Synapse Platform Team

Building Nimbus Stream

A Technical Walkthrough of Synapse's Real-Time Event Pipeline

Maya Chen
Staff Engineer, Platform Team

Engineering All-Hands ▪ July 2026

Agenda

What we'll cover in the next 25 minutes

1. Problem & context
2. System architecture
3. Core algorithm: adaptive partitioning
4. Results & benchmarks
5. Lessons learned

Why this matters

Nimbus Stream now routes **2.4B events/day** across Synapse's product surface, powering live dashboards for customers like **Vertex Analytics** and **Meridian Retail**. This talk is the design walkthrough we wish we'd had before we built it.

01 ▪ Problem & Context

Why the old pipeline stopped scaling

The Problem

Our batch pipeline was quietly falling behind

- Legacy pipeline batched events every **15 minutes** via nightly Apex Cloud jobs
- Customer dashboards (Vertex Analytics, Meridian Retail) demanded **sub-second** freshness
- A single noisy tenant could stall the batch window for *everyone*
- On-call was paging on "stale data" incidents 3–4 times a week

Cost of staleness

- 15 min median lag
- 4 sev-2 incidents / week
- 3 enterprise accounts at risk

Requirements vs. Non-Goals

Scoping the rebuild honestly

✔ Requirements

- p99 end-to-end latency < 2s
- Horizontal scale to 5M events/min
- Exactly-once delivery to sinks
- Graceful shard rebalancing

✖ Non-Goals

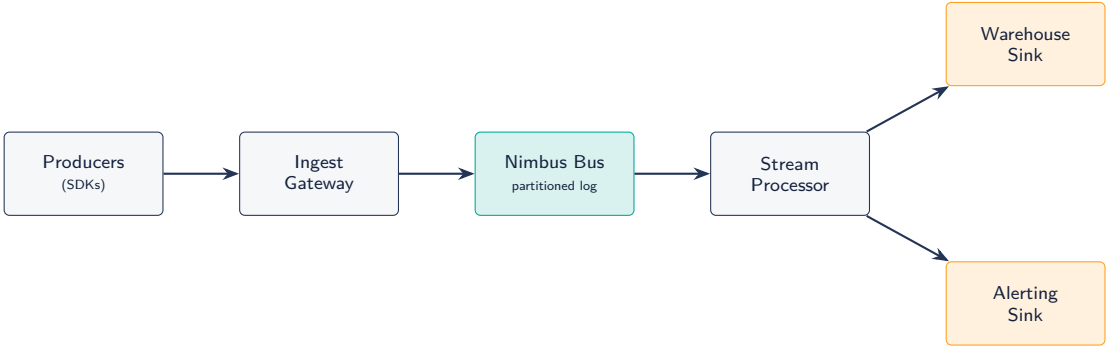
- Cross-region active-active writes
- Sub-millisecond streaming SQL
- Replacing the warehouse layer
- Supporting on-prem deployment

02 ▪ System Architecture

From ingest gateway to sink, end to end

Architecture at a Glance

Five stages, one guarantee: exactly-once



- **Ingest Gateway** validates schema and stamps a monotonic offset
- **Nimbus Bus** is a partitioned append-only log (32 shards, 3x replicated)
- **Stream Processor** holds windowed state; scales independently of the bus

Data Flow Walkthrough

Tracing one event from click to chart

1. Client SDK emits event with idempotency key
2. Gateway assigns partition via `hash(tenant_id)`
3. Bus appends to the shard's log, fsyncs to 2 of 3 replicas
4. Processor consumes in order, updates windowed aggregates
5. Sink writer commits offset only after downstream ack

Stage	Budget
Gateway	40 ms
Bus append	120 ms
Processing	300 ms
Sink commit	90 ms

Table: Latency budget per stage

03 ▪ Core Algorithm

Adaptive partitioning & consistent hashing

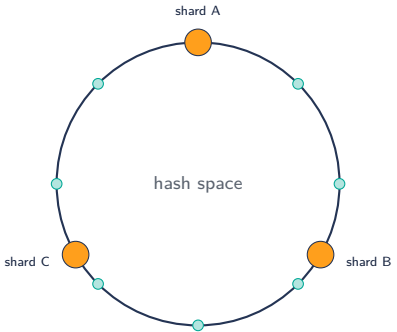
Consistent Hashing Ring

How we place shards without a full reshuffle

Algorithm 1: Route(key, ring)

Input: event key k, ring R
Output: owning shard s

```
h ← hash(k)
s ← first node in R clockwise from h
if s is marked draining then
    s ← next live node clockwise
return s
```



Only keys between the draining node and its clockwise neighbor move during a rebalance — roughly $1/N$ of all keys, not all of them.

Live Coding: The Rebalance Handler

What actually runs when a shard drains

```go
func (r *Ring) Rebalance(draining ShardID) error {
    next := r.NextLive(draining)
    keys := r.KeysOwnedBy(draining)

    for _, k := range keys {
        // Copy before repoint: never orphan a key mid-flight
        if err := r.Replicate(k, next); err != nil {
            return err // caller retries; ring stays consistent
        }
        r.Repoint(k, next)
    }
    r.MarkDrained(draining)
    return nil
}
```

Replicate-then-repoint keeps the ring consistent even if the handler crashes mid-rebalance — Nimbus Bus and the processor both read from the same ring version.

Complexity & Guarantees

What the algorithm buys us

Operation	Cost	Guarantee
Route lookup	$O(\log N)$	deterministic per ring version
Shard add/remove	$O(K/N)$ keys move	no full reshuffle
Rebalance	background, throttled	zero dropped events

Table: N = shard count, K = total keys

Why not a simple modulo hash?

`shard = hash(key) % N` reshuffles *every* key whenever N changes. At 32 shards that meant re-copying **100%** of state on every scale-out — consistent hashing bounds that to about **3%**.

04 ▪ Results

What changed in production

Benchmarks: Before vs. After

Measured over a 30-day production window

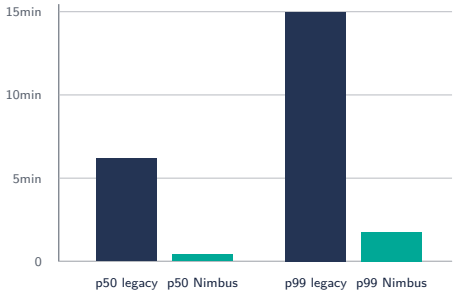
Metric	Legacy Batch	Nimbus Stream
p50 end-to-end latency	6.2 min	410 ms
p99 end-to-end latency	15.0 min	1.8 s
Max sustained throughput	620K events/min	5.1M events/min
Sev-2 incidents / week	3.6	0.2

Table: 30-day rolling averages, production traffic

A 63% reduction in on-call load came almost entirely from removing the shared batch window as a single point of contention.

Latency, Visually

p50 / p99 side by side



- p50 dropped from **6.2 min** to **410 ms**
- p99 dropped from **15.0 min** to **1.8 s**
- Tail latency is now dominated by processor window flush, not the bus
- Next target: p99 $<$ 1s by tightening window size

05 ▪ Lessons Learned

What we'd do differently next time

Lessons Learned

Pros we'd repeat, cautions we'd heed

Do Again

- Replicate-then-repoint for every state migration
- Ship the ring as a versioned, immutable snapshot
- Load-test rebalancing *before* the algorithm, not after

Watch For

- Hot tenants can still overload a single shard
- Virtual nodes needed tuning — too few caused skew
- Backpressure signals must reach producers, not just the bus

Thank You

Questions, war stories, and disagreements welcome

maya.chen@synapse.dev

github.com/synapse/nimbus-stream