CLASSICALLIBRARY : Pages:1–20

Original Research

Multi-Region Replication for Feature Pipelines with Latency-Aware Consistency and Conflict-Free Resolution

Ramesh Bahadur Khadka[1] and Sudarshan Pratap Koirala[2]

[1]Department of Information Technology, Far Western University, Mahendranagar–Bhasi Road, Kanchanpur 10400, Nepal.
[2]Department of Computer Applications, Madan Bhandari Memorial College, Bhaktapur–Tokha Road, Kathmandu 44600, Nepal.

Abstract

Modern feature pipelines increasingly operate across multiple cloud regions to reduce tail latency for online inference, satisfy data residency constraints, and improve availability under regional faults. Replicating feature state across regions is straightforward when features are immutable or when a single writer exists, but production feature stores routinely ingest concurrent updates from streaming joins, late-arriving events, backfills, and on-device signals that can originate in different regions. These realities create a tension between low-latency reads, bounded staleness, and deterministic conflict handling when updates race or arrive out of order. This paper studies multi-region replication for feature pipelines under latency-aware consistency objectives and conflict-free resolution requirements. The central idea is to treat feature values as mergeable state with explicit semantics and to couple replication policies to end-to-end inference service-level objectives rather than to a single global consistency level. We define a system model that connects feature freshness to model error through sensitivity estimates, enabling the system to allocate consistency and bandwidth budgets where they matter most. A latency-aware controller selects replication and read strategies per feature and per workload slice, while conflict-free resolution uses merge operators that preserve feature meaning across counters, sets, time-series aggregates, and embedding-like vectors. The design integrates approximate sketches and compression for cost control and exposes reproducible evaluation methodology based on workload traces and fault injection. The result is an architecture that can deliver predictable tail latency while keeping inconsistency-induced feature drift measurable, bounded, and operationally auditable.

1. Introduction

Online learning systems depend on feature pipelines that transform raw events into model-ready representations for inference and training [1]. In the common split architecture, streaming processors and batch jobs compute features, a feature store materializes them, and inference services read them at low latency. As deployment footprints extend to multiple regions, the feature store becomes a distributed, replicated database whose semantics must match the meaning of features rather than the semantics of generic key-value registers. A feature value often represents an aggregate over a time window, a deduplicated set of items, a counter with monotonicity expectations, or an embedding-like vector updated by incremental learning. These values are updated by heterogeneous writers, including stream processors in different regions, backfill jobs that replay historical data, and user-facing services that attach real-time signals. The system must tolerate network partitions, variable cross-region latency, and partial failures while maintaining the operational invariants required for reliable inference [2].

The simplest approach is to impose a single global consistency level, such as linearizability via a consensus group per key, or eventual consistency via asynchronous replication. The former typically incurs high tail latency and fragile availability under cross-region faults, while the latter can yield arbitrarily stale reads and nondeterministic outcomes when concurrent updates conflict. Feature pipelines are particularly sensitive to these trade-offs because the cost of inconsistency is not uniform across features. Some features are slow-changing and have limited influence on model outputs, while others capture

rapidly varying user intent and strongly affect predictions. Similarly, the latency budget at inference time is constrained by user experience, and the budget is dominated by p99 or p999 behavior rather than average latency [3]. A multi-region feature store must therefore mediate a three-way trade-off among read latency, freshness, and conflict semantics, with explicit knobs that map to model behavior.

Topology	Description	Typical regions	Primary use case
Single-region	All feature pipelines and online stores deployed in a single cluster.	1	Development, low-traffic tenants
Active–passive	Writes in primary region; async replication to warm standby.	2	Disaster recovery, cost-sensitive workloads
Active–active (sync)	Coordinated writes across regions with quorum replication.	2–3	High-value, low-latency-critical features
Active–active (async)	Independent writes with background reconciliation.	3–5	Global products with tolerant consistency needs
Hub-and-spoke	Central hub for aggregation, regional spokes for low-latency reads.	3+	Hybrid online/offline feature pipelines

Table 1: Deployment topologies for multi-region feature pipelines.

Mode	p95 read latency (ms)	Staleness budget	Guarantees
Strong global	80–150	≤ 5 ms	Linearizable reads across all regions
Bounded staleness	40–80	100 ms	Monotonic reads with time-bound lag
Session	30–60	250 ms	Read-your-writes within a client session
Eventual	20–40	2 s	Convergence without ordering guarantees
Latency-aware hybrid	25–70	Adaptive	Per-request trade-off based on SLO and region distance

Table 2: Consistency modes and their latency–staleness trade-offs.

