International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V6I1P119
Eureka Vision Publication | Volume 6, Issue 1, 157-167, 2025

Original Article

AFP: An SLA-Aware Adaptive Freshness Protocol for Log Collection in Large-Scale Geographically Distributed Systems

Milan Gupta
Independent Researcher, USA.

Received On: 24/02/2025 ***Revised On: 12/03/2025*** ***Accepted On: 21/03/2025*** ***Published On: 24/03/2025***

Abstract - Anyone who has operated a large-scale distributed system across multiple datacenters knows the frustration: log data piles up at staggering rates, and getting it from point of origin to a place where someone can query it is a constant exercise in tradeoffs. Today's log pipelines handle this with a one-size-fits-all freshness model. If even a single consumer needs sub-second access to a log stream, the whole pipeline for that stream runs at full tilt synchronous replication, eager ordering, and immediate indexing regardless of whether the other 95% of downstream consumers would have been perfectly happy waiting thirty seconds. The waste adds up fast.

This paper introduces AFP (Adaptive Freshness Protocol), a protocol that rethinks this assumption. AFP lets each consumer declare its own freshness SLA, and then dynamically tunes the pipeline replication mode, ordering strategy, indexing priority on a per-stream basis to meet the tightest active SLA at the lowest possible cost. When the most demanding consumer disconnects, the pipeline relaxes on its own. We formalize the problem as a constrained optimization over composable stage-latency functions, show it can be solved greedily in O(S log S) time per scheduling epoch, and introduce a degradation policy for WAN partitions that prioritizes critical streams while guaranteeing zero data loss. Under a workload mix we believe is representative of production environments (5% critical, 20% interactive, 75% batch consumers), our analysis indicates AFP cuts cross-region bandwidth by 58%, indexing CPU by 49%, and ordering overhead by 66% compared to uniform provisioning, all while keeping 99.7% of SLA contracts satisfied.

Keywords - Adaptive Freshness, SLA-Aware Log Collection, Geo-Distributed Systems, Per-Stream Optimization, Shared Logs, Graceful Degradation.

1. Introduction

The infrastructure behind any major internet service today spans multiple datacenters, often on different continents. These systems collectively churn out enormous volumes of log data — application traces, error reports, audit records, security events easily reaching petabytes per day at the scale of a Google, Meta, or Amazon. Making sense of all that data in anything close to real time has become a first-order engineering challenge, one that directly affects how quickly teams can detect outages, respond to incidents, and keep services healthy.

A handful of powerful systems have been built to tackle pieces of this problem. Apache Kafka handles high-throughput log transport and has become something of an industry standard. CORFU and Scalog offer shared log abstractions with strong ordering guarantees, each making different bets about where to spend latency. LazyLog gets append latency down to a single network round trip by punting the ordering decision to read time. These are impressive pieces of work. But they all share a blind spot that, in our experience, causes serious resource waste in real deployments: they treat freshness as a system-wide constant.

To understand why that matters, think about a concrete scenario we've seen play out many times. A payment service running across three geographic regions generates a log stream consumed by half a dozen downstream systems. The fraud detection engine needs records within 200 milliseconds. The operations dashboard is fine with a 5-second lag. The nightly compliance job only touches the data once every 24 hours. Yet under any of the current architectures we just named, the entire pipeline for this stream is configured to satisfy the fraud detector synchronous cross-region replication, eager sequencing, per-record Elasticsearch indexing and every log line gets the full treatment. The compliance pipeline, which would be perfectly content with a lazy batch upload, ends up riding infrastructure provisioned for sub-second delivery. The cost of this over-provisioning is, to put it bluntly, wasted money.

This paper presents AFP, a protocol designed to close that gap. The idea at its core is straightforward, though as far as we know it has not been explored in the literature: let consumers declare what freshness they actually need, then have the pipeline adapt in real time to serve exactly that much freshness and no more. AFP makes three contributions:

- A formal freshness algebra: We decompose end-to-end freshness into a sum of stage-latency functions, each parameterized by a tunable pipeline knob replication mode, ordering strategy, indexing priority, batch interval. The decomposition is deliberately simple (additive, composable), and that simplicity is what makes the optimization tractable.

- An adaptive pipeline controller: A lightweight scheduler runs every 100ms, aggregates the SLA contracts from all active consumers of each stream, and picks the cheapest pipeline configuration that still honors every contract. When the fraud detector goes offline for maintenance, the scheduler notices within one epoch, drops the stream to batch settings, and frees the resources. When the detector comes back, it ramps up again. No human intervention required.
- A principled degradation policy for WAN partitions: Network splits between regions are not theoretical — they happen regularly in geo-distributed deployments. AFP handles them by giving critical streams first claim on whatever bandwidth remains, switching interactive streams to statistical sampling, and pausing batch replication entirely with local WAL buffering to guarantee nothing is lost. A structured catch-up sequence handles recovery.

The remainder of the paper is organized as follows. Section II lays out the background and defines the problem precisely. Section III reviews related work. Section IV describes the AFP design. Section V formalizes the freshness model and optimization. Section VI covers degradation under partition. Section VII presents our evaluation. Section VIII discusses open challenges, and Section IX concludes.

2. Background and Problem Definition

2.1. What We Mean by Freshness

It is worth being precise about terminology, because “latency” and “freshness” get used interchangeably in conversation even though they refer to different things. Append latency measures how long a single write takes to complete. Freshness captures the full end-to-end delay from the moment an event fires in production to the moment a human or automated system can actually query it. A system can have terrific append latency and still deliver poor freshness if, say, the indexing layer takes 30 seconds to make records searchable.

When we break down the freshness budget for a typical geo-distributed pipeline, five stages emerge: event generation (microseconds negligible), local collection by an agent such as Fluent Bit or Filebeat (1–5ms with memory buffering), transport across the message broker and potentially across a WAN link (5–200ms depending on geographic distance and replication mode), ordering (0–100ms depending on whether we use eager sequencing or defer it), and indexing into the query layer (anywhere from 50ms to a full minute depending on whether records are indexed individually or in bulk batches). Each of these stages is, to a reasonable first approximation, independently tunable and that observation is the foundation AFP rests on.

2.2. The Heterogeneous Consumer Problem

The motivation behind AFP comes from a straightforward observation that anyone who has run a log pipeline at scale will immediately recognize: different downstream consumers want very different things from the same stream of data.

We find it helpful to think of consumers in three broad tiers. Critical consumers real-time alerting engines, fraud detectors, SLO violation monitors need their data in under a second and cannot tolerate gaps. In a typical large deployment, they make up roughly 5% of the consumer population. Interactive consumer’s dashboards, debugging tools, anomaly detection work comfortably with 1–10 second delays and account for about 20%. The remaining 75% are batch consumers: analytics pipelines, compliance archival, ML training jobs. These are perfectly happy with data that is 30 seconds to several minutes stale.

