Ursa: A Lakehouse-Native Data Streaming Engine for Kafka

Matteo Merli, Sijie Guo, Penghui Li, Hang Chen, Neng Lu
StreamNative, Inc.
ursa-paper@streamnative.io

ABSTRACT

Data lakehouse architectures unify the cost-efficiency of data lakes with the transactional guarantees of data warehouses. Yet, real-time ingestion often depends on external streaming systems such as Apache Kafka, along with bespoke connectors that read from Kafka and write into the lakehouse—leading to increased complexity and high operational costs. In particular, traditional leader-based data streaming platforms are designed for sub-100 ms low-latency workloads; however, when used for data-intensive ingestion in a cloud environment, cross availability-zone (AZ) disk-based replication significantly raises total infrastructure costs due to excessive network traffic and overprovisioned disk storage. This paper introduces Ursa, a leaderless, cloud-native, and Kafka-compatible streaming engine that writes data directly to open lakehouse tables on object storage. By eliminating leader-based replication, disk-based broker storage, and external connectors, Ursa markedly reduces infrastructure costs while preserving high throughput, exactly-once semantics, and near-real-time streaming capabilities. Experimental results show that Ursa matches the performance of traditional Kafka clusters at a fraction of the cost, offering up to a 10x reduction in infrastructure expenses.

PVLDB Reference Format:
Matteo Merli, Sijie Guo, Penghui Li, Hang Chen, Neng Lu. Ursa: A Lakehouse-Native Data Streaming Engine for Kafka. PVLDB, 18(12): 5184 - 5196, 2025.
doi:10.14778/3750601.3750636

1 INTRODUCTION

Enterprise analytics increasingly relies on lakehouse architectures [25], which unify the transactional guarantees, schema management, and performance optimizations of data warehouses with the elasticity and low-cost storage of data lakes. Concurrently, Apache Kafka [15] has become a de facto standard for real-time data streaming, offering low-latency ingestion from various sources. Although both paradigms have seen wide adoption, they are commonly operated as separate infrastructures: data flows into Kafka first, then is periodically moved to lakehouse tables. As organizations emphasize cloud-native deployments and cost consciousness, integrating Kafka with lakehouse environments can be challenging.

One complexity arises from leader-based streaming systems (e.g., Kafka, Pulsar [17], Redpanda [3]), which rely on disk-based replication for sub-100ms latency. However, lakehouse ingestion often only requires sub-second latencies (200–500ms). In these scenarios, cross-AZ communication and replication significantly increase costs, while overprovisioned brokers and complex maintenance tasks (leader elections, rebalancing) add unnecessary overhead.

A second challenge involves integrating Kafka with lakehouse. Moving data requires connectors or ETL pipelines, forcing organizations to manage two distinct systems with fragmented tooling. Data is copied multiple times, raising costs, while real-time data remains locked in Kafka until scheduled ingestion, delaying analytics availability.

This paper introduces Ursa, a leaderless, lakehouse-native, and Kafka-compatible streaming engine that writes data directly into open table formats in object storage. By decoupling compute from storage, Ursa tackles real-time data streaming challenges in two ways: (1) removing leader-based replication and disk-based broker storage cuts costs, especially in multi-AZ deployments, and (2) unifying stream and table semantics within a single storage layer eliminates connectors and staging areas. This direct-to-lakehouse design streamlines operations, shortens time-to-insight, and enables a universal data platform for analytics, event-driven applications, and AI—all without the overhead of multiple infrastructures.

2 BACKGROUND AND MOTIVATION

2.1 Lakehouse Architectures and Real-Time Data Pipelines

Modern data lakehouse architectures, built on open table formats (for example, Apache Iceberg [14] or Delta Lake [1]), unite the low cost of cloud object storage with transactional guarantees, schema evolution, and performance optimizations such as partition pruning and file compaction. As a result, lakehouses increasingly blur the line between batch-oriented data lakes and high-performance data warehouses.

Yet, real-time ingestion often relies on external streaming engines that temporarily store records on disk before converting them into columnar formats suitable for lakehouse queries. This additional step introduces latency and operational overhead. Apache Kafka has become an industry standard for low-latency streaming pipelines, offering publish-subscribe messaging, horizontally scalable partitions, and robust replication. However, integrating Kafka with a lakehouse typically involves a standalone Kafka cluster along with supplemental connectors (e.g., Kafka Connect [7]) for offloading and transforming disk-based topic partitions into columnar files. As retention periods grow from days to weeks or months, version management, error handling, schema validation, and resource consumption all become more complex, compounding latency and operational costs.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750636

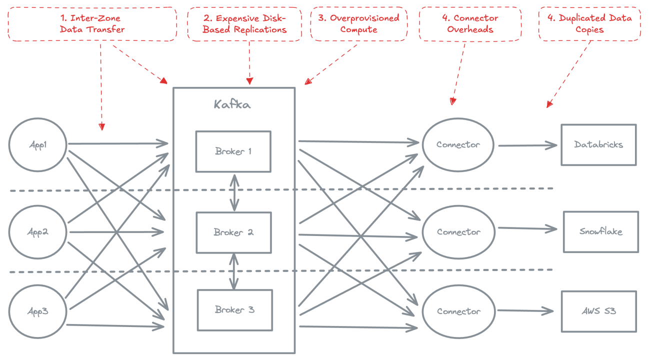

Figure 1: Cost drivers of data streaming

2.2 Data Streaming Cost Drivers

Although pairing Kafka with a lakehouse is widely adopted, organizations often encounter substantial expenses when examining each cost factor individually. Figure 1 summarizes the main cost drivers:

(1) **Inter-Zone Data Transfer.** Kafka's leader-based replication designates a single broker as leader for each partition, with additional brokers acting as followers. In multi AZ deployments, producers located in one zone may need to communicate with remote leaders, while cross-AZ replication further inflates network costs. Although multi-AZ redundancy strengthens fault tolerance, it becomes expensive at scale when a significant portion of traffic and replicas traverse zone boundaries.

(2) **Disk-Based Replication.** Kafka's default operational model relies on broker-level disk replication, often necessitating premium hardware. Retaining multiple copies of data on disk for extended durations inflates storage costs. Even when tiered storage is employed, core replication overhead still drives up both capacity and I/O demands, especially during peak loads or rebalancing events.

(3) **Overprovisioned Compute.** Because Kafka couples compute and storage resources, adding disk capacity requires provisioning additional brokers. This approach can lead to overprovisioning [21], with clusters incurring full operational costs during off-peak hours. Periodic partition rebalancing to spread data across new brokers complicates operations and may require planned downtime.

(4) **Connector Overheads.** Transferring data from Kafka into a lakehouse frequently involves specialized connectors (e.g., Kafka Connect), which consume CPU, memory, and I/O resources. They also introduce additional failure points and operational overheads related to schema validation, transformations, and version management. Large organizations may deploy multiple connectors to feed different analytics platforms, magnifying complexity.

(5) **Duplicate Data Copies.** Multi-step pipelines and numerous target systems often create multiple data copies in different formats and locations (e.g., raw events on Kafka brokers, transformed files in the lakehouse, intermediate landing zones). Storing these duplicates inflates costs, complicates governance, and risks inconsistent versions or incomplete lineage, especially when integrating with additional services such as Databricks or Snowflake.

These factors combine to dramatically increase infrastructure costs for lakehouse ingestion. Ursa addresses these drivers through a leaderless, lakehouse-native architecture.

3 OVERVIEW

3.1 Design Goals

Ursa's design is driven by the need to unify real-time data streaming with lakehouse-centric analytics while controlling infrastructure expenses. The key requirements include:

Cost Efficiency. Many organizations incur significant costs in multi AZ deployments due to cross-zone data transfers and disk-based storage. The system should eliminate the major cost drivers discussed in Section 2.2.

Lakehouse Native. The system must support native writes to open table formats, removing the need for bespoke connectors or staging areas and enabling a single data flow from ingestion to analysis.

Scalability and Elasticity. As data volumes and workloads change, the infrastructure should scale transparently without rebalancing partitions or migrating data.

Stream–Table Duality. The system should merge real-time event processing and tabular analytics on a single set of files, thereby minimizing data duplication and operational complexity.