This paper proposes a latency-aware replication and consistency framework for feature pipelines with conflict-free resolution grounded in feature semantics. The approach begins by modeling features as typed state with merge operators that are associative, commutative, and idempotent when possible, enabling convergence without coordination. When idempotent merges are not semantically correct, the system uses deterministic resolution that reduces to a merge on an augmented state space, such as including causality metadata, monotone sufficient statistics, or bounded-history summaries. Consistency is not expressed as a single global guarantee but as per-feature policies that can vary across entities, time, and

Feature class	Example features	Tolerance to staleness	Replication strategy
Real-time personalization	Clickstream embeddings, recency counters	< 100 ms	Strong or bounded-staleness, active–active
Risk and fraud	Velocity checks, device fingerprints	< 500 ms	Strong for writes, session for reads
Aggregated analytics	7-day aggregates, cohort statistics	Minutes	Async bulk replication, hub-and-spoke
Experiments and flags	Treatment assignment, rollout flags	1–2 s	Session consistency, cache with fast invalidations
Offline-derived features	Batch-trained embeddings, segment labels	Hours	Daily checkpoint snapshots with lazy backfill

Table 3: Feature categories and their replication requirements.

Conflict type	Detection signal	Resolution strategy	Notes
Concurrent counter updates	Divergent increments in multiple regions	CRDT grow-only or PN-counter merge	Preserves total count without coordination
Last-writer-wins field	Conflicting timestamps for mutable field	LWW register with clock skew guardrails	Drops stale updates beyond skew threshold
Set membership	Adds/removes from collections	OR-Set CRDT merge	Avoids resurrecting removed elements
Idempotent upserts	Duplicate writes with same idempotency key	Deduplication via operation log	Guarantees exactly-once semantics per key
Schema evolution	Mismatched feature schemas	Versioned schema registry with unions	Enables rolling upgrades across regions

Table 4: Conflict types and associated resolution strategies.

workload [**?**]. Policies are selected by a controller that observes latency distributions, replication lag, update conflict rates, and sensitivity of model outputs to feature perturbations. The controller enforces constraints through multi-objective optimization with explicit latency, bandwidth, and energy budgets.

A key requirement is that the system must support both training and inference. Training workflows often require time travel, reproducibility, and snapshot isolation over feature values aligned with event time. Inference requires fast reads, often at high QPS, and can tolerate bounded staleness if the induced error is controlled [4]. Multi-region replication adds complexity because training may run centrally while inference is distributed, and because backfills can introduce late updates that race with streaming updates. A conflict-free, semantics-aware representation helps unify these workloads: it permits convergence across regions and simplifies replay and reconstruction, while still supporting fast materialized reads.

The rest of the paper develops the model, algorithms, and systems mechanisms needed for this design. We first define a replication architecture for multi-region feature pipelines, including metadata for causality and feature typing. We then introduce latency-aware consistency control that selects read

Region	Traffic pattern	Peak QPS	Daily feature ingest (TB)
us-east	70% online / 30% offline	150k	18.2
us-west	60% online / 40% offline	95k	11.4
eu-central	80% online / 20% offline	110k	13.0
ap-southeast	65% online / 35% offline	75k	9.1
sa-east	50% online / 50% offline	40k	4.7

Table 5: Per-region workload characteristics for the evaluated deployment.

Channel	Payload encoding	Compression	Target one-way latency (ms)
Control plane RPC	JSON over gRPC	None	< 200
Metadata stream	Protobuf	LZ4	< 300
Feature deltas	Columnar binary	Zstd (level 3)	< 150
Checkpoint snapshots	Parquet files	Snappy	< 1000
Backfill replays	Protobuf	Zstd (level 5)	Best effort

Table 6: Replication transport configuration for different data channels.

Variant	p95 end-to-end latency (ms)	Staleness violations (ppm)
Baseline (no routing)	112	940
Latency-only routing	78	510
Staleness-only routing	101	120
Latency-aware hybrid	84	65
Hybrid + local caching	63	72

Table 7: Ablation of latency-aware routing strategies for online inference traffic.

System	p95 latency (ms)	Cross-region egress (GB/h)
Local-only features	41	0
Naïve multi-master	124	215
Global database	97	168
Proposed replication layer	58	89
Proposed + caching tier	46	73

Table 8: Comparison against baseline systems on latency and cross-region traffic.

and replication strategies under tail-latency constraints [5]. Next we define conflict-free resolution mechanisms for common feature classes, including numeric aggregates, categorical sets, time-decayed signals, and embedding-like vectors. We provide optimization formulations and complexity analysis, including NP-hardness of certain placement and quorum selection problems and practical heuristics. Finally, we describe performance engineering, storage internals, distributed execution mechanics, and an evaluation and reproducibility plan that enables auditing of staleness, conflicts, and model impact without relying on ad hoc assumptions.

