUNT(*) AS total_plays
    FROM   play_events_monthly
    WHERE  month = '2025-05'
    GROUP  BY artist_id
)
SELECT p.artist_id,
       SUM(ap.royalty_amount) AS total_paid,
       total_plays
FROM   artist_payouts ap
JOIN   plays          p USING (artist_id)
GROUP  BY p.artist_id, total_plays
HAVING ABS(SUM(ap.royalty_amount) / NULLIF(total_plays,0) - expected_rate) > 0.01;
```

Technology Mapping

Use-Case	Query Engine	Rationale
Operational joins, financial exactness	PostgreSQL	ACID semantics and referential integrity.
Event-stream ad-hoc, cohort, fraud analysis	ClickHouse	100 000 RPS ingestion and sub-second scans.
Search / autocomplete	OpenSearch	Token scoring and fuzzy match.
Event ingestion	Kafka	Decouples writes and enables real-time pipelines.

4 Improvements Over Workshop 1

4.1 Initial Database Architecture

Designing a database architecture for a large-scale music-streaming platform requires a balance of technical and business constraints: ultra-low playback latency, strict compliance with regulations such as GDPR, global scalability and financial integrity. The architecture therefore distributes data into specialised layers, selecting *only* open-source technologies to ensure flexibility, transparency and long-term sustainability.

4.2 Operating Model and Systemic Context

Spotify adopts a *freemium* model: free users are monetised through advertising while Premium users pay recurring subscriptions. Roughly 70 % of revenue is allocated to royalty payments for artists and record labels.

The data ecosystem involves multiple actors:

- End users (mobile / web clients)
- Backend and distributed storage layer
- Content creators
- Advertising subsystem
- Analytics and reporting platform
- Contractual royalty rules

4.3 Information Flow: Inputs, Outputs and Interactions

The system operates via asynchronous, bidirectional data flows. Principal routes are:

Inputs
: Multimedia files (audio, cover art, metadata); user events (play, like, follow); financial information and credentials.

Outputs
: Optimised music playback; personalised recommendations; analytical reports; real-time targeted advertisements.

Interactions

Client → Application → Microservice backend → Distributed storage, with lateral interactions to Advertising and Recommendation subsystems.

4.4 Critical Bottlenecks Identified

Subsystem	Problem	Risk	Recommendations
Intensive social-graph reads	Notable latencies (P99 > 200 ms)	User experience degradation	Partition graph, introduce specialised cache (e.g. Dgraph or RedisGraph).
Advertising	Cache saturation under high concurrency	Ad-decision latency > 50 ms	Hierarchical LRU cache, pre-computed segments.
Streaming	Real-time CDN and DRM selection	Start time > 300 ms causes abandonment	Edge nodes with pre-signed URLs; regional CDN hints.
Royalties	Massive event calculation	Legal/reputational risk	Incremental aggregation, ClickHouse materialised views, parallel pipelines.
GDPR	Data proliferation	Non-compliance fines	Unified subject-ID mapping and automated data-subject export routines.

4.5 System Stress Points

Potential Crisis	Technical and Business Cause
Explosive concurrent-user growth	Recommendation, playback or scaling failures due to horizontal stress.
Advertising-decision delays	Revenue loss and poor user experience for free tier.
Inaccurate royalty calculations	Legal and reputational consequences with rights holders.
Content-ingestion saturation (10 TB / day)	Bottlenecks in validation, ML pipelines and media persistence.
Social-graph synchronisation errors	Inconsistent likes/follows (latency > 5 s).
Autoscaling failures	Frozen experience due to unbalanced or mis-distributed services.

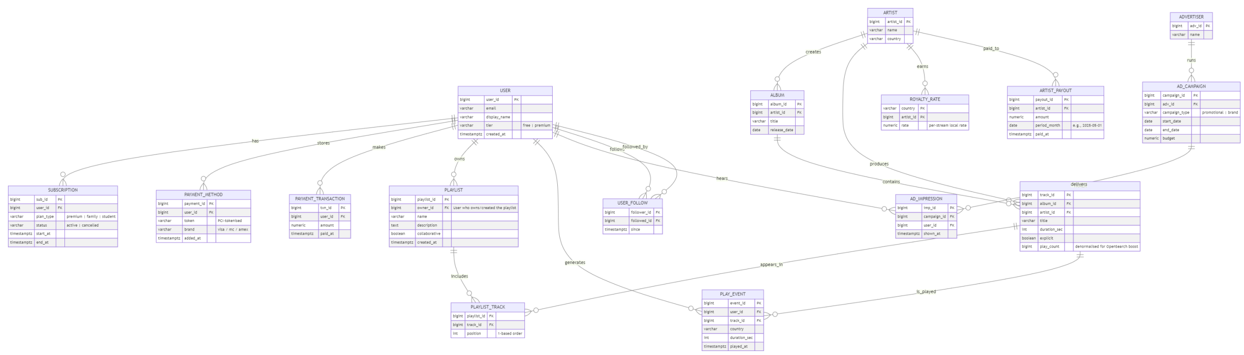

Figure 2: Caption

4.6 Diagnosis of the Central Problem

Spotify faces constant tension among horizontal scalability, ultra-low latency and contractual compliance: every new user, artist or advertising campaign represents a potential distributed load hotspot.