3.2 New CAP Theorem

Ursa's development revealed that no single configuration can optimize all requirements in cloud environments. We propose a "New CAP Theorem" for data streaming, identifying trade-offs among Cost, Availability, and Performance Tolerance:

- **Cost**: Infrastructure expenses per MB throughput—systems minimizing cross-zone replication can dramatically reduce networking costs.
- **Availability**: Resilience to AZ failures—systems requiring full multi-AZ availability must implement costly cross-zone replication.
- **Performance Tolerance**: Acceptable latency bounds—applications tolerating 200-500ms latency can leverage object storage for cost savings.

These properties cannot be maximized simultaneously. Sub-50ms latency with multi-AZ availability requires expensive replication, while accepting higher latencies enables dramatic savings. Ursa's pluggable architecture allows workloads to tune these trade-offs.

3.3 Architecture

Ursa is designed to be leaderless, cloud-native, and lakehouse-native to satisfy the key requirements discussed above. Figure 2 provides a conceptual view of Ursa's high-level architecture, which comprises three core layers:

Metadata Service: A centralized metadata service, currently implemented via StreamNative Oxia [24], is responsible for offset assignment, partition metadata, and transaction states. By delegating

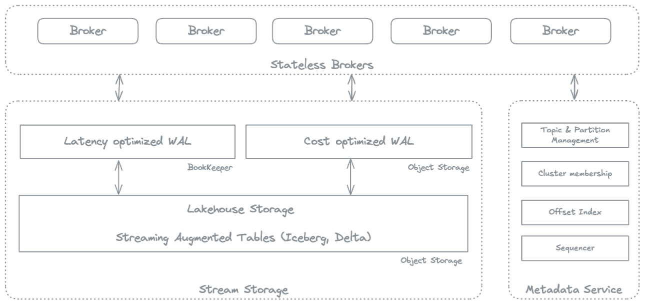

Figure 2: Ursa High-Level Architecture

these tasks to a separate service, Ursa eliminates the need for brokers to perform leader-based coordination. This approach ensures consistent offset ordering for each topic-partition and simplifies failover scenarios.

Stream Storage: Ursa separates storage from compute by persisting incoming data first to an external write-ahead log (WAL)—which can be remote disk-based or object-based—and later converting it into columnar files stored in open table formats for long-term retention. By avoiding local disk-based logs on brokers, this approach reduces operational overhead and simplifies data replication.

Stateless Brokers: Ursa brokers accept client connections, handle the Kafka protocol, and interface with the Stream Storage layer to write or read messages. They also communicate with the metadata service to commit updates and retrieve offset indexes. Because brokers do not maintain partition-specific state or disk-based logs, they can be scaled up or down without requiring partition rebalancing. Zone-aware routing ensures that producers and consumers within the same availability zone connect to local brokers, significantly reducing cross-AZ traffic.

Additionally, a **compaction service** periodically converts row-based WAL objects into columnar Parquet [16] files, which are then registered in open lakehouse formats (Apache Iceberg, Delta Lake, etc.). This conversion makes newly ingested data immediately available for both batch queries and continuous streaming reads.

3.4 Metadata Service

Ursa employs a centralized metadata service - currently implemented with StreamNative Oxia - to coordinate critical operations at scale. Oxia offers essential primitives for building distributed sequencers to generate offsets, stores partition- and topic-level metadata, and maintains membership information for all brokers in the cluster. By centralizing these functions, Ursa offloads tasks that would otherwise reside in brokers, thus, simplifying Ursa broker design and improving fault tolerance.

This approach underpins Ursa's leaderless model, where the metadata service functions as the single authoritative source of offsets and transaction state. Whenever a broker processes a 'Produce' or 'Fetch' request, it consults the metadata service to retrieve or update offset indexes and to guarantee correct ordering of incoming messages.

Oxia was chosen for its ability to scale seamlessly as new partitions, brokers, or topics are introduced. It manages concurrency through lightweight, optimistic protocols that preserve global ordering across metadata commits, without incurring the complexities often associated with leader-based broker models. While Oxia is the default, Ursa's pluggable design supports alternative implementations like ETCD or Spanner for different deployment scenarios. Further details about metadata structures and offset indexes appear in Section 4.

3.5 Stream Data Storage

Ursa's stream data storage layer unifies real-time data streaming and batch analytics by implementing both stream and table semantics within a single dataset. It achieves this duality by combining a write-ahead log (WAL) for newly ingested messages—stored in row-based formats—with an open lakehouse format (e.g., Apache Iceberg or Delta Lake) for long-term columnar storage in cloud object stores such as Amazon S3. A scalable offset index, maintained within Ursa's metadata service, tracks location whether it resides in the WAL or in the lakehouse tables.

3.5.1 Write-Ahead Log Storage. When brokers receive new data, they buffer incoming messages until reaching a configurable threshold, triggered by time or data size. At that point, the accumulated records are flushed into row-based WAL objects, and the corresponding metadata references are updated in the metadata service to allow efficient lookups. This design enables Ursa brokers to retrieve specific data blocks without scanning the entire WAL object, thereby supporting low-latency reads for streaming consumers. Acknowledgments to producers are deferred until the WAL object and its metadata references are durably recorded, ensuring that messages cannot be lost in the event of a broker or network disruption.

To address diverse workload requirements, Ursa offers a pluggable write-ahead log interface. For latency-sensitive applications, Ursa can employ Apache BookKeeper [12] with replicated disk storage to achieve p99 latencies from the single-digit to the low-hundreds of milliseconds. For cost-sensitive scenarios, Ursa can rely on object storage for the WAL, which supports sub-second latency while significantly reducing storage costs. This flexibility allows organizations to align data streaming configurations with both budgetary and performance goals.

3.5.2 Lakehouse Storage. Once data has been durably stored in the WAL, a background compaction service converts row-based WAL objects into columnar Parquet files that are partitioned by topic. As each Parquet file is produced, the compaction service updates the offset index in Ursa's metadata service, mapping the relevant offset range to the newly created file. This mechanism enables brokers to serve read requests directly from Parquet files, thereby enhancing retrieval efficiency. Following compaction, the original WAL objects can be safely removed to reclaim storage.

In addition to updating the metadata service, the compaction service also commits the compacted files to a lakehouse table, yielding a unified view of the dataset without creating extra copies. This design immediately exposes data to a wide range of processing engines (such as Spark [18], Trino [20], or Flink [13]), eliminating the need for additional connectors to migrate streaming data into a separate lakehouse environment. By reading and updating the same objects, Ursa streamlines data lifecycle management and reduces operational overhead.

Ursa's native mechanism for managing these lakehouse tables is referred to as a Stream-Backed Table (SBT). Under this model, the data resides in cloud object storage and is fully managed by Ursa, with the streaming index in the metadata service and the lakehouse table metadata both referencing the same underlying objects. External analytical engines can query the lakehouse tables in read-only mode.

3.5.3 Compaction Service. Beyond converting WAL objects into Parquet, Ursa's compaction service can also deliver data to external lakehouse tables (e.g., Databricks Unity Catalog [8] or Snowflake Open Catalog [22]) through a Stream-Delivered-to-Table (SDT) mode, and it supports Kafka topic compaction by retaining only the latest version of each key. Section 5 examines the internals of this service.

3.6 Stateless Brokers

Ursa's brokers are stateless and leaderless, diverging from the conventional model where each broker stores local state and often serves as a leader for specific partitions. Instead, brokers concentrate on fulfilling producer and consumer requests while delegating data persistence and offset sequencing to the metadata service and the storage layer. This division of responsibilities affords multiple benefits. Brokers can be added or removed without rebalancing partitions or migrating data, as they do not maintain local logs for each partition. Failures do not trigger leader elections or partition redistributions; any other broker can simply resume handling requests by consulting metadata and retrieving data from object storage.

By employing zone affinity, producers typically connect only to brokers in the same availability zone, thereby avoiding inter-zone hops that would otherwise escalate network expenses in conventional Kafka architectures. Furthermore, the absence of partition leadership or on-disk replication notably reduces the complexity of routine tasks such as upgrades and scaling. Section 5 elaborates on the implementation details of Ursa brokers, covering both the data flow and the features that ensure Kafka protocol compatibility.