Scenario	Impact on reads	Recovery strategy	Median recovery time (s)
Single-region outage	Increased cross-region latency	Failover to nearest healthy region	45
Inter-region link degradation	Higher tail latency, occasional timeouts	Latency-aware rerouting with throttling	30
Metadata store partition	Inconsistent feature schemas	Read-only mode for affected tenants	60
Backfill job failure	Missing historical features	Incremental replays from last checkpoint	180
Hot partition overload	Elevated error rate for specific keys	Dynamic key resharding and rate limiting	25

Table 9: Failure scenarios and observed recovery behavior in the replicated system.

2. System Model and Replication Architecture

We consider a set of regions $\mathcal{R} = \{1, \dots, R\}$ connected by a wide-area network with time-varying delays and occasional partitions. Entities are keyed by $k \in \mathcal{K}$, and features are indexed by $f \in \mathcal{F}$. A feature store maintains, for each k, f, a typed state $S_{k,f}$ that evolves under updates. Updates originate from writers in multiple regions; each update is a function application on state, written as $S \leftarrow \phi u, S$ where u includes payload and metadata such as event time, writer identity, and a causality marker. Readers are inference services in each region that request feature vectors $x_k = v_{k,f\, f \in \mathcal{F}_q}$ for a query-specific subset $\mathcal{F}_q \subseteq \mathcal{F}$. The query path includes network hops to one or more feature store replicas, optional compute for on-the-fly transformations, and cache layers [6]. The system target is to minimize end-to-end inference latency while bounding the inconsistency of the returned feature vector relative to a well-defined reference state.

Replication is modeled as dissemination of update deltas or merged state among regions over a directed overlay graph $G = \mathcal{R}, E$. Each edge $i, j \in E$ has an associated random variable for transmission delay D_{ij} and an effective bandwidth budget C_{ij}. The overlay may be full mesh, hub-and-spoke, or dynamically chosen based on cost and observed performance. Updates are appended to a per-partition log, where partitions are defined by consistent hashing of keys and possibly by feature families. Each partition has a replication group spanning multiple regions. Unlike classic database replication that treats values as opaque bytes, our model requires that each feature value has an associated merge semantics [7]. Therefore, the replica state is not only a materialized map $M : k, f \mapsto S_{k,f}$, but also a metadata store H that records causality markers, version vectors, or hybrid logical clocks sufficient to support deterministic merging and bounded staleness estimation.

A crucial aspect is the connection between feature inconsistency and model error. Let the inference model be a function $g_\theta : \mathbb{R}^{|\mathcal{F}_q|} \to \mathbb{R}^m$ with parameters θ, producing scores or probabilities. If a reader obtains an inconsistent feature vector $\tilde{x}_k$ rather than a reference $x_k^\star$, the prediction difference can be bounded using local sensitivity. For differentiable g_θ, a first-order approximation yields

$$g_\theta \tilde{x}_k - g_\theta x_k^\star \approx J_\theta x_k^\star \, \tilde{x}_k - x_k^\star, \tag{2.1}$$

where J_θ is the Jacobian. For scalar outputs, one may summarize sensitivity by per-feature weights $w_f k$ derived from gradients or from learned attribution models. This motivates an inconsistency cost defined as

$$\mathcal{E} k, \tilde{x}_k, x_k^\star = {}_{f \in \mathcal{F}_q} \, w_f k \, \rho_f \left(\tilde{v}_{k,f}, v_{k,f}^\star \right), \tag{2.2}$$

where ρ_f is a feature-specific discrepancy metric [8]. For numeric features, $\rho_f a, b = |a - b|$ or $a - b^2$; for categorical sets, ρ_f may be a Jaccard distance; for embeddings, $\rho_f a, b = 1 - \frac{\langle a,b \rangle}{\|a\|\|b\|}$ or $\|a-b\|_2$. The system goal is to keep $\mathbb{E}\mathcal{E}$ below a threshold while also meeting latency SLOs. Importantly, the system does not need the exact Jacobian at runtime; it can use coarse sensitivity estimates, quantiles, or feature importance proxies computed offline, then refreshed periodically.

The reference state $x_k^\star$ is defined relative to an event-time aligned, globally merged state at a chosen watermark. Because global instantaneous truth is often ill-defined under partitions and late events, we define $x_k^\star t, \delta$ as the state obtained by incorporating all updates with event time at most $t - \delta$ and with deduplication semantics applied, where δ is a safety margin. This aligns with common streaming practice where results are considered final after a lateness bound. Under this definition, a read at wall-clock time t is allowed to be stale up to δ in event time, and the system can quantify and budget that staleness [9]. For inference, δ might be small but nonzero; for training data generation, δ may be larger to ensure determinism.

The architecture separates three planes: an update plane that ingests events and applies feature-specific update functions, a replication plane that disseminates deltas and merges states across regions, and a read plane that serves feature vectors with optional consistency constraints. In the update plane, each writer region applies updates locally to minimize write latency and produce a local replica state. The replication plane then ships either deltas, compressed sufficient statistics, or periodic snapshots. The read plane can use local state, consult remote replicas, or require quorum responses depending on the selected consistency mode [10]. A metadata service maintains per-partition summaries of replication lag, conflict frequency, and causal frontier information, enabling the read plane to decide whether local reads satisfy the requested freshness constraints.

