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
e-mail: {qnzhou, pakkinepally, mdhanaraj, dysarkar, muthandapani}@ebay.com

***Abstract*—Processing massive volumes of historical offline data through complex, heavy-weight service applications presents a significant engineering challenge. Traditional batch processing methods often require refactoring sophisticated production logic, leading to code duplication and maintenance overhead. This paper proposes a methodology for large-scale offline data processing utilizing a Kappa Architecture. By treating offline data warehouses (DW) as streaming sources and wrapping complex service applications as asynchronous consumers, we enable high-throughput processing without rewriting core application logic. We compare this approach against Spark-native and micro-batch paradigms, and show that it better preserves logical parity while maintaining engineering velocity. We also present a multi-region intelligent job dispatcher that improves availability and throughput.**

***Keywords*-kappa architecture; offline processing; event streaming; distributed systems; big data.**

I. Introduction

In the modern data landscape, organizations frequently need to re-process vast amounts of offline data stored in Data Warehouses (DW). Common use cases include:

- **Historical Simulation:** Validating new features or business logic against historical snapshotted events.
- **Data Backfill:** Re-populating missing data on DW after a schema change or data loss event [1].
- **Batch Processing:** Periodic heavy-lift computing tasks that rely on processing large scale of offline data against complex service applications [2].

The core problem arises when the processing logic is not a simple SQL transformation but resides within a **heavy-weight service application**. These applications often contain intricate dependencies, local state management, and external dependency integrations that are difficult to port to offline environment such as Apache Spark or Hadoop. Engineers are faced with the dilemma of either rewriting the application or finding a way to bridge the gap between offline data and complex business logic execution.

The remainder of the paper is organized as follows: Section II evaluates architectural options. Section III details the proposed system architecture. Section IV presents performance results. Section V describes the cross-region extensibility, and Section VI concludes the work.

II. Architectural Options for Offline Processing

As an overview of the paper, we evaluated three different architectural paradigms, in order to address the challenge of processing large-scale offline data against complex service applications. Which are widely adopted and used across different domains [3].

A. Option I: Spark Native Application (Compute-to-Data Approach)

This approach involves refactoring the core logic of the service application (e.g., a Spring Boot service application) into a computing intensive Spark-native application. By refactoring the code and distributing Spark libraries only across the Hadoop cluster, we are capable of moving the computation directly to where the data resides in the Offline Data Warehouse (DW).

Mechanisms & Performance Metrics: Our internal benchmarking of this approach demonstrated exceptional raw performance for specific use cases.

- **High-Velocity Evaluation:** We observed a benchmark of evaluating an atomic web service Application Programming Interface (API) against **22 million historical records in** < 10 **seconds**.
- **Data Locality:** By executing service API libraries directly on top of Spark runtime, we eliminated the network I/O overhead typically associated with moving data to a service.
- **Native Resilience:** The architecture leverages the native failover and retry mechanisms of the Spark/MapReduce framework, ensuring robust handling of executor failures.

The "Dual Codebase" Constraint: Despite the performance wins for simpler web APIs, this approach was deemed unviable for the full-fledged complex service application due to **Architectural Divergence**.

- **Dependency Stripping:** Complex service application often rely on heavy I/O operations (database (DB) lookups, API integrations) which are anti-patterns as Spark-native application. Engineers require efforts to strip out these dependencies or mock them extensively, effectively rewriting the service application with a "dual" codebase.
- **High Maintenance:** The dual codebases from offline & real-time, results in high maintenance effort and long-term code sustainability issues.
- **Logic Parity:** Over time, as new features are added to the real-time application, the offline codebase inevitably falls behind. Guaranteeing 100% logical parity becomes an operational impossibility without a continuous, expensive synchronization effort between the two codebases.

B. Option II: Micro-Batch & Spark JDBC

This method utilizes a Spark Java Database Connectivity (JDBC) session [4] to read data in Hadoop partitions and execute synchronous calls to the service application via HTTP or gRPC connections.

- **Mechanism:** Spark executors iterate through Hadoop partitions, move to batch application for further data processing

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

and trigger synchronous requests to the service APIs for each record.

- **Pros:** Guarantees high logical parity as it uses the same service endpoints as production real-time.
- **Cons: Throughput Bottlenecks & Reliability:** We encountered throughput bottlenecks from both Spark JDBC sessions and synchronous API calls, which can overwhelm service applications and lead to timeouts. Moreover, micro-batches become a **Single Point of Failure** because data fetching and dispatch run in a single component.

C. Option III: Kappa Architecture

This approach treats the offline processing problem as a streaming problem [5][6]. The architecture decouples data movement from execution using asynchronous event queues.