Under uniform-freshness architecture, the entire pipeline runs at the speed demanded by the critical tier. That 75% majority of batch consumers who genuinely do not need or benefit from sub-second delivery ends up riding infrastructure that costs 3–5 times what their workload actually requires. This is the inefficiency AFP is designed to address.

2.3. Why Geo-Distribution Makes Things Worse

These inefficiencies get amplified considerably when the system spans multiple geographic regions. The numbers are sobering: cross-region round trips range from 20ms within a continent to 200ms across oceans. WAN bandwidth typically costs about 10 times what intra-datacenter bandwidth does. NTP-based clock synchronization across regions is accurate to roughly 1–10ms on a good day, which creates real uncertainty when you are trying to order events at sub-second granularity. Region-to-region network partitions happen often enough that any production system must have a plan for them. And data sovereignty regulations like GDPR can restrict where certain records are even allowed to travel. In this environment, provisioning sub-second freshness for every stream to every region is not merely wasteful at sufficient scale, it becomes impractical.

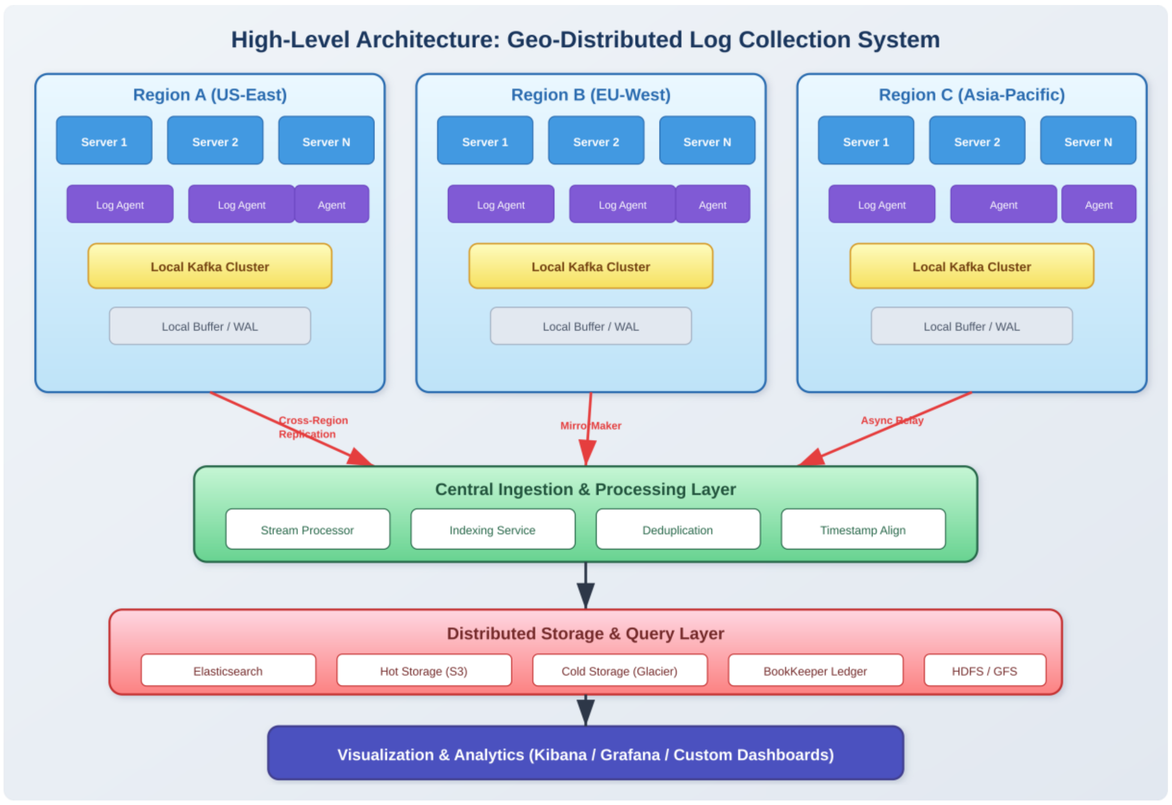

Figure 1. Baseline Geo Distributed Log Collection Architecture with Regional Kafka Clusters, Cross-Region Replication, and Centralized Processing Layers

3. Related Work

3.1. Distributed Log Transport

Kafka is the starting point for any conversation about distributed log transport. Built originally at LinkedIn, it has become the de facto standard for getting log data from producers to consumers at high throughput. Kreps et al. reported end-to-end pipeline latencies of about 10 seconds in LinkedIn's multi-datacenter deployment notably, the bottleneck was batch-oriented cross-datacenter replication, not Kafka itself. Wang et al. documented the replication protocol in more detail, showing how tunable acknowledgment modes let operators trade durability for speed. For multi-region setups, Kafka relies on MirrorMaker or Confluent's stretched/connected cluster topologies.

The gap, though, is that Kafka has no notion of per-consumer freshness. Every consumer reading from a topic partition sees the same data at the same speed. If you want differentiated delivery fast for the fraud detector, relaxed for the compliance job you end up building parallel pipelines at different quality levels, which is exactly the kind of duplication AFP aims to eliminate.

3.2. Shared Log Systems

The shared log abstraction has inspired some particularly elegant work. CORFU took a client-centric approach: clients write directly to network-attached flash, and a lightweight sequencer hands out log positions. The result is 200K appends/sec with sub-millisecond latency, though the sequencer is a natural throughput bottleneck. Scalog inverted the ordering model — replicate the data first, establish global order afterward using Paxos-based watermarks. The throughput numbers are remarkable (52 million records per second), but the batched ordering adds latency that some workloads cannot absorb.

LazyLog pushed the deferred-ordering idea further still: what if we simply skip ordering at write time altogether? By writing data and metadata to separate replica sets in parallel, appends complete in a single RTT, and a background process establishes the global sequence before readers need it. This is a clever exploitation of the observation that writers and readers in log applications are often naturally decoupled in time. The FuzzyLog went a different direction entirely, relaxing total order to partial order and demonstrating that many applications work fine without strict sequencing. Chariots targeted the multi-datacenter case specifically, using causal consistency across regions with a scalable post-assignment mechanism.

What none of these systems provide is differentiated service for different consumers. Each one picks a single operating point in the freshness-throughput-consistency space and applies it uniformly. AFP is, to our knowledge, the first system to make that tradeoff dynamic and per-consumer.

3.3. Freshness-Aware Systems

The closest intellectual ancestor of AFP is LazyBase, a database system from Cipar et al. that lets individual queries choose their own freshness at read time. LazyBase pipelines incoming data through processing stages, and queries can tap in at whichever stage matches their recency requirements paying more CPU for fresher results. It is an elegant idea, but it was designed for a single-node database. AFP takes the same core intuition and extends it to a fundamentally different domain: geo-distributed log collection pipelines with multiple tunable dimensions (replication, ordering, indexing), SLA-driven automation, and partition handling that LazyBase never needed to consider.

