AllData 2026

Large-Scale Offline Data Handling:
An Event-Streaming-Based Kappa Architecture

Quan Zhou · Pradeep Akkinepally · Monica Dhanaraj · Dyutimoy Sarkar · Muthaiyan Thandapani

eBay Inc. · San Jose, USA

Bridging massive offline data warehouses with heavy-weight production services
via asynchronous event streaming — without rewriting your application.

Motivation: The Offline-vs-Production Logic Gap

Re-processing massive historical data against complex, heavy-weight service applications is engineering-hard.

The Problem

- Logic lives in heavy-weight services, not SQL transforms
- Intricate dependencies, local state, external integrations
- Hard to port to Spark / Hadoop without major rewrite
- Refactoring spawns a dual codebase: real-time + offline
- Logic drift between codebases is inevitable over time

What We Want

- Run production logic over offline data, unchanged
- 100% logical parity – zero divergence drift
- High throughput on multi-million-row jobs
- Elastic scaling, isolated failures, ops-friendly
- One codebase – sustainable engineering velocity

Target Use Cases

Three recurring patterns motivate large-scale offline execution of production logic.

Historical Simulation

Validate new features or business logic against historical, point-in-time snapshotted events.

Data Backfill

Re-populate missing or corrupted data on the DW after a schema change or data-loss event.

Periodic Batch Processing

Heavy-lift periodic computation that must run the same complex service logic over large windows of offline data.

Three Architectural Options We Evaluated

I. Spark-Native (Compute-to-Data)	**Mechanism: Refactor service logic into a Spark-native job; move compute to where the data lives.** Trade-off: 22M records evaluated in <10s for atomic web APIs; but produces a dual codebase and unbounded logic drift for complex services.
II. Micro-Batch + Spark JDBC	**Mechanism: Spark executors read partitions and make synchronous HTTP/gRPC calls to the production service.** Trade-off: High logic parity — same endpoints as prod — but throughput is throttled by JDBC and the service; a single point of failure.
III. Kappa Architecture (Event Streaming)	**Mechanism: Treat offline DW data as a stream; wrap the production service as an async Kafka consumer.** Trade-off: Total logic parity, elastic scaling, dependency isolation. Pays a price in operational complexity and serialization.

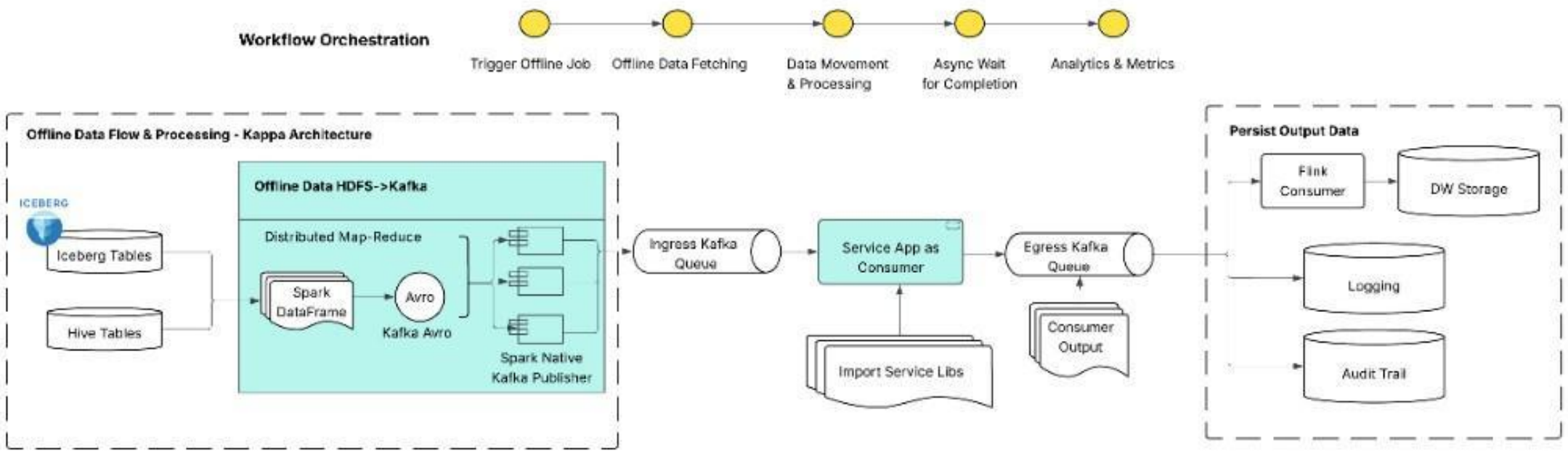
Comparison: Why Kappa Wins the Election

Kappa is the only option that scores well on parity, throughput, and maintainability simultaneously.

Feature	Spark Native	Micro-Batch	Kappa
Logic Parity	Low	High	**Total**
Throughput	Very High	Low	**High**
Maintenance	High	Moderate	**Minimal**
Scaling	Map-Reduce	API/JDBC bottleneck	**Elastic**

Takeaway: Kappa pairs production-grade logical parity with elastic horizontal scaling — the other options force a trade-off between the two.