- **Key components of Kappa Design:**
  1) **Ingress Kafka Queue:** A Spark-native application reads the offline DW data, and streams it into an **Ingress Kafka Queue** within Spark Map-Reduce runtime;
  2) **Process by Async Consumer** The heavy-weight service application is packaged as a scalable **Consumer Application**, which is capable of processing events in an asynchronous manner;
  3) **Egress Kafka Queue:** Results are published to an Egress Kafka Queue and persisted back to DW storage using lightweight processors such as Apache Flink.

Pros:

- **Total Logical Parity:** Because the consumer is a consumer wrapper *exact* of service executable libraries, there is zero risk of "logic drift." The same code logic will be unified between real-time & offline environment.
- **Elastic Scalability:** The execution tier is decoupled from the data storage. We can auto-scale the consumer fleet (e.g., via Kubernetes) from zero to hundreds of pods to match the simulation workload, optimizing compute costs.
- **Dependency Isolation:** Complex site dependencies and local caching mechanisms remain intact within the application container, removing the need to mock external services.

Cons:

- **Operational Complexity:** Asynchronous architectures are inherently more difficult to debug than synchronous batch jobs. Triaging issues such as "data loss" requires sophisticated offset tracking across Ingress and Egress queues.
- **Serialization Overhead:** There is a computational cost to transforming offline data (e.g., Spark data frame) into stream events (Avro/Protobuf) at the Ingress layer. Strict schema management is required to ensure the DataFrame matches the event consumer data schema.

D. Comparison Summary

We selected the Kappa Architecture based on its unique ability to balance throughput with strict logical parity, as summarized in Table I.

III. System Architecture

The system is composed of five primary components designed to decouple data movement from execution logic.

TABLE I. Comparison of Architecture Options

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**

Figure 1 illustrates the end-to-end data flow across these components.

- **Workflow Orchestration:** The high-level workflow orchestrator (e.g., Apache Airflow) managing the end-to-end (E2E) lifecycle. It triggers the data pipelines and tracks workflow run status.
- **Data Movement Pipeline:** A distributed Spark process responsible for fetching Point-in-Time (PiT) snapshots from the Offline DW and producing them as events into the **Ingress Kafka Queue**.
- **Service as Consumer:** The production service application packaged as a Kafka consumer. It processes each transaction using production-grade logic and publishes results to the **Egress Kafka Queue**.
- **Data Persistence:** A lightweight stream processor (e.g., Flink consumer [7]) that consumes egress events and sinks them into the Hadoop Distributed File System (HDFS) for long-term storage.
- **Analytics Layer:** SQL-based interface (Hive/Spark) allowing users to query the landed HDFS tables to calculate metrics or perform varied offline analysis.

A number of other challenges we addressed by adopting Kappa Architecture:

- **Backpressure Handling:** The Spark-based ingress pipeline is capable of producing events at a rate significantly higher than the service application can process. Without intervention, this leads to consumer lag accumulation and potential resource exhaustion. To mitigate this, we tuned the Kafka consumer configuration to strictly limit pre-fetching. Specifically, we reduced `max.poll.records` to align with the service's p99 latency, ensuring that the consumer never fetches more records than it can process within a session timeout window. Furthermore, we implemented an application-level rate limiter that dynamically pauses consumption if local thread pools become saturated [8].
- **Schema Transformation:** Our Ingress Pipeline performs a mandatory schema validation. Before publishing, the Spark DataFrame—often loosely typed—is mapped to a rigorous Avro schema governed by a central Schema Registry. This step handles type coercion (e.g., casting Hive timestamps to Avro longs) and null-safety checks. By enforcing strict Avro serialization at the ingress, we guarantee that the complex service consumer is protected from malformed data that could cause exceptions during execution [9].

These components together form a unified platform that cleanly separates data movement from execution logic, providing the foundation upon which we evaluate performance

