SYSTEMS ENGINEERING PROJECT REPORT

Globally Distributed Event Streaming Platform

Commit Log • Exactly-Once • Tiered Storage • Stream Processing

Youdahe Asfaw

Computer Science • Gustavus Adolphus College
Distributed Systems • Infrastructure • Reliability Engineering

youdaheasfaw@gmail.com • linkedin.com/in/youdaheasfaw • youdahe.dev

Abstract

This report details the architecture of a globally distributed event streaming platform designed for multi-datacenter operation from inception. The system implements a partitioned, replicated commit log with configurable durability, idempotent producers with exactly-once delivery semantics, a custom binary network protocol optimized for high-throughput batched communication, tiered storage that automatically migrates data between memory, SSD, and object storage, cross-region asynchronous replication, and a built-in stream processing engine with stateful transformations and checkpointing.

Table of Contents

1. System Architecture Overview

The event streaming platform is architected as a distributed commit log designed for multi-datacenter deployment from day one. Unlike retrofitted replication solutions, every design decision—from the wire protocol to the storage engine—accounts for geographic distribution, variable latency, and independent failure domains. The system is organized into five layers: producers that publish events, a broker cluster that stores and replicates partitioned logs, a tiered storage engine that manages the data lifecycle, consumers that process events with configurable delivery guarantees, and a cross-region replication layer that mirrors data between geographically distributed clusters.