4 URSA STREAM STORAGE

This section describes Ursa's stream storage design in detail, including its core stream format, data objects, offset index, and the distributed compaction service that converts WAL objects into Parquet files for lakehouse formats. It also explains how the same mechanisms are extended to handle external lakehouse tables and implement Kafka topic compaction.

4.1 Stream Format

Each topic-partition in Ursa is stored as a Stream—an append-only sequence of records, each assigned a monotonically increasing offset. Records within a stream cannot be modified once appended, preserving the strict order that is central to Kafka's consumption model. Throughout this paper, we use stream and topic-partition interchangeably.

Figure 3 illustrates the Ursa stream storage layout: every stream is composed of physical data objects residing in either the WAL

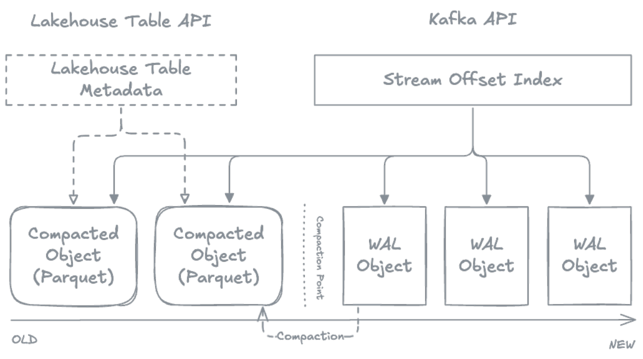

Figure 3: Ursa Stream Format

or the lakehouse storage, with all references tracked by an offset index in the metadata service. Each stream has a unique **Stream ID**, assigned at creation time by the metadata service.

Ursa distinguishes between two primary types of data objects. The first, a **WAL object (WO)**, aggregates records from multiple streams in row-based format, enabling efficient appends. The second, a **Compacted object (CO)**, is stored as a Parquet file specific to an individual stream. When a latency-optimized WAL is used, WAL objects reside in Apache BookKeeper (see Section 3.5.1); otherwise, they are stored in object storage under a designated 'storage' folder. Parquet files are placed under a 'compaction' folder, organized by topic-partition.

4.1.1 WAL Objects. WAL objects contain batched messages from multiple streams. To maximize write efficiency, Ursa sorts entries by their stream identifiers before writing them to each WAL object, minimizing the number of index segments needed for lookups. Although these objects can accumulate data from many partitions, the offset index ensures that consumers only retrieve the relevant ranges for a given topic-partition. This approach prevents the creation of excessively small files while keeping memory usage within acceptable bounds. Once the data in a WAL object is durably written to storage, Ursa commits the corresponding metadata updates atomically in the metadata service.

Each WAL object is assigned a unique name (often a GUID), and the association between any WAL object and a particular stream is determined by the corresponding offset index.

4.1.2 Compacted Objects. Compacted objects are topic-partition–specific data objects used to store data in columnar formats. They reside in the compaction folder and are organized by topic-partition. The compaction service transforms WAL objects into these columnar objects, and Ursa also provides APIs that allow writers to append Parquet files directly or update the offset index to replace WAL references with Parquet references for subsequent reads.

Parquet was chosen as the columnar format because it offers compression flexibility, supports nested data structures, and constitutes the de facto file format standard for open lakehouse systems. By adopting Parquet, Ursa can seamlessly expose these compacted objects to lakehouse tables without duplicating data and can immediately benefit from new Parquet features or optimizations, as well as integrations with other engines.

Each compacted object is assigned a unique name—often a GUID—and may be referenced by the stream's offset index or by the metadata in any lakehouse tables that share the same underlying data.

4.1.3 Stream Offset Index. Each topic-partition maintains an offset index in the metadata service to reference the entries produced into that partition. The offset index consists of a sequence of location objects, each mapping a continuous range of logical offsets to a physical data object—either a WAL object or a compacted Parquet file. This mapping uses a composite key formed from 'StreamID', 'OffsetEnd', and 'CumulativeSize':

- **StreamID** uniquely identifies a topic-partition.
- **OffsetEnd** is the exclusive upper-bound offset covered by this index entry.
- **CumulativeSize** tracks the total byte count from the start of the stream up to the data covered by this index entry.

The value associated with this composite key is an index entry containing the metadata necessary to locate the data in its physical data object:

- **Location** indicates where the data is stored.
- **FileType** specifies whether the data resides in a WAL object or a Compacted object.
- **EntryCount** indicates the number of entries indexed by this entry.
- **MessageCount** indicates the total number of messages indexed by this entry.
- **OffsetInObject** specifies the exact position within the physical location for efficient data retrieval.
- **EntryOffsets** is a local index that references the data entries captured by this index entry, reducing the total number of offset-index entries needed.

All key-value pairs in the offset index are ordered by 'OffsetEnd' and 'CumulativeSize'. This ordering enables the metadata service to quickly identify the correct data block for a given offset. For instance, when a broker needs to read from offset 'x', it locates the smallest key such that `OffsetEnd > x`, then uses the corresponding metadata to retrieve the data from the appropriate storage location.

4.1.4 Table Metadata. Because all compacted objects in Ursa are stored in Parquet, these files can be committed to a lakehouse table without duplicating or re-copying data. As an example, for Delta Lake, each newly produced Parquet file can be incorporated via Delta Lake's primitives, recording an 'Add File' entry in the Delta log. When smaller files are compacted into a larger Parquet file, the compaction service issues corresponding 'Remove File' and 'Add File' actions. Ursa applies a similar workflow for Iceberg, thereby allowing the same compacted objects to be exposed as either Iceberg or Delta tables. This design enables stream-table duality on a single set of data objects and avoids maintaining multiple data copies. The overall approach is similar to Delta UniForm [9] or Apache XTable [19], which aim at interoperability among table formats, whereas Ursa targets interoperability between streaming semantics and table semantics.

4.2 Access Protocols

Ursa's access protocols are designed to ensure linearizability when multiple clients concurrently append records to the same stream.

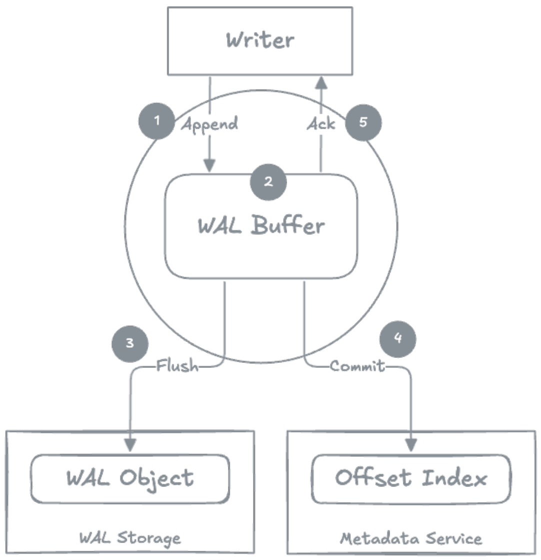

Figure 4: Append entries to a Ursa stream

Once a client's append operation returns, any subsequent reader should be able to observe the newly appended record at the correct offset.

4.2.1 Append Entries. Figure 4 illustrates how entries are appended to a Ursa stream:

(1) **Entry Buffering**. The Ursa library buffers incoming entries from different writers in memory until a size or time threshold is reached. All buffered entries are then sorted by stream id. It is shown as step 2 in Figure 4.
(2) **WAL Persistence**. The library flushes the buffer to a WAL storage layer. When using the cost-optimized WAL, entries are persisted as an object in object storage; for the latency-optimized WAL, entries are stored as a blob in BookKeeper. The entire WAL object must be durably recorded before proceeding. It is shown as step 3 in Figure 4.
(3) **Atomic Offset Index Update**. Next, the Ursa library invokes a transactional update in the metadata service to assign new offsets and add corresponding index entries. This step ensures that offsets remain strictly increasing. It is shown as step 4 in Figure 4. The pseudocode below illustrates this procedure:

```
with oxia.transaction(stream_id) as tx:
    last = tx.get_last_key(stream_id)
    new_key = (stream_id,
        last.Offset_end + entry.MsgCnt,
        last.CumulativeSize + entry.Size)
    tx.put(new_key, BatchMetadata(
        “wal_object_location”,
        batch.MsgCnt, batch.Size,
        batch.OffsetInObject))
```

(4) **Consistency**. By composing 'OffsetEnd' and 'CumulativeSize' monotonically, Ursa ensures no overlapping offset

ranges. The metadata service's concurrency control enforces linearizable writes, guaranteeing that newly committed entries are immediately visible to subsequent readers.