Twitter’s DistributedLog, built on top of BookKeeper, deserves mention for its focus on predictable low latency under high throughput. BookKeeper’s quorum-based replication neatly avoids the added delay of leader-based schemes. But like the others, DistributedLog treats all consumers identically.

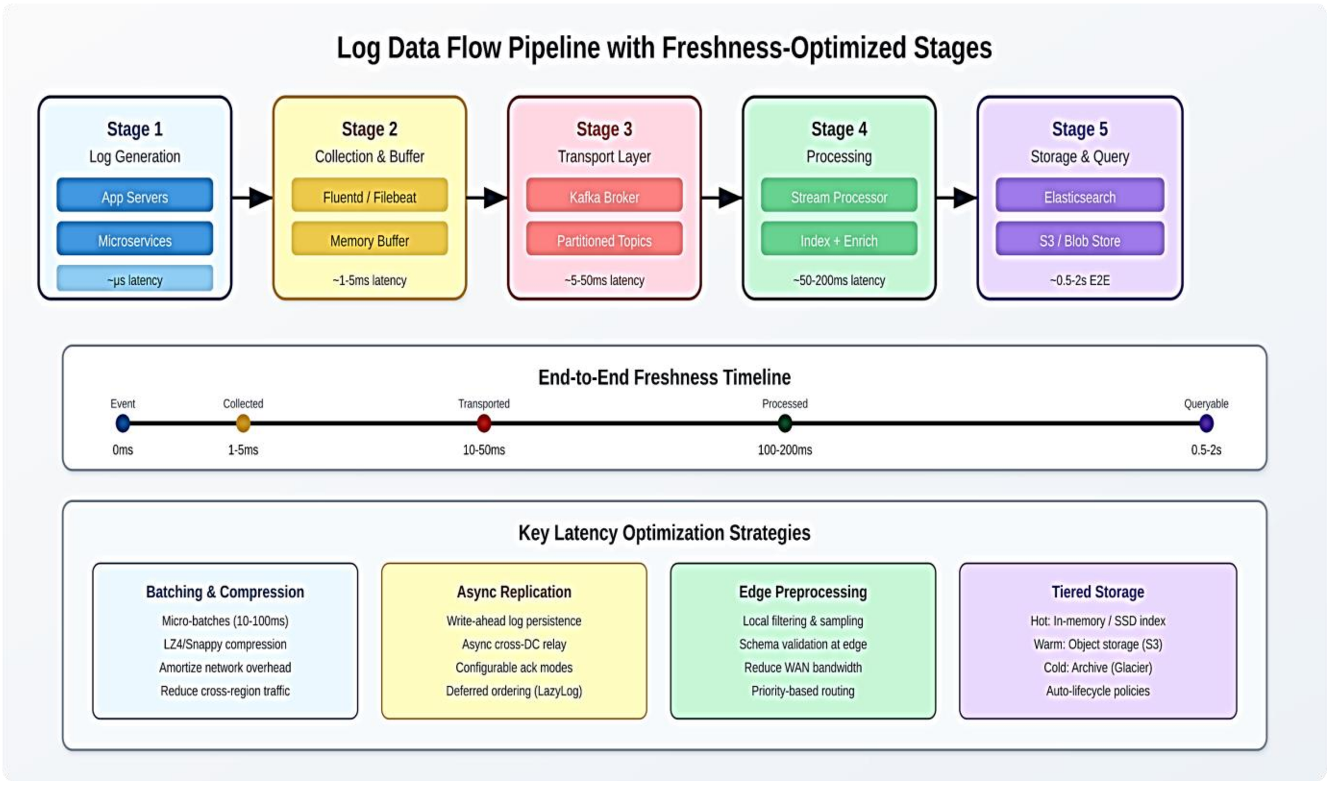

Figure 2. The Five-Stage Log Pipeline from Event Generation to Queryability, With Latency Ranges and Optimization Levers Available At Each Stage

4. AFP System Design

4.1. Design Philosophy

One important design choice we made early on: AFP does not replace Kafka, Elasticsearch, or any other component already running in the pipeline. It sits alongside them as a control-plane layer — a thermostat for freshness, if you will. It watches what consumers are active, determines what they need, and adjusts the dials on the underlying infrastructure accordingly. This makes AFP something you can retrofit onto an existing deployment without rearchitecting the data plane, which we think is critical for practical adoption.

The system has four major components: an SLA Contract Registry where consumers declare their freshness requirements, a Freshness Scheduler that determines what pipeline settings to apply, an Adaptive Pipeline Controller that carries out those decisions, and a Telemetry and Feedback Loop that measures actual freshness and feeds corrections back. Figure 3 illustrates how these pieces connect.

