Portfolio

Deep technical investigations, infrastructure decisions, and systems work with measurable operational impact.

CASE STUDIES

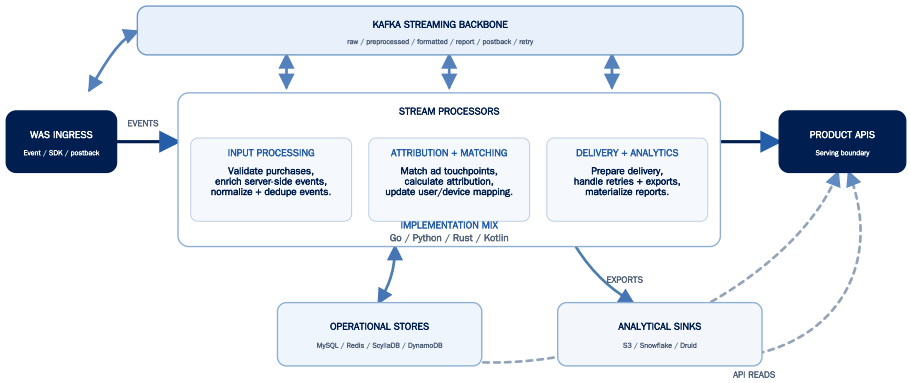
SERVICE ARCHITECTURE / 2026

[Service Architecture] Airbridge Datapipeline

Airbridge's data pipeline turns incoming events into attribution decisions, partner postbacks, reporting data, and product-facing state.

AIRBRIDGE REAL-TIME DATA PIPELINE ARCHITECTURE

A compact map of the main processing boundaries and data movement.



EVENT LIFECYCLE

The ingest APIs receive SDK events, server-side events, redirects, and postback validation requests, then create the raw event streams in Kafka. The pipeline validates purchases, enriches server-side events, normalizes and deduplicates raw events, then splits work into attribution calculation and touchpoint processing. The downstream workers produce formatted events, report outputs, user/device mapping state, and partner delivery work.

STORES AND SERVING BOUNDARIES

Each stage is a domain worker behind a Kafka topic boundary. Purchase validation, server-side event enrichment, preprocessing, attribution/touchpoint processing, user/device mapping updates, partner delivery, analytical export, and report materialization each have separate failure points and scaling units. Operational stores hold configuration, checkpoints, user/device mapping, and low-latency serving state. S3, Snowflake, and Druid-style analytical stores keep raw events, formatted events, reports, and delivery logs for analytics and reporting. Product APIs read those stores and materialized outputs instead of recalculating attribution at request time.

- After an event is accepted, ownership is explicit: ingestion/preprocessing, attribution, touchpoint processing, partner delivery, and analytical loading each have responsible workers, output streams, and observable failure points.
- Kafka is not a single middle buffer. It is used across raw, preprocessed, formatted, report, postback, and retry paths to separate processing stages and make replay or retry boundaries explicit.
- After normalization, attribution calculation and touchpoint processing are handled as separate responsibilities, so ingest/preprocessing changes and attribution business-rule changes can be deployed and inspected independently.
- Partner delivery runs through separate work queues, retry handling, and failover paths so external API latency or outages do not directly block the main event path or report generation.
- Configuration, checkpoints, user/device mapping, and low-latency serving state remain in operational stores, while raw/formatted events, reports, and delivery logs flow into analytical stores, separating product-serving reads from BI and reporting reads.

DATABASE DEEP DIVE / 2024

[Digging Deep] Optimizing ScyllaDB Materialized Views

Investigated ScyllaDB materialized views that had grown disproportionately compared with their base tables, creating storage pressure and avoidable scale-out risk.

MATERIALIZED-VIEW REMOVAL DECISION

The investigation moved from storage skew to root-cause validation, then compared whether the read path could be served without the materialized view.