The write protocol requires a metadata service with either a built-in "atomic increment" primitive (as provided by Oxia) or transactional capabilities for metadata operations to ensure the atomicity of offset generation and offset-index updates.

By sequencing the steps of appending entries to the WAL and then committing the corresponding metadata, multiple writers can operate concurrently without blocking each other, thereby maximizing throughput and maintaining linearizability. If a writer fails after writing a WAL object but before committing its metadata, or if it fails to commit the metadata, the resulting unreferenced WAL object becomes orphaned and is subsequently cleaned up by the compaction service.

4.2.2 Read Entries. Reading entries from offset x in a Ursa stream involves three main steps:

1. **Index Lookup**. The broker (or Ursa reader) queries the metadata service to locate the smallest index key for which `OffsetEnd > x`. This lookup immediately identifies the object that stores the required records.
2. **Data Retrieval**. After retrieving the relevant index entry, the reader knows the precise location of the records. It then reads the data from the appropriate offset within the WAL or compacted object.
3. **Optional Format Conversion**. If the requested data resides in a WAL object, the broker can return it without conversion. If the data is stored in a Parquet file, Ursa converts it from columnar to row-based format to comply with the Kafka protocol. Although this conversion introduces additional CPU cost, Parquet's columnar layout typically yields faster disk I/O, mitigating the overhead. In the future, enhancements to the Kafka protocol (e.g., support for Arrow) could eliminate the need for such conversions entirely.

Although the read process appears straightforward, Ursa employs various optimizations to minimize load on both the metadata service and the underlying storage layer. For example, Oxia supports a notification mechanism that alerts readers in real time when offset-index updates occur—such as when WAL objects are replaced by compacted objects—avoiding repeated offset-index queries. Further read-path optimizations for reducing amplification are discussed in Section 5.3.

4.2.3 Update Offset Index. Streams in Ursa are typically immutable, so there is no need for in-place updates to existing records. However, because Ursa stores row-based WAL objects for low-latency ingestion and later compacts them into columnar objects, the offset index must occasionally be updated to replace multiple entries referencing different WAL objects with a single entry referencing a compacted Parquet file. Ursa provides a primitive for this workflow:

```
with oxia.transaction(stream_id) as tx:
    tx.remove(index_entry_1_key)
    tx.remove(index_entry_2_key)
    tx.remove(index_entry_3_key)
    tx.put( compacted_index_entry_key,
            BatchMetadata("compacted_object_location",
                batch.MsgCnt, batch.Size))
```

By removing old index entries and inserting a new one within the same transaction, Ursa ensures that no partial updates occur and that consumers see consistent state at all times.

4.2.4 Consistency Guarantees. Ursa provides linearizable writes, read-your-writes semantics, and durable commits while keeping brokers stateless.

- **Linearizable writes.** All partition metadata—offset indices, fencing states, and group state—is committed to Oxia, a shared key-value store whose shards each elect a leader and replicate every update to a quorum of followers before acknowledgement. This Raft-based replication protocol ensures all writes for a given shard are totally ordered, precluding intermediate states.
- **Read-your-writes.** Because every 'Produce' is only acknowledged after commiting to Oxia, subsequent 'Fetch' requests necessarily observe an `offset >=` the caller's last commit. Brokers stamp cache entries with the latest Oxia revision to ensure session monotonicity.
- **Durability & atomicity.** A commit spans (i) the replicated WAL object and (ii) its Oxia index entry; the index is committed only after the WAL write succeeds, guaranteeing that every acknowledged record resides on durable media.
- **Ephemeral caches.** In-memory broker buffers accelerate hot reads but are never authoritative; if lost, a broker rebuilds cache directly from the WAL and Oxia without data loss.

Note. Oxia's consensus protocol, heartbeat mechanism, and fencing semantics are described in detail in [24]; we omit them here for brevity.

4.2.5 Failure Recovery. The same design upholds consistency during both crash-stop and brown-out events.

- **Broker failover.** Heart-beat loss (≈5s) marks a broker offline; the discovery service instantly remaps its partitions to healthy peers, which resume service by reading the last committed offset from Oxia and the corresponding WAL bytes—no shard copy is required.
- **Brown-out mitigation.** A broker that remains alive yet shows sustained latency spikes or throughput collapse is flagged degraded when its publish/fetch latency exceeds $P_{99} + 3\sigma$ (standard deviation) for 30 s. The discovery service (i) evicts the node and (ii) immediately re-assigns its partitions, a zero-copy operation because brokers hold no durable state. Producers/consumers can pickup the new assignment on retries, so tail-latency inflation is capped to a single retry. If the broker's metrics return to the nominal envelope for ≈60 s, it is automatically re-admitted, preventing transient slowdowns from throttling the cluster while preserving elastic capacity.
- **Metadata resilience.** If an Oxia shard leader fails, a fully caught-up follower is promoted; quorum replication ensures every committed index entry survives.
- **Client impact.** Producers and consumers see only a short interruption—typically one retry RTT (<100ms when the

TCP connection closes) and, in the silent-failure case, at most the heartbeat interval (≈5s)—after which requests flow to the new broker; no acknowledged record is lost, no duplicate emitted, and linearizability as well as read-your-writes hold across broker crashes, brown-outs, and metadata fail-overs.

4.3 Distributed Compaction Service

4.3.1 Overview. While WAL objects enable high-throughput, low-latency appends, interleaving many streams in one file hurts catch-up reads. A broker can issue range requests to fetch the exact slice containing a given offset, yet a lagging consumer still has to issue numerous, non-contiguous range reads across many WAL objects—one for every batch that belongs to its stream. This fragmented access defeats object-store read-ahead and cache locality, so in practice the client winds up touching most of each file even if it needs only a small fraction of the bytes. When multiple consumers back-fill different partitions, the amplification compounds.

To alleviate this problem and to satisfy heterogeneous retention goals, Ursa runs a dedicated compaction service that periodically rewrites WAL objects into larger, topic-partition–specific Parquet files. After compaction a consumer can obtain its entire backlog with a handful of sequential reads, restoring I/O efficiency while live tail-reads continue to use range requests on the latest WAL segments. During the rewrite the service leverages the embedded schema ID (Avro, JSON, or Protobuf) to convert row batches into columnar Parquet and registers the output with the lakehouse catalog. The next subsections describe the compaction service, how it mitigates read amplification, and how the same mechanism supports tasks such as upserts to external lakehouse tables and Kafka topic compaction.

4.3.2 Compaction Workflow. Ursa's compaction service follows a three-stage pipeline, inspired by MapReduce [10], that ensures exactly-once transformations while preserving the strict ordering of each stream. The offset index in the metadata service remains central to this process, guaranteeing that no data is duplicated, reordered, or lost. Newly compacted Parquet files are committed atomically to an object storage system resilient to availability zone failures, ensuring continuous data accessibility.

Stage 1: Task Generation.

A leader node inspects the offset ranges for each stream and partitions them into smaller, independently processable segments. The pseudocode below demonstrates how the leader determines segment sizes dynamically, based on each topic's throughput. By checkpointing its state to Oxia every 30 seconds, the leader can tolerate partial failures and recover gracefully.

```
def generate_tasks(topic):
    offsets = Oxia.get_offset_range(topic)
    segment_size = adaptive_window(throughput(topic))
    return [
        CompactionTask(topic, start,
            min(start + segment_size, offsets.end))
        for start in range(
            offsets.start, offsets.end, segment_size)
    ]
```

Stage 2: Distributed Compaction.

Once the leader generates tasks, worker nodes retrieve the designated WAL objects, optionally filter or preprocess them, and convert the row-based data into columnar Parquet files. Ursa uses the relevant schema (e.g., Avro, JSON, or Protobuf) to perform this transformation. The workers then produce metadata describing the new Parquet files—location, message count, and offset bounds—and send this information back to the leader. Because compaction occurs outside Ursa brokers, it can scale horizontally across hundreds of thousands of topics without burdening real-time ingestion. The pseudocode below illustrates this procedure.

```
def wal_to_parquet(wal_segment):
  df = createDataFrame(wal_segment.messages, schema)
    df.write.parquet(temp_path, compression="ZSTD")
    return FileMetadata(
        temp_path, df.count(), min_max_offsets)
```

Stage 3: Commit.