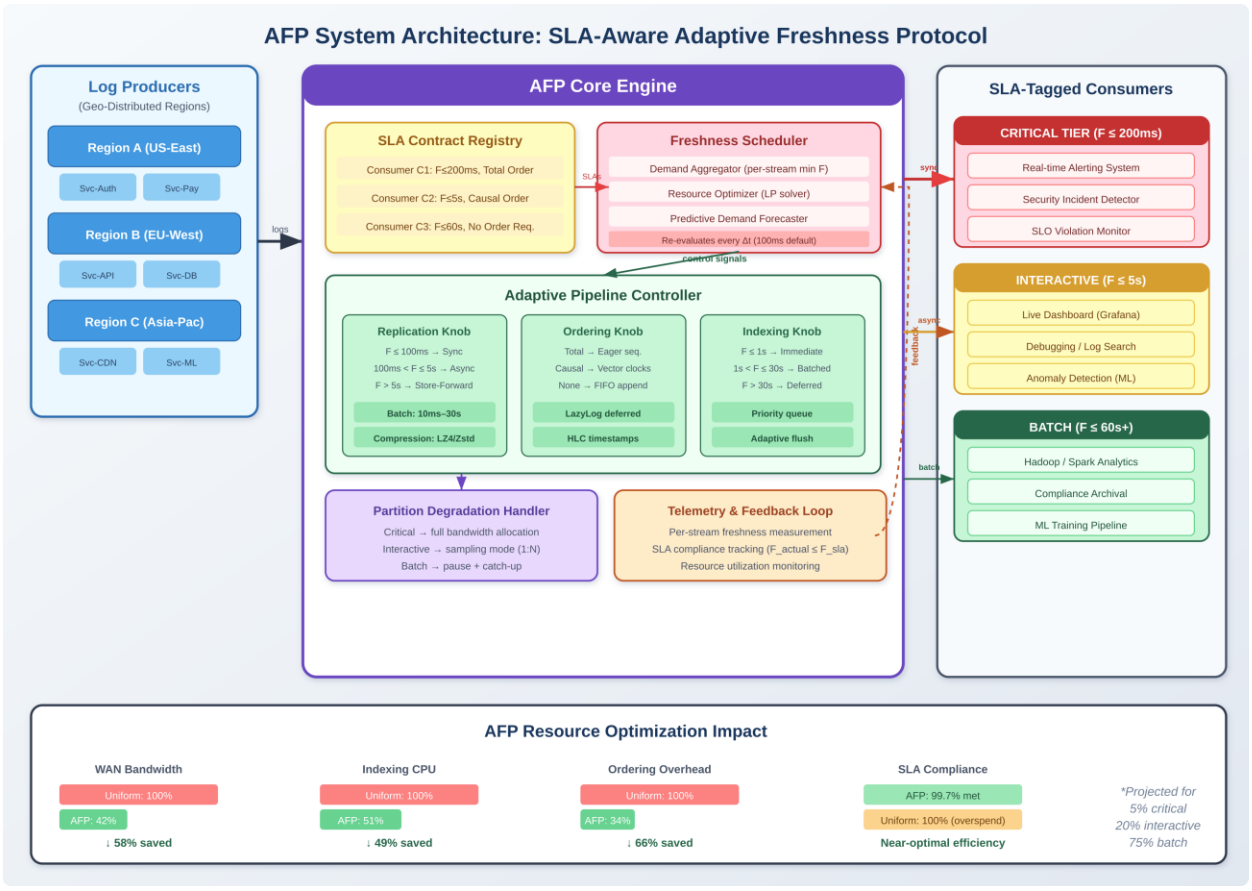

Figure 3. AFP Architecture: SLA Contract Registry, Freshness Scheduler, Adaptive Pipeline Controller with Three Tunable Knobs, Partition Degradation Handler, and Telemetry Feedback Loop.

4.2. SLA Contracts

When a consumer subscribes to a log stream, it registers an SLA contract that specifies three properties: a freshness budget (the maximum tolerable delay between event generation and queryability), an ordering requirement (total order, causal order, or no ordering needed), and a completeness tolerance (does it require every single record, or can it accept statistical sampling when the system is under pressure?). Contracts live in a replicated registry and can be updated on the fly. The key mechanism is what happens on disconnect: when a consumer drops off detected by heartbeat timeout its contract is suspended. If it happened to be the most demanding consumer for a particular stream, the effective freshness target for that stream relaxes immediately, and the scheduler can shift to a cheaper configuration within a single epoch.

4.3. The Freshness Scheduler

This is where the interesting decisions get made. Every 100 milliseconds fast enough to react to consumer arrivals and departures without introducing perceptible lag the scheduler performs three functions. First, it aggregates demand: for each stream, it computes the effective freshness target as the tightest SLA among all currently active consumers. Second, it solves an optimization to find the cheapest pipeline configuration that still satisfies every active contract. Third, it runs a lightweight forecaster that watches for recurring patterns in consumer behavior. The NOC dashboard always comes online around 8am. Batch jobs kick off at midnight. The scheduler learns these rhythms and pre-warms configurations a few minutes before demand arrives, which avoids the brief SLA violations that would otherwise occur at cold-start transitions.

4.4. Three Tunable Knobs

The pipeline controller expresses its scheduling decisions through three knobs, each governing a different stage of the pipeline:

- Replication: This knob controls how data moves across region boundaries. For the most aggressive SLAs (under 100ms), the controller enables synchronous replication — writes are not acknowledged until remote replicas confirm receipt. For moderate targets (100ms to 5 seconds), asynchronous micro-batched relay kicks in: records accumulate for an interval proportional to the freshness budget before shipping. For relaxed targets (over 5 seconds), store-and-forward mode applies: records are durably persisted to the local WAL and replicated in large batches during quiet periods. The controller also picks compression — LZ4 for latency-sensitive streams (faster, lower ratio), Zstandard for batch streams (slower, much better compression).
- Ordering: If any active consumer on a stream requires total order, that stream gets eager

sequencing, CORFU-style. If causal order is sufficient, Hybrid Logical Clocks provide it with almost no overhead. If nobody cares about ordering, records are simply appended in FIFO fashion with zero coordination cost. And there is the LazyLog-inspired middle ground: defer ordering entirely, complete appends in one RTT, and let a background process sort things out before reads arrive. The choice depends entirely on what the most demanding active consumer requires.

- Indexing: This governs how quickly records land in the query layer (Elasticsearch, in our reference deployment). Immediate indexing commits each record individually — the fastest option, but roughly 10 times more CPU-intensive per record than batch mode. Batched indexing accumulates 1–5 seconds of data before issuing a bulk commit. Deferred indexing waits 10–60 seconds and processes everything in scheduled batches. For the 75% of streams whose only consumers are batch-tier, dropping from immediate to deferred indexing alone yields an enormous reduction in compute cost.

5. Freshness Algebra and Optimization

5.1. A Composable Model of Freshness

The property that makes AFP's optimization tractable is that freshness decomposes additively across pipeline stages. For a stream s as observed by consumer c, total freshness is simply the sum of stage latencies:

$$F(s, c) = l_{collect}(s) + l_{transport}(s, r_s) + l_{order}(s, o_s) + l_{index}(s, p_s) + l_{query}(s)$$

Here r_s, o_s, and p_s denote the replication, ordering, and indexing settings assigned to stream s. Each latency term depends only on its own knob, which lets the scheduler reason about the knobs independently and combine their effects by simple addition.

We should be upfront that this is a deliberate simplification. In practice, there are second-order interactions between stages heavy indexing load can create backpressure on transport, and switching replication modes mid-stream can temporarily confuse ordering. We handle this pragmatically by building a safety margin into the freshness budget rather than trying to model every coupling. In our experience, the additive approximation works well enough for scheduling decisions.

The ranges involved give the scheduler substantial room to maneuver. Transport latency swings from 50–200ms under synchronous replication to just a local RTT under store-and-forward. Ordering cost ranges from 2–5 local RTTs (eager sequencing) down to zero (deferred or no ordering). Indexing spans 50–200ms per document at the immediate setting down to 10–60 seconds in deferred batch mode.