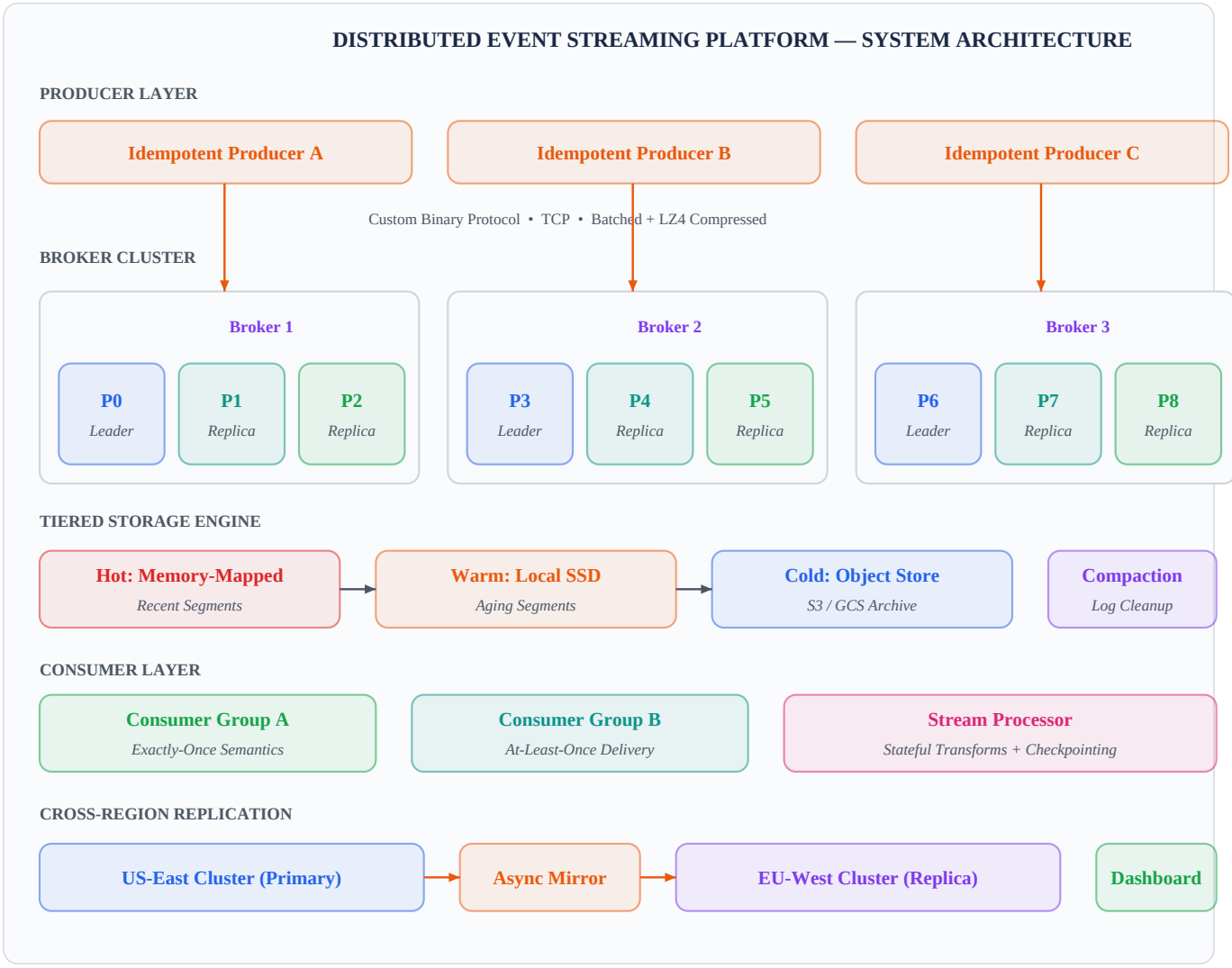

Figure 1 — Full platform architecture from producers through cross-region replication.

2. Commit Log & Partitioning

Each topic is divided into partitions, where each partition is an independent, ordered, append-only log. Producers assign messages to partitions by hashing the message key, ensuring all messages with the same key

are routed to the same partition and thus maintain strict ordering. Topics without explicit keys use round-robin assignment for load distribution.

Each partition's log is divided into segments—fixed-size files that are sequentially written and become immutable once full. Each segment maintains a sparse index mapping offsets to file positions, enabling O(log n) lookups for any offset within the segment. Segments are timestamped at both boundaries, allowing efficient time-based seeks (e.g., "give me all events after 3:00 PM"). Active segments accept appends from the partition leader; closed segments are candidates for compaction and tiered storage migration.

3. Replication & Fault Tolerance

Each partition is replicated across a configurable number of brokers (typically 3). One replica is designated the leader and handles all reads and writes; the others are followers that replicate the leader's log. The system maintains an In-Sync Replica (ISR) set—the subset of followers whose logs are within a configurable lag threshold of the leader. A message is considered committed only when all ISR members have acknowledged it.

When a leader fails, a new leader is elected from the ISR set, guaranteeing no committed messages are lost. If the ISR shrinks to a single member (the leader itself), the system can be configured to either block writes until replicas catch up (prioritizing durability) or continue accepting writes with reduced redundancy (prioritizing availability). This tunable tradeoff allows operators to match the system's behavior to their specific durability and availability requirements.

4. Exactly-Once Semantics

4.1 Idempotent Producers

Each producer is assigned a unique producer ID upon initialization. For every partition it writes to, the producer maintains a monotonically increasing sequence number. The broker tracks the latest sequence number per producer-partition pair. If a producer retries a write (due to a network timeout, for example), the broker detects the duplicate sequence number and acknowledges it without re-appending, preventing duplicate messages.

4.2 Transactional Writes

Transactional producers can atomically write to multiple partitions. The transaction coordinator assigns a transaction ID and tracks all partitions touched by the transaction. On commit, the coordinator writes commit markers to each partition; on abort, it writes abort markers. Consumers configured for read-committed isolation only see messages belonging to committed transactions—uncommitted or aborted messages are filtered out during consumption.

4.3 End-to-End Exactly-Once

The consume-transform-produce pattern achieves end-to-end exactly-once by atomically committing consumer offsets and producer output within a single transaction. If the consumer crashes after processing but before committing, the transaction is aborted and processing restarts from the last committed offset. This

guarantees that each input message is processed exactly once, with its output appearing exactly once in the target partition.

5. Custom Binary Protocol

A custom binary protocol over TCP replaces text-based APIs to maximize throughput. Messages are serialized using a compact binary encoding with fixed-width headers (request type, correlation ID, payload length) followed by variable-length payloads. Producers batch multiple messages into a single request, and the entire batch is compressed using LZ4 (for low-latency) or Zstandard (for high-ratio) before transmission.