STORAGE DIAGNOSIS	1 Table-level skew The view consumed far more disk than its base table.	2 Update amplification Changing view keys creates delete and insert maintenance work.	3 Retention window Compaction must preserve tombstones until gc_grace_seconds allows purge.	4 Capacity pressure The retained tombstones made scale-out look necessary.
VALIDATED REPLACEMENT PATH	1 Base-table query A direct query covered the product access pattern.	2 Query tracing Tracing confirmed the simpler path stayed within acceptable read cost.	3 Negligible delta Observed performance was close enough for production use.	4 Remove view Reduced storage pressure by simplifying the data model.

CONTEXT

The starting point was cluster capacity, not query latency. Table-level disk metrics showed that one materialized-view family was much larger than its base table, so I first checked whether the product read path truly needed the view. After validating the direct base-table query with query tracing, I tied the storage behavior back to ScyllaDB internals: materialized-view key changes create old-view-row deletes and new-view-row inserts, and the resulting tombstones can remain until gc_grace_seconds allows cleanup.

RESULT

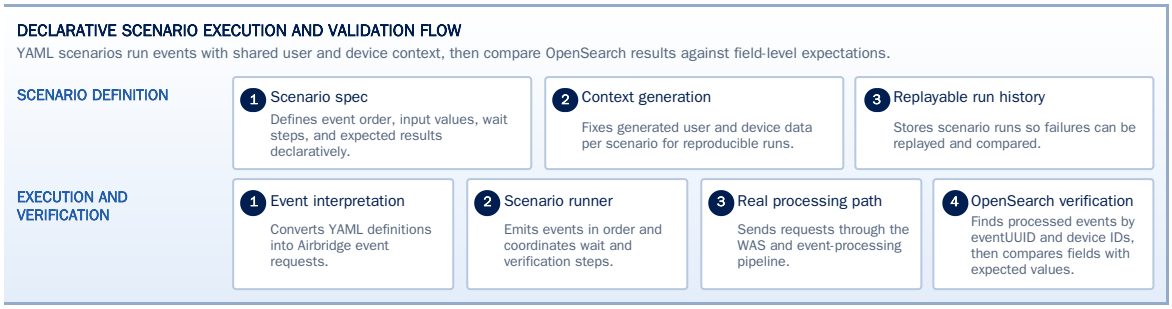
The decision changed from adding nodes to simplifying the data model. Once the base-table query proved sufficient for the access pattern, removing the materialized view eliminated a write-amplified storage path and reduced cluster disk usage by 30%, about 9 TiB.

- Treated materialized views as read models with ongoing write and storage cost, not as free query shortcuts.
- Added a review criterion before scaling the cluster: confirm whether the access pattern still justifies each derived table.
- Left an operational example where simplifying the data model was the smaller change than adding infrastructure capacity.

QA PLATFORM / 2024-2026

[Innovation] Implementation of a declarative QA system

Built a declarative QA system that runs end-to-end data pipeline scenarios and asserts processing results from OpenSearch in real time.



CONTEXT

Before this system, QA often meant manually crafting several related events and checking the asynchronous pipeline result in OpenSearch afterward. The system models ordered user journeys, such as install, signup, purchase, and postback scenarios, as one executable definition. It fixes user, device, app, region, and branch context, then sends the events through the real ingest APIs and event-processing pipeline.

RESULT

Validation is based on the OpenSearch documents produced by the pipeline, not UI state or mocked outputs. The runner finds processed events by eventUUID and device identifiers, compares field-level expectations and downstream invariants, and keeps run history, request traces, and failed-step details so engineers can rerun a scenario and understand exactly where behavior changed.

- Converted YAML or editor-authored input into executable E2E scenarios, replacing manual event creation with repeatable tests.
- Handled event request generation, ordered execution, asynchronous wait steps, OpenSearch lookup, and field-level assertions in one runner.
- Fixed generated user data, device metadata, app settings, data source, region, and feature branch as scenario context so the same conditions can be rerun for regression checks.
- Stored failed steps, request traces, and OpenSearch results together so engineers can see which input failed to produce the expected pipeline output.