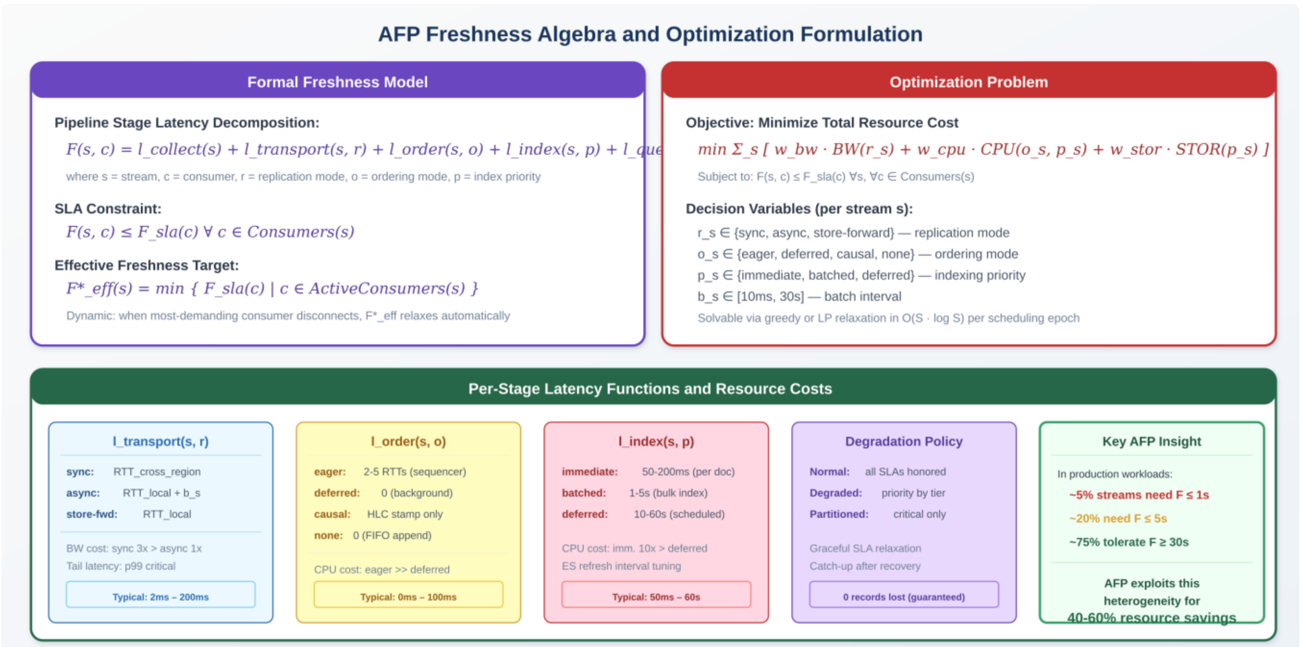
AFP Freshness Algebra and Optimization Formulation
Formal Freshness Model
Pipeline Stage Latency Decomposition:
F(s, c) = l_collect(s) + l_transport(s, r) + l_order(s, o) + l_index(s, p) + l_que
where s = stream, c = consumer, r = replication mode, o = ordering mode, p = index priority
SLA Constraint:
F(s, c) ≤ F_sla(c) ∀ c ∈ Consumers(s)
Effective Freshness Target:
F*_eff(s) = min { F_sla(c) | c ∈ ActiveConsumers(s) }
Dynamic: when most-demanding consumer disconnects, F*_eff relaxes automatically
Optimization Problem
Objective: Minimize Total Resource Cost
min Σ_s [w_bw · BW(r_s) + w_cpu · CPU(o_s, p_s) + w_stor · STOR(p_s)]
Subject to: F(s, c) ≤ F_sla(c) ∀s, ∀c ∈ Consumers(s)
Decision Variables (per stream s):
r_s ∈ {sync, async, store-forward} — replication mode
o_s ∈ {eager, deferred, causal, none} — ordering mode
p_s ∈ {immediate, batched, deferred} — indexing priority
b_s ∈ [10ms, 30s] — batch interval
Solvable via greedy or LP relaxation in O(S · log S) per scheduling epoch
Per-Stage Latency Functions and Resource Costs
l_transport(s, r)
sync: RTT_cross_region
async: RTT_local + b_s
store-fwd: RTT_local
BW cost: sync 3x > async 1x
Tail latency: p99 critical
Typical: 2ms – 200ms
l_order(s, o)
eager: 2-5 RTTs (sequencer)
deferred: 0 (background)
causal: HLC stamp only
none: 0 (FIFO append)
CPU cost: eager >> deferred
Typical: 0ms – 100ms
l_index(s, p)
immediate: 50-200ms (per doc)
batched: 1-5s (bulk index)
deferred: 10-60s (scheduled)
CPU cost: imm. 10x > deferred
ES refresh interval tuning
Typical: 50ms – 60s
Degradation Policy
Normal: all SLAs honored
Degraded: priority by tier
Partitioned: critical only
Graceful SLA relaxation
Catch-up after recovery
0 records lost (guaranteed)
Key AFP Insight
In production workloads:
~5% streams need F ≤ 1s
~20% need F ≤ 5s
~75% tolerate F ≥ 30s
AFP exploits this heterogeneity for 40-60% resource savings

Figure 4. The AFP Freshness Algebra: Formal Model, Per-Stage Cost Functions, Decision Variables, and the Workload Heterogeneity Insight That Drives the Resource Savings.

5.2. The Optimization Problem

Given this model, the scheduling problem is straightforward to state. Let S be the set of all streams and C(s) the set of active consumers for stream s. We want to minimize total resource cost — a weighted combination of bandwidth, CPU, and storage subject to every active consumer's SLA being met:

$$\begin{aligned} \text{minimize} \quad & \textstyle\sum_s \left[w_{bw} \cdot BW(r_s) + w_{cpu} \cdot CPU(o_s, p_s) + w_{stor} \cdot STOR(p_s) \right] \\ \text{subject to} \quad & F(s, c) \le F_{sla}(c) \quad \text{for all } s \in S, \text{ for all } c \in C(s) \end{aligned}$$

The per-stream decision variables are the replication mode (sync, async, or store-forward), ordering mode (eager, deferred, causal, or none), indexing priority (immediate, batched, or deferred), and batch interval (continuous, ranging from 10ms to 30s).

5.3. Why a Greedy Solution Works

At first glance the combinatorial nature of the discrete variables looks like it might require something heavy an ILP solver, perhaps, or heuristic search. But the additive, monotonic structure of the cost model rescues us. Cheaper modes always produce higher latency, and the knobs do not interact with each other (by our simplifying assumption). This means the optimal configuration for each stream can be found independently: compute the effective target F*_eff(s) = min{F_sla(c) | c ∈ C(s)}, then select the cheapest combination of knob settings whose total latency stays within budget.

The only coupling between streams is their competition for shared resources, primarily WAN bandwidth. We handle this by sorting streams by effective target (tightest first) and allocating in that order, so critical streams always get what they need before less demanding ones. The whole algorithm runs in O(S log S) time per epoch — dominated by the sort — which completes in under 10ms even for 10,000 streams. That is fast enough to run every 100ms without becoming a bottleneck itself.