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

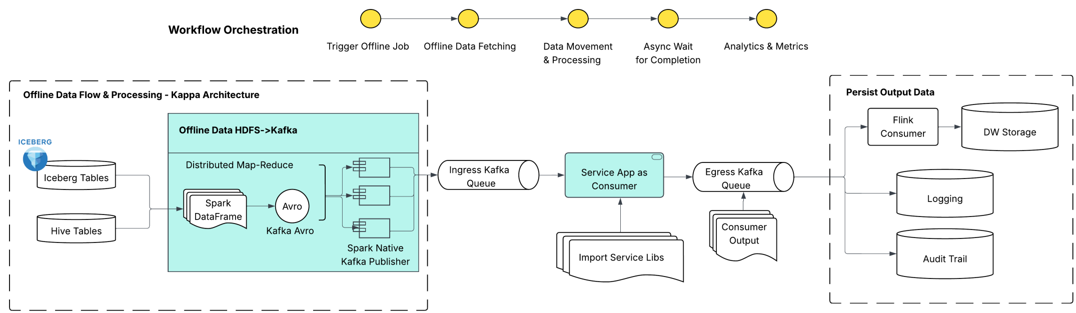

Figure 1. End-to-End Kappa Architecture Data Flow

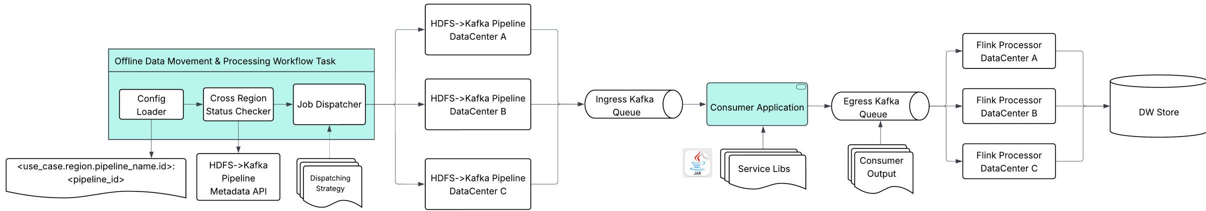

Figure 2. Cross-Region Intelligent Job Dispatching

characteristics and cross-region extensions in the following sections.

IV. PERFORMANCE BENCHMARKING

For the above end-to-end (E2E) Kappa Architecture, we have done performance benchmarking with varied size of historical data as input. We observed high throughput & low E2E latency with horizontal scaling of Kafka Queue & machines allocated to consumer pool.

A. Experimental Setup

To evaluate the efficacy and scalability of the proposed Kappa Architecture, we conducted a series of performance benchmarks in a controlled production-like environment. The experimental cluster was configured to mirror the constraints of a high-throughput, multi-tenant system.

Infrastructure Configuration: The messaging backbone was hosted on a dedicated Kafka cluster. To ensure high parallelism, both the Ingress and Egress Kafka topics were configured with **100 partitions**. This partitioning strategy was chosen to maximize consumer concurrency.

Compute Resources: The consumer application fleet was deployed on a Kubernetes cluster with resource quotas strictly enforced to simulate real-world constraints:

- **Baseline Capacity:** The fleet was initialized with a minimum of 24 pods per data center.
- **Elastic Scaling:** Horizontal Pod Autoscaler (HPA) was configured to scale the fleet up to a maximum of 48 pods per region based on CPU utilization and consumer lag metrics.
- **Pod specs:** Each consumer pod was provisioned with **2 vCPU** and **8 GB of memory**. This memory profile was specifically chosen to accommodate the local caching and high computing intensity of the heavy-weight service application.

B. Benchmarking Metrics

- **Scalability:** Capable of processing > 10 million offline data records in a single job run.
- **E2E Latency:** Achieved a processing Service Level Agreement (SLA) of < 15 minutes for the core execution phase.

TABLE II. PERFORMANCE BENCHMARKING RESULTS

Data Volume	Job Duration	Throughput
1,000,000 (1M)	7 min 03 sec	2400 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 msg/sec

Analysis of Results: The benchmarking metrics (Table II) demonstrate a non-linear relationship between data volume and job processing time, indicating high elasticity. This efficiency is driven by three architectural factors. First, the **Spark-native Kafka Publisher** runs within the Map-Reduce framework, allowing publishing throughput to scale horizontally with the number of Spark executors. Second, **Consumer Auto-Scaling** leverages Kubernetes HPA to automatically provision additional consumer pods, effectively increasing the processing rate from 2,400 to 12,000 msgs/sec. Finally, **Elastic Capacity** is managed via a strict Time-To-Live (TTL) policy on ingress topics, ensuring minimal disk footprint even with high-velocity streams.

C. Overall System Availability

- **Job Availability:** $> 99\%$ success rate.
- **Data Integrity:** End-to-end data loss was kept below 1%, primarily due to transient connection timeouts when publishing events from the ingress pipeline to the Kafka queues under large-scale traffic volume. Once events are

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-397-2

durably written to Kafka, the streaming pipeline operates with *at-least-once* delivery guarantees and did not introduce additional loss downstream.