4.7 Possible Technical Solutions

1. Strengthen distributed cache for personalization and advertising.
2. Apply edge-computing for initial playback near the client.
3. Decouple royalty pipelines into asynchronous, parallel layers.
4. Deploy full distributed tracing (OpenTelemetry).
5. Optimize graph-partitioning algorithms and advertising clusters.

Update: A systemic problem analysis has been added, identifying key points, bottlenecks, information flows and a clearer operating model, clarifying the issue to solve and updating the ER diagram accordingly.

4.8 ER Diagram

4.9 Overview of the Relational Model

The proposed relational model structures the critical information of a musical-content, advertising and social-interaction platform, based on a microservices architecture designed for large-scale operation. The implementation relies on PostgreSQL 16, taking advantage of referential integrity, ACID transactions, advanced types (e.g. JSONB) and native sharding (via `pg_partman` or Citus).

The model is organised into six functional domains, each with independent entities and clear relationships, enabling horizontal scalability, auditability and extensibility in line with distributed-design principles and regulatory compliance.

4.10 Description of Entities and Relationships

User and Subscription Domain

- **USER**: end-user profiles, including unique identifier, subscription type (free / premium) and creation date.
- **SUBSCRIPTION**: transactional table storing the history of active and cancelled plans per user.
- **PAYMENT_METHOD** and **PAYMENT_TRANSACTION**: manage tokenised payment methods (PCI-DSS) and related transactions, linked to `USER` via foreign keys.

This flow provides complete traceability of the account life-cycle and its monetisation.

Catalog Domain

- **ARTIST**, **ALBUM** and **TRACK**: compose the musical catalogue hierarchy. `TRACK` includes key metadata such as duration, explicit-content flag and a denormalised `play_count` field to improve analytical indexing in OpenSearch.
- Relationships are modelled *one-to-many* from artist to album and album to track.

Playlists and Social Domain

- **PLAYLIST**: associated with a user (`owner_id`) and supporting collaborative lists.
- **PLAYLIST_TRACK**: many-to-many bridge table preserving sequential order by `position` column.
- **USER_FOLLOW**: represents the social follow graph, using composite keys and timestamps for temporal tracking.

These entities enable social functionality and personalisation with low logical coupling.

Playback and Events Domain

- **PLAY_EVENT**: records each playback session with contextual metadata such as country, effective duration and timestamp. Modelled as a very high-cardinality event table, suitable for replication into OLAP or time-series systems.

Advertising Domain

- **ADVERTISER**, **AD_CAMPAIGN**, **AD_IMPRESSION**: model advertising campaigns and their interaction with users.
- `AD_IMPRESSION` functions as a fact table, registering which user was exposed to which campaign and when.

Royalties and Finance Domain

- **ROYALTY_RATE**: defines the per-play fee by artist and country.
- **ARTIST_PAYOUT**: represents consolidated monthly payments to each artist.

Both entities are populated from aggregated `PLAY_EVENT` data, enabling contractual and regional revenue calculations with full financial traceability.

4.11 Storage Strategies and Data Flow

- Implemented on PostgreSQL 16 with horizontal sharding by `user_id` in high-volume tables (`PLAY_EVENT`, `USER_FOLLOW`, `AD_IMPRESSION`).
- Events are written to PostgreSQL and synchronised to external analytical stores (ClickHouse, OpenSearch) for real-time exploitation.
- Sensitive tables such as `PAYMENT_METHOD` integrate external tokenisation and at-rest encryption (AES-256).
- Retention, auditing (`created_at`, `paid_at`, `shown_at`) and strong consistency policies are enforced on critical tables (`SUBSCRIPTION`, `TRACK`, `ARTIST_PAYOUT`).

5 References

References

[1] Corporate Finance Institute. *Business Model Canvas Examples*. Available at: https://corporatefinanceinstitute.com/resources/management/business-model-canvas-examples/

[2] Business Model Analyst. *Spotify Business Model*. Available at: https://businessmodelanalyst.com/spotify-business-model/

[3] Music Business Research. (19 August 2024). *The Music Streaming Economy Part 10: Spotify's Business Model*. Available at: https://musicbusinessresearch.wordpress.com/2024/08/19/the-music-streaming-economy-part-10-spotifys-business-model/

[4] Investopedia. *How Spotify Makes Money*. Available at: https://www.investopedia.com/articles/investing/120314/spotify-makes-internet-music-make-money.asp

[5] IIDE. *Business Model of Spotify*. Available at: https://iide.co/case-studies/business-model-of-spotify/

[6] GrowthX Club. *Spotify Business Model*. Available at: https://growthx.club/blog/spotify-business-model#spotify-revenue-model

[7] Software Developer Diaries. *How Spotify's Playback Works Under the Hood*. YouTube video. Available at: https://www.youtube.com/watch?v=K26bGXVR-mE

[8] Lopez, M. A. *Deep Dive into the Tech Stack Used by Spotify*. LinkedIn article. Available at: https://www.linkedin.com/pulse/deep-dive-tech-stack-used-spotify-marny-a-lopez

[9] Intuji. *How Does Spotify Work? – Tech Stack Explored*. Available at: https://intuji.com/how-does-spotify-work-tech-stack-explored/