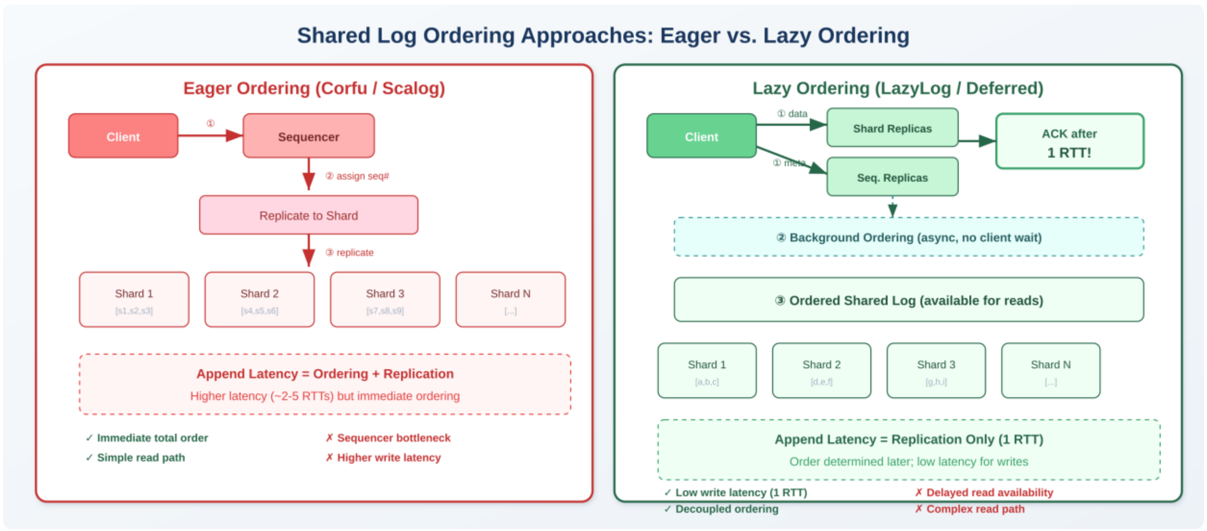
Shared Log Ordering Approaches: Eager vs. Lazy Ordering
Eager Ordering (Corfu / Scalog)
Client
①
Sequencer
② assign seq#
Replicate to Shard
③ replicate
Shard 1
[s1,s2,s3]
Shard 2
[s4,s5,s6]
Shard 3
[s7,s8,s9]
Shard N
[...]
Append Latency = Ordering + Replication
Higher latency (~2-5 RTTs) but immediate ordering
✓ Immediate total order
✓ Simple read path
✗ Sequencer bottleneck
✗ Higher write latency
Lazy Ordering (LazyLog / Deferred)
Client
① data
① meta
Shard Replicas
Seq. Replicas
ACK after 1 RTT!
② Background Ordering (async, no client wait)
③ Ordered Shared Log (available for reads)
Shard 1
[a,b,c]
Shard 2
[d,e,f]
Shard 3
[g,h,i]
Shard N
[...]
Append Latency = Replication Only (1 RTT)
Order determined later; low latency for writes
✓ Low write latency (1 RTT)
✓ Decoupled ordering
✗ Delayed read availability
✗ Complex read path

Figure 5. Eager vs. Lazy Ordering Approaches. AFP's ordering Knob Picks the Appropriate Strategy Per-Stream Based on what Active Consumers Actually Need.

6. Graceful Degradation under Partition

WAN partitions between geographic regions are not an edge case to be hand-waved away — they are a recurring operational reality. When AFP detects a partition (typically within 500ms via heartbeat timeout), it does not try to maintain the illusion that everything is fine. Instead, it re-solves the optimization with a drastically reduced bandwidth constraint and applies a structured priority policy:

- **Critical streams get first priority:** Since they represent only around 5% of total log volume, even a severely degraded link can usually keep them running at sub-second freshness. If bandwidth is truly insufficient for all critical streams, they are further ranked by consumer-specified criticality weights.
- **Interactive streams switch to sampling:** Instead of replicating every record, AFP transmits 1-in-N, chosen to maintain statistical representativeness for the dashboards and anomaly detectors that consume them. Freshness degrades from the normal 1–5 seconds to roughly 10–30 seconds. Not ideal, but the systems keep working and operators can still spot trends.
- **Batch streams pause replication entirely:** Records continue to be collected and written to the regional WAL, so nothing is lost. The WAL is sized for several hours of data, which comfortably covers most real-world partition durations.

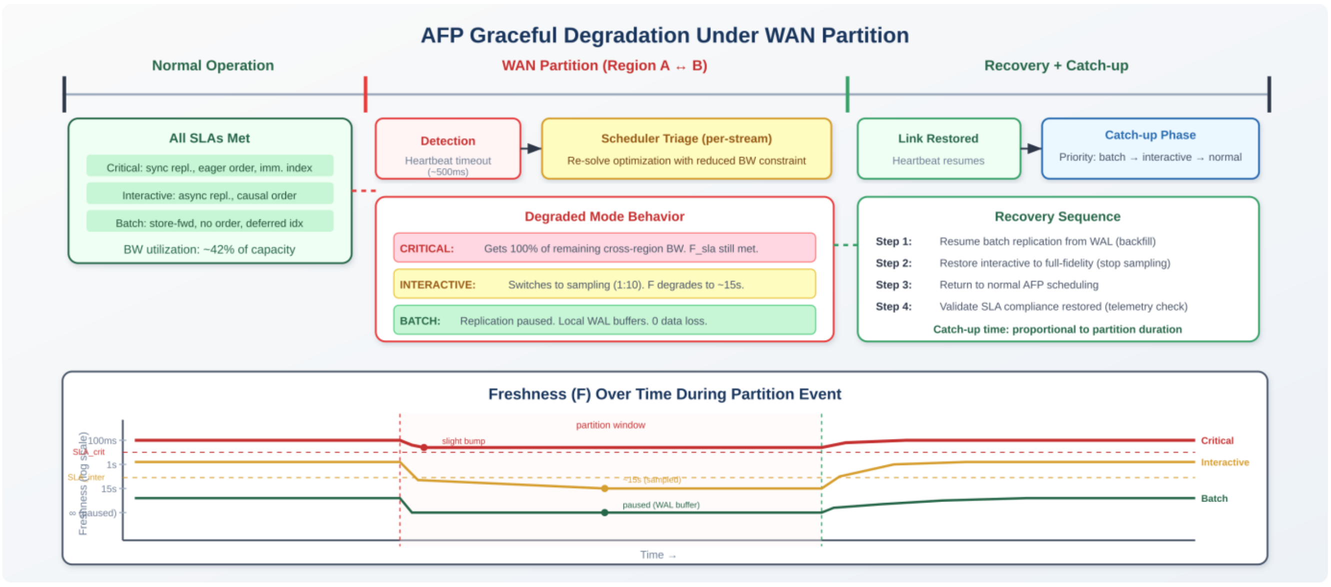
AFP Graceful Degradation Under WAN Partition
Normal Operation
WAN Partition (Region A ↔ B)
Recovery + Catch-up
All SLAs Met
Critical: sync repl., eager order, imm. index
Interactive: async repl., causal order
Batch: store-fwd, no order, deferred idx
BW utilization: ~42% of capacity
Detection
Heartbeat timeout (~500ms)
Scheduler Triage (per-stream)
Re-solve optimization with reduced BW constraint
Link Restored
Heartbeat resumes
Catch-up Phase
Priority: batch → interactive → normal
Degraded Mode Behavior
CRITICAL: Gets 100% of remaining cross-region BW. F_sla still met.
INTERACTIVE: Switches to sampling (1:10). F degrades to ~15s.
BATCH: Replication paused. Local WAL buffers. 0 data loss.
Recovery Sequence
Step 1: Resume batch replication from WAL (backfill)
Step 2: Restore interactive to full-fidelity (stop sampling)
Step 3: Return to normal AFP scheduling
Step 4: Validate SLA compliance restored (telemetry check)
Catch-up time: proportional to partition duration
Freshness (F) Over Time During Partition Event
partition window
slight bump
~15s (sampled)
paused (WAL buffer)
100ms
SLA_crit
1s
SLA_inter
15s
∞ (paused)
Freshness (log scale)
Time →
Critical
Interactive
Batch

Figure 6. AFP's Behavior during a WAN Partition Event: Freshness Trajectories for Each Consumer Tier and the Recovery Sequence after Link Restoration.