System Architecture: End-to-End



Five decoupled components — storage is separated from execution:

- Workflow Orchestration: Airflow drives the E2E lifecycle: trigger, fetch, dispatch, await, analyze.
- Data Movement Pipeline: Distributed Spark fetches PiT snapshots and publishes them to the Ingress Kafka queue (Avro).
- Service-as-Consumer: Production service is packaged as a scalable async Kafka consumer — zero logic rewrite.
- Data Persistence: Lightweight Flink stream processor sinks egress events back to HDFS / DW for long-term storage.
- Analytics Layer: Hive / Spark SQL on landed tables for metrics, validation, and ad-hoc analysis.

Engineering Challenges We Solved

Backpressure Handling

Spark ingress out-paces the heavy-weight consumer.

- Tuned max.poll.records to align with p99 service latency
- Bounded in-flight work → no session-timeout cascades
- Application-level rate limiter pauses consumption when local thread pools saturate
- Blocking queue inside consumer evens out CPU saturation to a steady ~50%

Schema Transformation

Loosely-typed DW frames must hit a strict event contract.

- Mandatory Avro validation at the ingress publisher
- Central Schema Registry governs evolution & compatibility
- Explicit type coercion (e.g., Hive timestamp → Avro long) and null-safety checks
- Protects complex consumers from malformed records that would otherwise crash production logic

Trade-off: asynchronous ops complexity → invest in Ingress/Egress offset tracking and observability.

Performance Benchmarking

100 partitions per topic · Kubernetes consumer fleet (24 → 48 pods, 2 vCPU / 8 GB each) · HPA on CPU + consumer lag.

Data Volume	Job Duration	Throughput
1,000,000 (1M)	7 min 03 sec	2,400 msgs/sec
3,000,000 (3M)	12 min 45 sec	9,000 msgs/sec
5,000,000 (5M)	15 min 05 sec	10,900 msgs/sec
10,000,000 (10M)	18 min 05 sec	11,900 msgs/sec

> 10 M
records / job

< 15 min
core-execution SLA

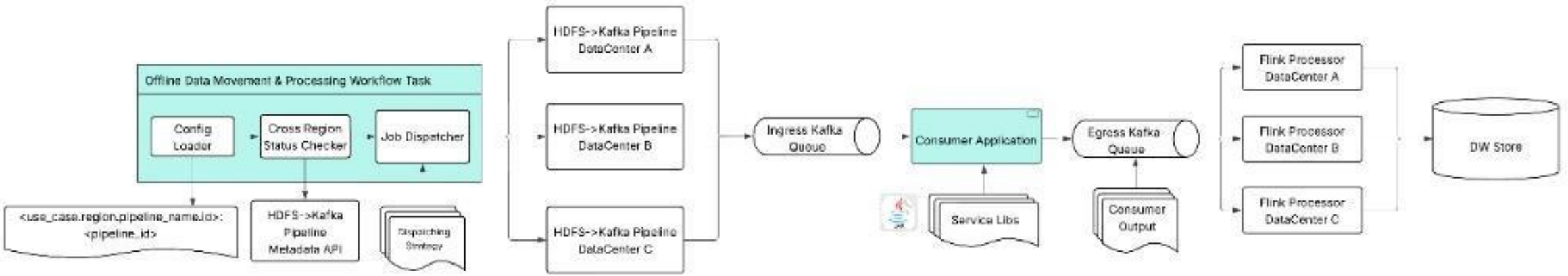
> 99 %
job success rate

< 1 %
E2E data loss (at-least-once)

- Sub-linear job-time vs. data-volume → confirms horizontal elasticity from Spark publishers + K8s HPA consumers
- Throughput climbs 2,400 → 11,900 msgs/sec without changing service-application code
- HPA · idle-time scale-down cuts provisioned cores by ~50% while preserving SLA

Cross-Region: Resilience + Availability + Performance

Decoupling DW (storage) from the service (compute) lets us treat data centers as a single elastic execution pool.



Active-Active Availability

If a DC fails, the orchestrator transparently reroutes pending jobs to a healthy region — same logic, same consumers.

Intelligent Headroom Dispatch

Dispatcher routes each job to the region with the highest $\text{Headroom}_r = \text{Cap}_{max} - (\text{Jobs}_a{}^{ct}{}_i{}^{v}{}_e + \text{Jobsq}_{ueue}d)$, avoiding bulk-submission hot spots.

Split-and-Conquer Speedup

30-day historical job split into N child jobs across N regions → ingress/egress Kafka in parallel → E2E SLA reduces ~linearly with N.

Conclusion & Future Work

Conclusion

- Kappa bridges large-scale offline data with heavy-weight production services — no rewrite required
- Total logical parity by reusing the exact service libraries as Kafka consumers
- >10 M records / run · <15 min SLA · >99% job success · ~50% cost reduction via HPA
- Cross-region dispatcher delivers active-active availability and near-linear speedup
- Cost-vs-convenience trade-off is worth it: avoid dual codebases, preserve dependencies, reuse prod logic

Future Work

- Strengthen guarantees: at-least-once → exactly-once with idempotent producer + transactional egress sinks
- Cost-aware dispatcher: extend Headroom_r with $/core and carbon signals per region
- Auto schema reconciliation between Hive/Iceberg sources and Avro contracts
- Deeper observability: end-to-end offset lineage and automatic data-loss triage tooling
- Generalize the pattern to non-Java consumers and stateful streaming services

Thank you · Questions? — Quan Zhou et al. · eBay Inc. · AllData 2026