The leader gathers metadata from the workers and batches the newly generated files—often 100 to 1,000 at a time—then updates the offset index in the metadata service. Concurrently, it performs an atomic commit via Iceberg or a comparable API to register these Parquet files with the underlying lakehouse table. Grouping multiple files reduces the metadata overhead in both Ursa's metadata service and the lakehouse catalog. Version rollback is available, as Oxia retains a few previous committed versions for swift recovery or partial rewinds. Once the new files are successfully committed, the system can safely reclaim older WAL objects, minimizing data duplication and read amplification for future data consumers.

4.3.3 Read Amplification Mitigation. During Stage 2, compaction worker nodes might repeatedly read WAL objects, risking excessive read amplification. To address this, workers benefit from a zonal distributed cache, discussed in Section 5.3.1, which reduces redundant loads when multiple compaction tasks require the same data. By limiting re-fetching overhead, Ursa lowers costs and latency for large-scale compaction activities.

4.3.4 Data Cleanup. The leader node maintains a holistic view of the compaction progress and can readily purge any WAL objects fully converted into Parquet. Orphaned objects—those no longer referenced by the offset index—are also removed. This ongoing cleanup conserves storage resources while preserving a clear, streamlined index structure.

4.4 External Lakehouse Tables

The existing compaction framework also supports "SDT" (Stream-Deliver-to-Table) scenarios for external lakehouse tables. In Stage 2 of the compaction process, the worker reuses the same WAL reading pipeline to avoid duplicate reads, but writes separate Parquet files optimized for the target catalog (e.g., Databricks Unity Catalog or Snowflake). While this creates two Parquet copies, one for Ursa's internal storage and one for the external table—it enables catalog-specific optimizations and flexible data sharing across platforms.

4.5 Schema Evolution

Ursa inherits column-addition, type-promotion, and field-rename rules from Iceberg/Delta, but adds a stream-friendly policy that avoids write-time compatibility checks. Each producer tags every record with a schema-ID obtained from the Kafka Schema Registry. During compaction, the worker deserialises records with their own writer schema and rotates to a new Parquet file whenever the schema-ID changes; a single compaction cycle can therefore emit several files—one per contiguous schema version—while each file remains internally self-consistent. On the lakehouse side Ursa builds the table schema by union-merging all historical versions, preserving every additive field and marking it nullable (primary-key columns stay required). Deletions are logical only: they vanish from query output but the physical column is retained for backward compatibility. Because the catalog stores that merged schema and each WAL object retains its original schema-ID, analytic engines can always rehydrate a record with the correct writer schema even as the table evolves. The result is zero write failures during rapid producer roll-outs, graceful support for mixed-version streams, and schema management that stays orthogonal to Ursa's high-throughput streaming path.

Handling malformed records. If deserialization fails—for example, because a producer sends data that does not conform to the declared schema—the compaction worker routes the offending payload to a dead-letter topic (DLT) linked to the original stream. Offsets of discarded records are still recorded in the main index, preserving gap-free ordering, while operators can inspect or reprocess the DLT asynchronously without blocking the primary ingestion path.

4.6 Kafka Topic Compaction

Ursa's compaction framework can also implement Kafka-style topic compaction with only minor adjustments. In Stage 1, the leader generates specific compaction tasks based on configured thresholds (e.g., size- or time-based triggers) and the data's freshness requirements.

In Stage 2, compaction worker nodes receive these tasks and undertake additional steps beyond standard row-to-columnar conversion. Specifically, they maintain a key–offset mapping in memory or on disk, tracking the highest offset observed for each key (potentially loading this information from the last compacted object). The worker then scans any uncompacted WAL segments. For each record (key,value,offset):

(1) The worker checks the key–offset map.
(2) If the existing offset for that key exceeds the current record's offset, the worker discards this older record.
(3) Otherwise, the worker updates the map to mark this record as the latest and retains it for compaction.

The worker finally writes these retained records—the most recent per key up to the scanned offsets—into a new Parquet file, then returns metadata describing the file's location and offset bounds to the leader. In Stage 3, the leader performs an atomic update of the offset index to reference the newly compacted object.

By rewriting into a fresh Parquet file, Ursa ensures older compacted objects remain intact until the new one is fully written. This approach enables zero-downtime read operations, as existing consumers can continue reading the old compacted objects until the system transitions to the new object. Although the design permits multiple concurrent compaction tasks, Ursa typically ensures only one active Kafka-topic compactor operates on each partition at a time, simplifying concurrency control and ensuring consistency.

5 DATA FLOW

This section describes how Ursa handles data production and consumption while addressing a key challenge: minimizing inter AZ traffic.

5.1 Broker Discovery

In a conventional Kafka cluster every partition has a leader broker [15]; clients therefore begin with a Metadata request to learn which brokers lead which partitions, then open separate connections to those leaders. Even with rack-aware reads this design causes unavoidable cross-AZ hops whenever a producer or consumer sits in a different zone from the leader.

5.1.1 Zone-aware, Leaderless Design. Ursa removes the per-partition leader and lets any broker handle produce or fetch for any partition. To keep traffic local, each broker advertises its AZ (read from Kubernetes or EC2 metadata) in Oxia; this forms a cluster-wide map `<broker → zone>` cached by every node. When a client adds `zone_id=<zone>` to its 'client.id', the receiving broker:

(1) hashes `<topic, partition, zone>` to pick a zonal owner—the broker in the same AZ that should serve that partition;
(2) returns a Metadata reply whose leader field is set to that zonal owner;
(3) lets the client connect directly to the owner for all subsequent traffic.

Because all brokers are stateless, the mapping is purely for locality; internally Ursa remains leaderless.

5.1.2 Partition Assignment. The consistent-hash ring used above keeps all requests for a given partition on the same broker, improving batching and cache hits while eliminating cross-AZ transfers. If a broker fails or is removed for a brown-out (4.2.5), the ring instantly remaps its partitions to healthy peers without data copy, and clients learn the new owners on their next retry. This simplicity aids capacity planning and tail-latency debugging.

5.1.3 Group Coordinator Assignment. Consumer-group metadata is also stored in Oxia. A hash of `<groupID, zone>` selects a group coordinator, preferring a broker in the consumer's AZ; traffic volumes are small, so occasional cross-AZ placement has negligible cost.

Result. Clients normally talk only to brokers in their own zone, achieving high throughput and low network cost without the complexity of leader elections or partition rebalancing.

5.2 Produce Data

Once producers have identified the broker responsible for their target topic-partitions, they begin sending records. Ursa's brokers buffer these Produce requests—aggregating messages from multiple topic-partitions—until a configured size threshold or time interval is reached. The broker then writes the entire batch to the write-ahead log (WAL).

Buffered data is stored in row-based format as a single WAL object. After the broker has successfully persisted this object, it commits the WAL metadata to Ursa's metadata service and acknowledges all produce requests in the batch. Figure 4 illustrates this process.

By combining multiple produce requests for different topic-partitions into one WAL object, Ursa avoids the proliferation of small files and limits how much data remains in memory at any given time. Using a dedicated WAL system also simplifies cluster operations. First, it removes the need for partition rebalancing, a common challenge in Kafka. Second, brokers can be scaled up or down without triggering data transfers. Third, the system acknowledges produce operations only after the WAL has been durably stored, ensuring zero data loss even under broker failures.

5.2.1 Durability, Cost and Latency. Ursa defers produce acknowledgments until the WAL object is persisted and its metadata is committed, mirroring Kafka's `acks=all` semantics. However, Ursa offers a pluggable approach for WAL storage, enabling applications to decide between cost optimization and lower latency.

- **Cost-Optimized WAL.** Data may be buffered for up to 200 ms or until a batch reaches around 4 MB, reflecting the typical write latency of hundreds of milliseconds for object storage. This approach removes the need for Ursa to replicate data across zones because object-storage providers already replicate data internally, substantially lowering networking costs.
- **Latency-Optimized WAL.** For scenarios demanding lower latency, Ursa can use disk-based replication (e.g., Apache BookKeeper) to achieve WAL write latencies on the order of single-digit milliseconds. To accommodate such workloads, the buffering interval might be as low as 5 ms or the batch size might be restricted to 512 KB.