D. Resource Efficiency

Beyond raw speed, the Kappa architecture demonstrated significant efficiency in resource utilization. By utilizing a blocking queue within the consumer application to buffer Kafka messages, we achieved a consistent CPU saturation of $\approx 50\%$ on consumer pods. And by leveraging Kubernetes Auto Scaling capability, this allows us to reduce the total provisioned cores by 50% during site idle time, while maintaining the same throughput and job SLA during runtime execution.

E. Cost vs. Convenience Trade-off

While the Kappa Architecture provides strong guarantees on logical parity and engineering velocity, it incurs additional infrastructure cost compared to a pure Spark-native implementation. Running a fleet of heavy-weight Spring Boot service containers to process batch data typically consumes more CPU and memory per record than executing equivalent logic inside a Spark job. In return, this architecture avoids maintaining a dual codebase, preserves complex dependencies and caching behavior, and enables teams to reuse production-grade logic.

V. Enhanced Feature: Cross-Region Extensibility

A key advantage of converting offline data into an event stream is the inherent decoupling of storage (Data Warehouse) from compute (Service Application). This decoupling allows the architecture to be extended easily to support **Cross-Data Center (Multi-Region) Orchestration** [10].

Figure 2 illustrates how the intelligent job dispatcher coordinates jobs across geographically distributed data centers.

By abstracting the physical location of the execution layer, the system achieves three critical operational enhancements:

A. Platform Availability & Resilience

In a traditional batch model, job failure often requires a full restart in the same cluster. The Kappa model enables an **Active-Active** availability posture. If a specific data center experiences an outage or transient infrastructure instability, the orchestration layer can instantly reroute pending jobs to an alternative region. Since consumers in Region B operate on the same logic as Region A, the failover is transparent to the end-user.

B. Distributed Resource Utilization

To prevent resource fragmentation—where one region is overloaded while others sit idle—we implemented an **Intelligent Dispatcher**. This component queries the global infrastructure state before job submission, calculating the available "Headroom" for each region: $Headroom_r = Cap_{max} - (Jobs_{active} + Jobs_{queued})$. Jobs are dynamically routed to the region with the highest $Headroom_r$, preventing "bulk submission" bottlenecks and ensuring uniform cluster utilization.

C. Improving Job E2E Latency via Partitioning

For processing jobs requiring massive historical data (e.g., 30 days of historical traffic), executing sequentially in a single region is inefficient. The streaming nature of Kappa architecture allows us to implement a **Split-and-Conquer** strategy:

1) **Partitioning:** The master job is logically split into N child jobs (e.g., 3 jobs of 10 days each).
2) **Parallel Execution:** These child jobs are dispatched *simultaneously* to different data centers (e.g., Job A $\rightarrow$ Region 1, Job B $\rightarrow$ Region 2).
3) **Throughput Multiplication:** By dispatching traffic into ingress/egress Kafka queues allocated on multiple regions, the End-to-End (E2E) SLA is reduced linearly by the number of regions.

VI. Conclusion and Future Work

The Kappa Architecture offers a robust solution for bridging the gap between large scale offline data and complex & heavy-weight service application. By decoupling data movement from execution, the system can achieve high-throughput processing and guaranteed logical parity without incurring the high maintenance costs between offline and real-time environment. In practice, we found that strict schema validation and back pressure aware consumers are essential to maintaining stable high-throughput processing at scale. As future work, we plan to strengthen observability and cost-efficiency and to extend the cross-region orchestration layer to support a broader set of offline simulation workloads.

References

[1] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc., 2017.

[2] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", in *OSDI*, vol. 51, 2004, pp. 137–150.

[3] M. Kiran, P. Murphy, I. Monga, C. Jonnalagadda, and B. Sriprasad, "Lambda architecture for cost-effective batch and speed layer operations", in *2015 IEEE International Conference on Big Data (Big Data)*, IEEE, 2015, pp. 2785–2792.

[4] M. Armbrust et al., "Spark sql: Relational data processing in spark", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1383–1394.

[5] J. Kreps, "Questioning the lambda architecture", *O'Reilly Radar*, 2014.

[6] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing", in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[7] P. Carbone et al., "Apache flink: Stream and batch processing in a single engine", *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[8] M. Welsh, D. Culler, and E. Brewer, "Seda: An architecture for well-conditioned, scalable internet services", in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, 2001, pp. 230–243.

[9] Reactive Streams, "Reactive streams specification", 2015, Accessed: 2026-04-09. [Online]. Available: http://www.reactive-streams.org/

[10] L. Wang et al., "Streams: High performance and reliable geo-distributed stream processing", in *USENIX Annual Technical Conference*, 2021.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2026. ISBN: 978-1-68558-397-2