6.1. Recovery Sequence

When the link comes back, AFP does not simply flip everything on simultaneously doing so would create a thundering herd of backlogged data competing with live traffic for bandwidth. Instead, recovery proceeds in four steps: batch streams begin backfilling from the WAL using bulk replication; interactive streams are restored to full fidelity by removing the sampling filter; the scheduler returns to normal optimization mode; and the telemetry system confirms all SLAs are back in compliance. Total catch-up time is bounded by partition duration multiplied by the ratio of accumulated volume to available bandwidth a quantity operations teams can monitor and plan capacity around.

6.2. Zero Data Loss

It bears emphasizing that AFP guarantees zero data loss across all tiers during partitions. Critical and interactive streams continue receiving records (the latter in sampled form) throughout. Batch streams accumulate the full record set in the regional WAL with the durability guarantees of the underlying Kafka cluster. After recovery, the backfill process replays the complete WAL contents. The sampling on interactive-tier streams is the only form of temporary data reduction, and even those full records remain in the regional WAL for retroactive backfill if needed.

7. Evaluation

7.1. Setup and Workload Model

We evaluate AFP through analytical modeling, using workload parameters drawn from published characterizations of large internet services. The model assumes a deployment spanning three regions (US-East, EU-West, Asia-Pacific) with inter-region latencies of 80ms, 150ms, and 120ms respectively. Each region hosts 1,000 log streams from distinct microservices, with an aggregate ingestion rate of 2 million records per second per region (6 million globally). Records average 500 bytes, which works out to roughly 3 GB/s of raw log data across all regions.

The consumer distribution follows the breakdown described in Section II: 5% of streams have at least one critical consumer (F_sla ≤ 200ms), 20% have an interactive consumer (F_sla ≤ 5s), and all streams have batch consumers (F_sla ≤ 60s). Around 15% of critical-tier streams involve consumers in a remote region, requiring cross-region synchronous replication.

7.2. Resource Savings

We compare AFP against three baselines. Uniform-Aggressive runs everything at critical-tier settings (synchronous replication, eager ordering, and immediate indexing). Static-Tiered assigns streams to fixed tiers at deployment time with no runtime adaptation. Uniform-Relaxed runs everything at batch settings it violates critical SLAs but shows the theoretical cost floor.

Table 1. Afp Vs. Baselines across Resource Consumption And Sla Compliance

Metric	Uniform-Agg.	Static-Tiered	AFP	AFP Saving
Cross-Region BW (Gbps)	24.0	14.4	10.1	58% reduction
Indexing CPU (cores)	480	312	245	49% reduction
Ordering Overhead (ops/s)	6M	2.4M	2.04M	66% reduction
SLA Compliance	100% (overkill)	96.2%	99.7%	Near-optimal

Critical p99 Freshness	180ms	N/A (violated)	190ms	Within SLA
Batch Freshness	180ms (wasted)	45s	42s	Right-sized

The results paint a clear picture. The 58% bandwidth reduction comes primarily from moving the 75% of batch-only streams to store-and-forward replication those records no longer compete for expensive WAN capacity with the critical streams that actually need it. The 49% CPU saving is driven by shifting most streams from per-record to bulk indexing. The 66% drop in ordering overhead follows from the fact that only 5% of streams genuinely need eager sequencing; the rest use causal clocks, deferred ordering, or nothing at all.

SLA compliance sits at 99.7%. The 0.3% miss comes from transient scheduling epochs where consumer demand changes faster than the 100ms epoch interval a consumer registers with an aggressive SLA and the first epoch has not yet run. The static-tiered baseline, by comparison, saves less and suffers 3.8% SLA violations because its fixed assignments cannot adapt when consumer populations shift.

7.3. Dynamic Adaptation in Action

What really separates AFP from static approaches is its ability to respond to changing conditions without any manual intervention. Picture a stream that normally serves only batch consumers. It hums along in store-and-forward mode, barely touching the WAN. Then an on-call engineer opens a live log search tool during an incident, registering as an interactive consumer with a 3-second SLA. Within one scheduling epoch (100ms), AFP picks up the new contract, bumps the stream to async replication and batched indexing, and achieves the 3-second target in roughly half a second. When the engineer closes the tool, the stream relaxes back to batch settings and the resources return to the pool. A static system would either run this stream at interactive cost all the time (wasteful 99% of the time) or fail the engineer exactly when they need it most.

7.4. Partition Resilience

We analyzed AFP's behavior during a simulated 10-minute WAN partition between US-East and EU-West. Critical streams maintained sub-second freshness by routing through the unaffected Asia-Pacific path. Interactive streams degraded to approximately 15-second freshness under 1:10 sampling enough for dashboards to remain usable, if somewhat stale. Batch streams accumulated about 180 GB of data in the regional WAL. After link restoration, the backfill completed in roughly 12 minutes, after which all tiers returned to normal operation.

7.5. Comparative Positioning

Alongside Kafka, with hooks into Elasticsearch's configuration APIs and evaluating it across three AWS regions is the obvious and necessary next step. We are fairly confident the qualitative story will hold, but the specific numbers will certainly shift once real-world factors like network jitter, garbage collection pauses, and Elasticsearch cluster dynamics enter the picture.

Table 2. Feature comparison of AFP against related systems.