This design lets organizations tailor WAL behavior to specific latency and cost goals, striking an optimal balance for each workload.

5.2.2 Ordering and Idempotency. Ursa preserves Kafka's ordering and idempotency guarantees [4] by leveraging a centralized metadata service that tracks offset indexes for each partition. Once a broker writes a WAL object, the metadata service increments and assigns offsets, returning them to the broker. Consequently, data ordering is finalized upon the metadata commit, rather than when data is flushed to the WAL. This design accommodates large-scale parallel writes—potentially from multiple brokers—to the same partition while upholding ordering and idempotency semantics.

Decoupling metadata flow from data flow similarly simplifies data replication and migration. Offsets can be preserved using conditional writes at specified offsets in the metadata store, which is particularly helpful for multi-datacenter deployments or incremental adoption in existing Kafka environments.

5.3 Consume Data

When a consumer sends a Fetch request specifying an offset 'x', the broker consults the metadata service for a list of storage locations—pointers to WAL objects or compacted Parquet files—for the relevant topic-partition. This list respects Kafka's ordering semantics, allowing the broker to retrieve the requested messages in the correct sequence and forward them to the consumer.

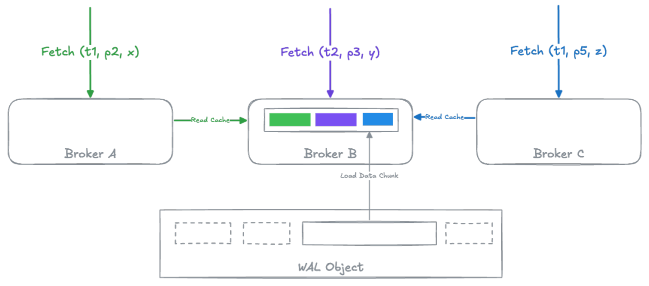

Figure 5: Distributed Cache

Although the logic is straightforward, the leaderless nature of Ursa means any broker can serve fetch requests for any partition, which risks increased read amplification in object storage if multiple brokers repeatedly fetch the same data. Ursa employs additional strategies to mitigate these effects.

5.3.1 Local Write Buffer and Distributed Read Cache. Ursa reduces read amplification by ensuring data is fetched from storage only once per AZ and by coalescing requests for large data chunks.

First, Ursa uses consistent hashing to designate a "responsible" broker for Produce and Fetch operations in each AZ, even though the design remains leaderless at a global scale. Directing both producers and consumers of a partition to the same broker allows tail reads to be served from that broker's local write buffer, bypassing repeated retrievals of recently written data.

Second, if the requested data has already been flushed out of the local buffer, Ursa employs a zone-local distributed cache to further minimize repeated storage reads. The system assigns a specific cache broker in each AZ to fetch data chunks and cache them in memory. Other brokers in the same zone that receive a Fetch request can then obtain the data from the cache broker rather than from object storage. Although this approach may still require an initial fetch per zone, it prevents costly cross-AZ traffic and enables subsequent reads to be fulfilled from memory.

By consistently routing produce and consume operations to a deterministic broker for each partition in a given AZ, Ursa increases cache hit ratios and consequently improves tail-read performance. Meanwhile, storage retrieval remains localized to each zone, mitigating read amplification for catch-up reads. Figure 5 illustrates this design.

In Figure 5, Broker A receives a fetch request for a particular topic, partition, and offset. Broker A looks up which broker is responsible for caching that data chunk, identifies Broker B, and issues a request to Broker B's cache API. If Broker B has already cached the data from a prior fetch, it returns it from memory. Otherwise, Broker B fetches the relevant chunk from storage, caches it, and then serves the request. Subsequent requests for the same chunk—like those from Broker C—are satisfied from Broker B's cache, avoiding redundant data transfers from the underlying storage.

6 USE CASES

Ursa supports the full spectrum of Kafka-style streaming use cases while offering specialized advantages for lakehouse integration.

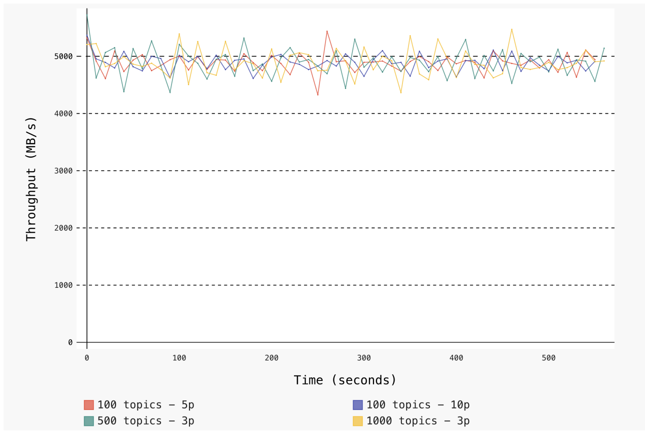

Figure 6: Max throughput: Impact of number of partitions

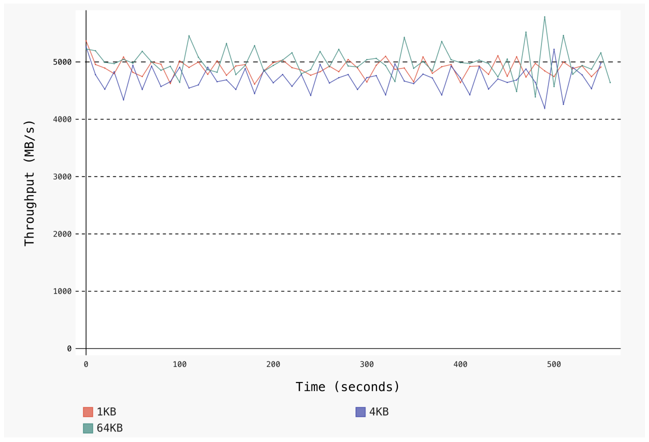

Figure 7: Max throughput: Impact of message size

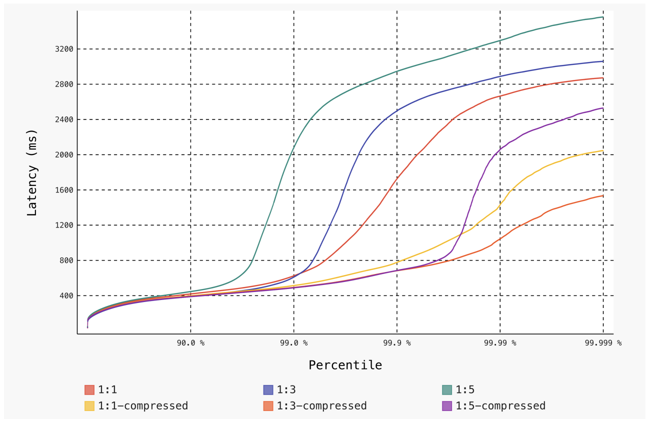

Figure 8: Publish latency under 1GB/s workload

Key applications include: (1) **Real-time analytics** where Ursa reduces latencies from hours to seconds for clickstream analytics, fraud detection, and operational monitoring by ingesting directly into Iceberg/Delta Lake tables; (2) **ML feature engineering** with seamless schema evolution enabling IoT sensor aggregation and real-time inference pipelines; and (3) **Multi-cloud modernization** using CDC streams for incremental warehouse migration while maintaining Delta/Iceberg consistency across regions. Ursa streamlines these workloads by serving both streaming consumers and analytical queries from unified stream-table storage, eliminating traditional ingestion lag while maintaining Kafka compatibility.

7 PERFORMANCE EVALUATION

We evaluated Ursa's performance on Amazon Web Services (AWS) using a cost-optimized WAL configuration, leveraging the Open-Messaging Benchmark to generate event streams at various volumes. All benchmark profiles are publicly accessible at https://github.com/streamnative/openmessaging-benchmark.

Test Environment. The setup comprised 12 ingestion nodes (`m6i.8xlarge`) deployed across three availability zones, with six nodes allocated to producers and six to consumers. An additional 12 nodes (`m6i.8xlarge`) hosted Ursa brokers and Oxia coordination services. Amazon S3 served as the primary storage layer.