The protocol supports **request pipelining**—multiple requests can be in-flight simultaneously on a single connection without waiting for responses, dramatically improving throughput on high-latency links. Connection multiplexing allows a single TCP connection to carry traffic for multiple partitions, reducing connection overhead. Built-in flow control uses a credit-based backpressure mechanism: the broker grants the producer a window of outstanding bytes, and the producer pauses when the window is exhausted, preventing broker memory exhaustion under load spikes.

6. Tiered Storage Engine

The storage engine manages the complete data lifecycle across three tiers, balancing access latency against storage cost. The **hot tier** consists of memory-mapped files for the most recent segments, providing sub-millisecond read latency for real-time consumers. As segments age beyond a configurable time or size threshold, they are migrated to the **warm tier**—local SSD storage with microsecond access latency, suitable for consumers with moderate lag.

Eventually, cold segments are offloaded to the **cold tier**—object storage (S3, GCS, or Azure Blob) that provides virtually unlimited capacity at minimal cost. Metadata indexes (offset-to-segment mappings and timestamp boundaries) are retained locally to enable fast seeking into cold data. A local LRU cache stores frequently accessed cold segments, optimizing for access patterns like replay or reprocessing.

7. Stream Processing Engine

The built-in stream processing layer enables stateful transformations over event streams without requiring an external framework. Supported operations include filtering, mapping, flat-mapping, windowed aggregations (tumbling windows, sliding windows, session windows with configurable gap), and stream-stream joins with configurable join windows.

State is maintained in local RocksDB instances, one per processing task, providing fast key-value access for aggregation counters, join buffers, and window contents. State is periodically checkpointed to durable storage (the commit log itself), enabling recovery after failures: on restart, the processor restores its last checkpoint and replays events from the corresponding offset.

The engine supports **event-time processing** using watermarks—monotonically increasing timestamps that track how far the stream has progressed in event time. Late events (those arriving after the watermark has passed their window) are handled via configurable late-arrival policies: drop, emit a correction to a side output, or recompute and emit an updated result. Exactly-once processing guarantees are maintained by tying state checkpoints and consumer offset commits into a single atomic transaction.

8. Cross-Region Replication

Asynchronous mirroring replicates topics between geographically distributed clusters. A mirror agent consumes from the source cluster's commit log and reproduces records into the target cluster, preserving partition assignments and key ordering. The agent tracks its progress via committed offsets, enabling crash recovery without data loss or duplication.

Each region operates independently—producers write to their local cluster, and consumers read from it. For use cases requiring global consumption (e.g., a centralized analytics pipeline), consumers can subscribe to the aggregated view that merges locally produced and mirrored records. Conflict-aware consumer groups use vector clocks to detect and handle ordering conflicts when the same key is written concurrently in multiple regions. Latency-aware routing directs producers and consumers to their nearest cluster, minimizing round-trip time for the common case.

9. Observability Dashboard

A real-time Grafana-based dashboard provides comprehensive visibility into platform health. Key metrics include per-partition throughput (messages/sec and bytes/sec), end-to-end latency percentiles from producer send to consumer delivery, consumer group lag (the gap between the latest committed offset and the partition's high watermark), ISR shrinkage events, and cross-region replication lag. Alerts are configured for conditions such as sustained consumer lag growth (indicating a slow consumer), ISR shrinkage below minimum (indicating replication health risk), and tiered storage migration failures.

10. Technology Stack Summary

Component	Technology / Approach
Commit Log	Custom append-only log in Rust with segment indexing and compaction
Replication	Leader-follower with ISR tracking and configurable durability
Protocol	Custom binary over TCP with batching, LZ4/Zstd compression, pipelining
Tiered Storage	Memory-mapped files, local SSD, S3/GCS with LRU cache
Exactly-Once	Producer ID + sequence dedup, transactional writes, atomic offsets
Stream Processing	Stateful transforms, RocksDB state stores, checkpoint/restore

Windowing	Tumbling, sliding, session windows with event-time watermarks
Cross-Region	Async mirror agent, vector clocks, conflict-aware consumers
Observability	Prometheus metrics, Grafana dashboards, alerting on lag and ISR
Languages	Rust (broker, protocol, storage), Go (mirror agent, tooling)