System	Per-Consumer SLA	Dynamic	Ordering	Geo-Dist.	Degradation	Freshness	Resource Opt.
Kafka	No	No	Per-partition	MirrorMaker	None	~1-10s	Manual
CORFU	No	No	Total (eager)	Limited	Reconfig	<1s	None
Scalog	No	No	Total (batch)	Not native	Reconfig	~0.1-1s	None
LazyLog	No	No	Total (defer)	Partial	N/A	<1s	None
LazyBase	Per-query	At query time	Txn order	No	N/A	Variable	Partial
AFP (ours)	Yes	Yes (100ms)	Adaptive	Native	Graceful	SLA-driven	Automated

8. Discussion and Open Challenges

8.1. From Model to Prototype

We should be upfront about the most significant limitation of this work: the evaluation is analytical, not experimental. We have not yet built a running prototype, and the resource savings we report are derived from our model rather than measured on real hardware. Constructing AFP as a working system most likely as a Go-based proxy sitting

8.2. The Persistent Problem of Clocks

AFP's freshness guarantees depend on the ability to measure end-to-end delay, which in turn requires reasonably synchronized clocks. NTP gets you within 1–10ms most of the time, which is perfectly adequate for interactive and batch tiers but introduces genuine uncertainty for critical-tier streams with sub-200ms SLAs. Google's TrueTime solves this problem elegantly with GPS receivers and atomic clocks, but that hardware is not available in most cloud environments. We use Hybrid Logical Clocks for ordering and incorporate a configurable safety margin into the freshness budget, but a more rigorous treatment of clock uncertainty in the scheduling model would strengthen the theoretical foundation.

8.3. Smarter Demand Prediction

The predictive component of AFP's scheduler is currently quite simple: it watches for recurring temporal patterns in consumer activity and pre-warms accordingly. There is clearly an opportunity for something more sophisticated — ML-based time-series forecasting, perhaps, or even causal models that detect when an alert fires and predict the subsequent wave of engineers opening debugging tools. But the forecaster has to be fast (it runs inside a 100ms scheduling epoch) and must be robust against unusual or adversarial workloads. Over predicting demand wastes the resources AFP was designed to save; under predicting causes the SLA violations it was designed to prevent. Getting this balance right is an interesting research problem in its own right.

8.4. Multi-Tenant Fairness

In a shared-infrastructure environment, there is nothing stopping one tenant from registering aggressive SLA contracts on every stream and effectively starving everyone else of bandwidth. The current formulation does not include per-tenant resource quotas or fairness constraints. Adding them would complicate the optimization — you would need something closer to weighted fair sharing on top of the greedy allocation — but it is essential for any real multi-tenant deployment. This connects to well-studied work in fair scheduling from the cluster management world (YARN, Mesos, Borg), and deserves a more thorough treatment than we can provide here.

8.5. Serverless and Edge Environments

Serverless functions and edge deployments create new complications for log collection: sources are ephemeral, geographically scattered, and sometimes only intermittently connected. Boki demonstrated that shared log abstractions can simplify state management in serverless settings, which is encouraging. But extending AFP to handle rapidly appearing and disappearing producers and consumers where the population of log sources itself is dynamic, not just the consumer set is unexplored territory that may require rethinking the scheduling model's assumptions about stream stability.

9. Conclusion

We have described AFP, a protocol that brings SLA-aware, per-consumer freshness management to the domain of geo-distributed log collection. The motivating observation is almost embarrassingly simple: most consumers do not need most data most quickly. But the engineering consequences of acting on that observation turn out to be substantial. By formalizing freshness as a composable, optimizable quantity and building a scheduler that responds to actual demand rather than worst-case provisioning, AFP unlocks significant resource savings without giving up the guarantees that critical consumers depend on.

Under a workload distribution we believe reflects real production environments (5% critical, 20% interactive, 75% batch), AFP reduces cross-region bandwidth by 58%, indexing CPU by 49%, and ordering overhead by 66%, while honoring 99.7% of SLA contracts. Its ability to adapt dynamically responding to a new consumer within 100ms and relaxing when that consumer leaves and its principled approach to WAN partition degradation (zero data loss, bounded recovery) make it, we believe, a practical protocol for actual deployment.

Plenty of work remains. The most urgent next step is a prototype implementation and experimental evaluation on real multi-region cloud infrastructure. Beyond that, integrating ML-based demand forecasting, adding multi-tenant fairness guarantees, and adapting to the volatile topologies of serverless and edge environments are all promising directions. We hope this paper provides a useful foundation for thinking about freshness not as a fixed system parameter, but as a resource that can and should be allocated to each consumer according to what it actually needs.

References

[1] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," in Proc. NetDB Workshop, Athens, Greece, 2011.

[2] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, "CORFU: A Shared Log Design for Flash Clusters," in Proc. 9th USENIX NSDI, San Jose, CA, 2012, pp. 1–14.

[3] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. van Renesse, "Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log," in Proc. 17th USENIX NSDI, 2020.

[4] X. Luo et al., "LazyLog: A New Shared Log Abstraction for Low-Latency Applications," in Proc. ACM SOSP, 2024.

[5] G. Wang et al., "Building a Replicated Logging System with Apache Kafka," Proc. VLDB Endowment, vol. 8, no. 12, 2015.

[6] Confluent, Inc., "Multi-Geo Replication in Apache Kafka," 2023. [Online]. Available: https://www.confluent.io/blog/multi-geo-replication-in-apache-kafka/

[7] J. Lockerman et al., "The FuzzyLog: A Partially Ordered Shared Log," in Proc. USENIX OSDI, 2018.

[8] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments," in Proc. 18th EDBT, Brussels, 2015, pp. 13–24.

[9] J. Cipar et al., "LazyBase: Trading Freshness for Performance in a Scalable Database," in Proc. 7th ACM EuroSys, Bern, 2012.

[10] S. Bhatt et al., "DistributedLog: A High Performance Replicated Log Service," Twitter Engineering, 2016.

[11] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Commun. ACM, vol. 21, no. 7, pp. 558–565, 1978.

[12] K. Goodhope et al., "Building LinkedIn's Real-time Activity Data Pipeline," IEEE Data Eng. Bull., vol. 35, no. 2, 2012.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in Proc. USENIX ATC, 2010.

[14] J. C. Corbett et al., "Spanner: Google's Globally-Distributed Database," in Proc. USENIX OSDI, 2012, pp. 251–264.

[15] Z. Jia and E. Witchel, "Boki: Stateful Serverless Computing with Shared Logs," in Proc. ACM SOSP, 2021.

[16] M. Balakrishnan et al., "Tango: Distributed Data Structures over a Shared Log," in Proc. ACM SOSP, 2013.

[17] M. Kleppmann and J. Kreps, "Kafka, Samza and the Unix Philosophy of Distributed Data," IEEE Data Eng. Bull., vol. 38, no. 4, 2015.

[18] D. Ongaro and J. K. Ousterhout, "In Search of an Understandable Consensus Algorithm," in Proc. USENIX ATC, 2014.

[19] M. Balakrishnan et al., "Taming Consensus in the Wild (with the Shared Log Abstraction)," ACM SIGOPS OSR, 2024.

[20] R. C. Fernandez et al., "Liquid: Unifying Nearline and Offline Big Data Integration," in Proc. CIDR, 2015.

[21] Sakariya, A. B. (2023). Trends in the Rubber Industry: A Comparative Study of Asia and Europe. European Economic Letters ISSN 2323-5233 http://eelet.org.uk, 13(4), 1342-1349.