7.1 Maximum Throughput

This set of experiments measured Ursa's peak throughput when consumers could keep pace with incoming data, focusing on the system's ability to scale under different partition counts and message sizes.

We first evaluated how increasing the number of partitions affects Ursa's throughput. Figure 6 shows that consume throughput remained around 5 GB/s, regardless of partition count, validating Ursa can scale horizontally without performance degradation.

Next, we investigated whether Ursa's throughput is sensitive to message size. Figure 7 depicts the results for 1 KB, 4 KB, and 64 KB message sizes, showing that both publish and consume throughputs remained around 5 GB/s across all tested configurations.

7.2 Latency

Beyond maximum throughput, we examined publish latency and end-to-end latency under different write-to-read ratios (1:1, 1:3, and 1:5) to assess how fan-out levels affect Ursa's produce latency. As shown in Figures 8, the P99 publish latency remained below 1 s even at elevated fan-out ratios. Compression can reduce storage overhead and further lower these latency distributions. However, the P999 latency rose significantly with higher fan-out, likely due to hitting the network constraints.

7.3 Catchup read

We next evaluated Ursa's ability to process large backlogs under continued high-ingestion rates. We tested backlog draining while continuing to ingest up to 2 GB/s. Latency remained stable except for spikes at the 99.9th percentile as shown in Figure 9.

7.4 Resource Utilization

Under peak loads, Ursa nodes typically operated at 30–60% CPU utilization, with network I/O emerging as the principal bottleneck. Amazon S3 provided sufficient throughput for ingestion, although

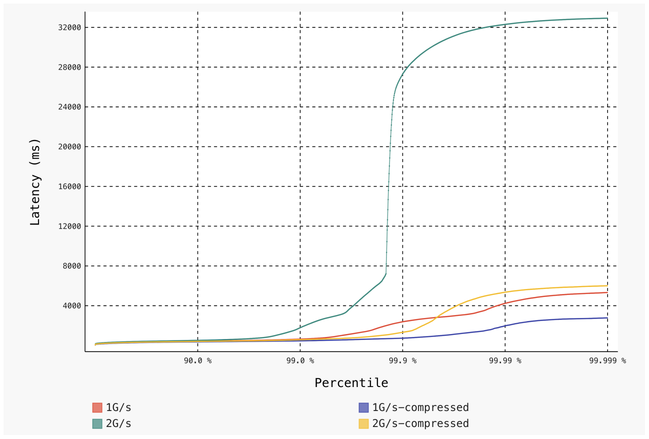

Figure 9: Publish latency during backlog draining

short memory queueing occurred during traffic spikes. Overall, Ursa effectively leveraged available resources while sustaining high throughput and manageable latency.

7.5 Production Deployment

Ursa has run in continuous production on StreamNative Cloud for two years and now starts to back every service tier—from fully managed single-tenant clusters to Bring-Your-Own-Cloud (BYOC) installs. Today StreamNative Cloud operates across AWS, GCP, and Azure in hundreds of clusters, scaling from small developer sandboxes to large scale pipelines that ingest multiple GB/s. Field measurements match our benchmark results: end-to-end latency stays < 1 s while infrastructure spend is ≈ 5% of a comparably sized Kafka stack. One large enterprise cut total infrastructure cost by 10× after replacing a multi-AZ Kafka deployment with a single Ursa cluster that writes Iceberg tables directly to S3. Because brokers are stateless and multi-tenant—with per-namespace auth, quotas, and encryption—multiple business units can safely share capacity, boosting utilisation without extra ops work.

8 DATA STREAMING COST ANALYSIS

To evaluate how Ursa alleviates these cost drivers discussed in Section 2.2, we ran 200 JR clients [2] for two hours, producing ten billion records (180GB compressed with LZ4) into a six-partition topic. This workload was executed on six `m6a.large` nodes.

During this experiment, Ursa's WAL storage received approximately 70,000 objects, and the compaction service generated 2,597 Parquet files (totaling 55GB) for the lakehouse table. Storing the data in Parquet yielded a threefold reduction in size compared to the raw data, while 144,000 S3 read requests were issued to transform row-based WAL objects into Parquet files.

8.1 Cost Comparison

To quantify Ursa's cost advantages, we compared it against two Kafka configurations: one using local disks ("Kafka (Disk)") and another employing tiered storage ("Kafka (TS)"). The same workload was applied to each system. Table 1 summarizes expenses over a two-hour run, assuming seven-day retention for Kafka-based systems (to support data replays) and a one-month retention period for lakehouse data.

Table 1: Cost comparison: Ursa vs Kafka.

	Ursa	Kafka (Disk)	Kafka (TS)
Server EC2	$1.04	$0.52	$0.52
Connector EC2	$0	$0.52	$0.52
Interzone Network	$0	$6	$6
Storage	$1.27	$28.27	$4.94
S3 Requests	$0.40	$0	$0
Total	$2.71	$35.31	$11.98

Server EC2. Ursa uses six `m6a.large` nodes (three brokers plus three lakehouse compaction services), running for two hours at $0.0864 per hour. This totals $1.04, whereas the Kafka-based systems employ three brokers, incurring roughly $0.52 for the same duration.

Connector EC2. Because Ursa natively writes data to the lakehouse, it requires no separate connector layer. By contrast, the Kafka configurations use an additional three-node connector service to fetch from follower replicas before storing data in a columnar format, adding $0.52 in each case.

Interzone Network. Ursa's leaderless, zone-aware design eliminates nearly all cross-AZ data transfers except for minor metadata traffic, contributing $0 in interzone network costs. The Kafka systems, with leader-based replication, produce inter-AZ network usage of $6 in this scenario, as outlined in Section 2.2.

Storage. Ursa removes WAL objects once compaction completes, retaining only the final 55 GB in Parquet format on S3. At $0.023 per GB-month, the total storage cost is $1.27. Kafka (Disk) keeps three replicas of 200 GB for seven days (for replay) plus 55 GB in lakehouse storage, resulting in $28.27. Kafka with tiered storage uses local disks for 20 GB and offloads 180 GB to object storage, alongside 55 GB in the lakehouse, totaling $4.94.

S3 Requests. Ursa triggers about 73 K S3 write operations for WAL objects and a similar number of S3 read operations for compaction. At $0.005 per thousand writes and $0.0004 per thousand reads, the resulting API costs total $0.40. The Kafka systems rely primarily on local disks or specialized tiered storage and thus incur no direct S3 request charges.

8.2 Cost Savings

As shown in Figure 10, Ursa delivers a 92% cost reduction relative to Kafka (Disk) and a 78% reduction compared to Kafka (TS). These savings derive from minimizing cross-AZ transfers, eliminating disk-based replication, and avoiding dedicated connector instances. Furthermore, by storing data in Parquet, Ursa consolidates real-time ingestion and analytical querying into a single, compact dataset, thereby reducing overhead for large-scale batch processing.

Overall, Ursa's leaderless, lakehouse-native architecture directly addresses the cost drivers highlighted in Section 2.2, enabling organizations to accommodate real-time ingestion needs without incurring the excessive infrastructure expenses common in traditional streaming systems.

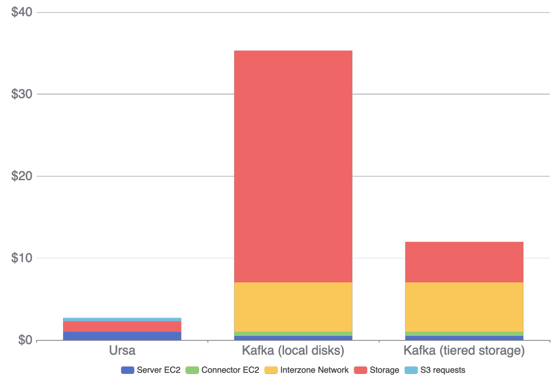

Figure 10: Data Streaming Cost Comparison: Ursa vs. Kafka (lower is better).

9 RELATED WORK

Modern streaming systems commonly adopt leader-based replication protocols to ensure consistency under failures. Notably, Apache Kafka employs an In-Sync Replica (ISR) mechanism, Redpanda uses Raft [26], and Apache Pulsar integrates a Paxos-like protocol [23]. While robust, these architectures anchor every partition to a designated leader, inflating inter-AZ network traffic and complicating fail-over in multi-zone deployments. Ursa's leaderless design removes that single-leader bottleneck, reducing cross-zone chatter and simplifying recovery.

A second line of work eliminates local disks entirely. For example, WarpStream [6] or Confluent Freight Cluster [5] employs a "zero-disk" architecture, persisting all events directly to an object store. Although this reduces local storage dependencies, WarpStream maintains a proprietary format and centralized metadata service, limiting interoperability. Ursa, by comparison, persists data in open formats—Apache Iceberg or Delta Lake—while placing metadata under the user's control. This enables compatibility with external engines (e.g., Databricks, Snowflake) and lowers total costs. In addition, WarpStream and Confluent Freight Clusters focus only on object storage based solution that can only support latency-relaxed workloads that can accept sub-second latency. Ursa, in contrast, supports pluggable WAL storage that is able to support both latency-optimized and cost-optimized workloads.

Flink Dynamic Tables [11] and Apache Paimon [27] pursue the opposite, table-first path to stream–table unification. Dynamic Tables are a logical abstraction inside Flink that treats streams as continuously updating tables, yet the state is ephemeral and confined to the running job. Apache Paimon (born as Flink Table Store) materialises that state persistently: it stores updates in an LSM-style layout, enabling ACID snapshots and real-time queries across Flink, Spark, or Hive. Ursa, by contrast, is stream-first and Kafka-API compatible—producers and consumers interact through log semantics while the engine simultaneously commits those logs to Iceberg/Delta tables. Thus, Dynamic Tables / Paimon maximise integration with the Flink SQL API for CDC and high-frequency updates, whereas Ursa offers a drop-in Kafka replacement that collapses streaming and lakehouse layers with a leaderless, cost-oriented architecture. Both achieve stream–table duality but target different ecosystems and optimisation fronts.

Connector frameworks like Kafka Connect and Pulsar I/O provide a route for moving streaming data into external systems but generally require separate deployments and configurations. These connector-based pipelines also risk introducing data duplication and added operational overhead. Ursa circumvents this fragmentation through a zero-ETL design, directly stores data in real-time data streams and exposing the underlying data as Lakehouse tables—achieving "stream-table duality" within a single engine with a single copy of data.

Finally, while systems such as Delta Lake and Apache Iceberg support limited streaming ingestion, they often rely on micro-batch cycles and do not natively provide a Kafka streaming abstraction. Ursa augments these table formats with a write-ahead log and offset indexing, thus enabling both near-real-time ingestion and flexible storage tiering (latency-optimized vs. cost-optimized) within the same architecture.

10 CONCLUSION

We have introduced Ursa, a leaderless, lakehouse-native engine that significantly simplifies and lowers the cost of real-time data ingestion. By removing disk-based broker storage and leader-based replication, Ursa mitigates the key cost drivers of multi AZ environments. Integration with open table formats further streamlines data management and analytics workflows, delivering high throughput and near-real-time latency at a fraction of traditional costs.

Looking ahead, we plan to expand Ursa's capabilities in four directions: (i) leveraging machine learning to dynamically switch between cost-optimized (object-based) and latency-optimized (disk-based) modes, (ii) extending stream–table duality by generating real-time change logs from table updates, (iii) offering Ursa's storage as a standalone library so applications can directly append Arrow or Parquet data without broker intermediaries, and

(iv) exposing live WAL objects through a catalog extension so engines can scan hot WAL rows alongside Parquet, enabling millisecond-fresh OLAP without compaction delays.

ACKNOWLEDGMENTS

Ursa is the culmination of sustained effort by numerous individuals—past and present—who worked tirelessly to realize the vision of a truly lakehouse-native data streaming platform. We thank all who contributed their time, expertise, and creativity along the way. We are especially grateful to our customers and users, whose real-world scenarios, requirements, and continual feedback shaped Ursa's features and guided our development. We also extend our thanks to the Apache Pulsar, Apache Iceberg, and Delta Lake communities for their invaluable insights and feedback. Lastly, we appreciate the guidance from contributors and committers in the Apache Kafka community, whose support on Kafka protocol internals enabled our Kafka compatibility.

REFERENCES

[1] Michael Armbrust, Tathagata Das, Liwen Sun, et al. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proceedings of the*

VLDB Endowment 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560
[2] JR Tool Authors. 2025. JR: A High-Performance Data Generation and Benchmarking Tool. Retrieved March 1, 2025 from https://github.com/jr-tool
[3] Praseed Balakrishnan. 2022. Redpanda Cloud brings the fastest Kafka® API to the cloud. Retrieved March 1, 2025 from https://redpanda.com/blog/introducing-redpanda-cloud-for-kafka
[4] Confluent. 2017. Exactly-Once Semantics in Apache Kafka. Retrieved March 1, 2025 from https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/
[5] Confluent. 2025. Freight Clusters: High-Capacity Kafka Deployments. Retrieved March 1, 2025 from https://www.confluent.io/product/freight-cluster/
[6] Confluent. 2025. WarpStream: A Cloud-Native, Zero-Disk Apache Kafka Alternative. Retrieved March 1, 2025 from https://www.warpstream.com
[7] Inc. Confluent. 2025. Kafka Connect Fundamentals: What is Kafka Connect? Retrieved March 1, 2025 from https://www.confluent.io/blog/kafka-connect-tutorial/
[8] Databricks. 2025. Databricks Unity Catalog. Retrieved March 1, 2025 from https://www.databricks.com/product/unity-catalog
[9] Databricks. 2025. Delta Lake UniForm: Unified Table Formats for Interoperability in the Lakehouse. Retrieved March 1, 2025 from https://www.databricks.com/blog/2023/05/16/introducing-delta-lake-uniform-way-unify-your-data.html
[10] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. 137–150. https://www.usenix.org/legacy/events/osdi04/tech/dean.html
[11] Apache Flink. 2025. Flink Dynamic Tables. Retrieved Jun 9, 2025 from https://nightlies.apache.org/flink/flink-docs-master/docs/dev/table/concepts/dynamic_tables/
[12] Apache Software Foundation. 2025. Apache BookKeeper. Retrieved March 1, 2025 from https://bookkeeper.apache.org/
[13] Apache Software Foundation. 2025. Apache Flink. Retrieved March 1, 2025 from https://flink.apache.org/
[14] Apache Software Foundation. 2025. Apache Iceberg. Retrieved March 1, 2025 from https://iceberg.apache.org/
[15] Apache Software Foundation. 2025. Apache Kafka. Retrieved March 1, 2025 from https://kafka.apache.org/
[16] Apache Software Foundation. 2025. Apache Parquet. Retrieved March 1, 2025 from https://parquet.apache.org/
[17] Apache Software Foundation. 2025. Apache Pulsar. Retrieved March 1, 2025 from https://pulsar.apache.org/
[18] Apache Software Foundation. 2025. Apache Spark. Retrieved March 1, 2025 from https://spark.apache.org/
[19] Apache Software Foundation. 2025. Apache XTable (Incubating): Universal Table Formats Interoperability. Retrieved March 1, 2025 from https://incubator.apache.org/projects/xtable.html
[20] Trino Software Foundation. 2025. Trino. Retrieved March 1, 2025 from https://trino.io/
[21] Sijie Guo. 2024. A Guide to Evaluating the Infrastructure Costs of Apache Pulsar and Apache Kafka. Retrieved March 1, 2025 from https://streamnative.io/blog/a-guide-to-evaluating-the-infrastructure-costs-of-apache-pulsar-and-apache-kafka
[22] Snowflake Inc. 2025. Snowflake Open Catalog. Retrieved March 1, 2025 from https://www.snowflake.com/en/product/features/open-catalog/
[23] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (December 2001), 51–58. https://lamport.azurewebsites.net/pubs/paxos-simple.pdf
[24] Matteo Merli. 2023. Introducing Oxia: Scalable Metadata and Coordination. Retrieved March 1, 2025 from https://streamnative.io/blog/introducing-oxia-scalable-metadata-and-coordination
[25] et al. Michael Armbrust, Ali Ghodsi. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf 11th Conference on Innovative Data Systems Research (CIDR), January 2021, Virtual.
[26] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 14)*. 305–319. https://doi.org/10.5555/2643634.2643666
[27] Apache Paimon. 2025. Apache Paimon. Retrieved Jun 9, 2025 from https://paimon.apache.org/