Storage is organized around a log-structured representation to support high write throughput from streams and backfills. Each partition maintains an append-only update log with sequence numbers, and a materialized state store. The state store may be an LSM-tree keyed by k, f plus a version dimension. To support merging and conflict-free semantics, the stored value is typically a structured state that includes both the feature payload and merge metadata, such as per-writer components for counters, bounded histories for time-series, or vector clocks [11]. While this increases storage overhead, it enables convergence without coordination and permits fast incremental compaction that folds deltas into a canonical form.

The replication plane uses anti-entropy protocols to ensure eventual dissemination. In practice, replication can be a combination of push-based streaming of new log segments, pull-based repair when gaps are detected, and gossip of causal frontiers. The causal frontier can be summarized by a vector $c_i \in \mathbb{N}^R$ at region i, where $c_i r$ indicates the highest sequence number from region r that has been incorporated. Vector clocks provide precise causality but can be large when R is large; hybrid logical clocks or dotted version vectors can reduce overhead. The system can also maintain per-partition scalar watermarks for common cases, accepting that this reduces precision but may suffice when conflicts are rare or when deterministic merge rules handle concurrent updates safely [12].

3. Latency-Aware Consistency Control

Traditional consistency levels are described qualitatively, but latency-aware feature serving requires quantitative control that explicitly accounts for tail latency, network variability, and the model impact of staleness. We define a read policy π that maps a query context to a set of replica contacts and acceptance conditions. The context includes region i, query feature subset $\mathcal{F}_q$, entity key k, and a latency budget $L_{\max}$ derived from the inference service. A policy returns a plan specifying whether to read local only, read local with fallback, read from the nearest among a subset of regions, or perform a quorum read requiring responses from multiple regions. It also specifies a freshness constraint expressed in terms of a causal frontier or watermark.

Let $T_{i \to j}$ be the random variable for round-trip time from region i to replica j including serialization and queuing. A read plan that contacts a set $A \subseteq \mathcal{R}$ and waits for the q-th fastest response has latency

approximately equal to the q-th order statistic of $\{T_{i\to j} : j \in A\}$ plus local processing. The expected tail latency at percentile α can be modeled by distributional assumptions or empirically estimated [13]. For policy selection, it is useful to work with a conservative bound:

$$\mathrm{p}_\alpha\left(\max_{j\in B} T_{i\to j}\right) \le \max_{j\in B} \mathrm{p}_\alpha T_{i\to j}, \tag{3.1}$$

where B is the set of replicas whose responses are required. Although loose, this bound supports safe admission control. The plan must satisfy $\mathrm{p}_\alpha$latency $\le L_{\max}$ for a chosen α such as 99%.

Freshness is captured by a staleness measure. Let $F_i k, f, t$ be the causal frontier of region i for key-feature pair k, f at wall-clock time t. For vector-clock-like metadata, F_i can be a set of incorporated update identifiers; for watermark-like metadata, it can be a scalar event-time watermark [14]. Define the reference frontier $F^\star k, f, t$ as the frontier that would exist in a hypothetical globally merged state under the chosen lateness bound. Staleness can be quantified as the missing mass of updates:

$$\Delta_i k, f, t = |F^\star k, f, t \setminus F_i k, f, t|, \tag{3.2}$$

or, for event-time watermarks, as $\Delta_i = \max 0, W^\star - W_i$ where W is a watermark. A read is considered acceptable under a bounded-staleness constraint τ if $\Delta_i \le \tau$. The challenge is that determining Δ_i exactly may require global knowledge [15]. Therefore, the system uses estimators based on replication lag statistics, per-region frontiers, and probabilistic bounds derived from observed delay distributions.

We define a probabilistic freshness guarantee of the form $\Pr \Delta_i k, f, t \le \tau_f \ge 1 - \epsilon_f$, where τ_f and ϵ_f can vary per feature based on sensitivity. A latency-aware controller chooses τ_f and the read plan to satisfy both latency and freshness constraints. When the local replica likely violates freshness, the plan may contact an additional replica that is known to be ahead in the relevant causal dimension. The controller can maintain, for each partition and region, a predicted lag distribution $\widehat{L}_i^p$ and choose contacts accordingly.

To connect these decisions to model impact, consider a per-feature staleness-to-error curve $e_f \delta$, where δ is staleness in seconds or in missing updates. This curve can be learned offline by replay experiments: take historical traffic, serve features at varying staleness levels, and measure prediction drift or downstream business metrics [16]. The controller then solves a constrained optimization problem per query class:

$$\min_{\pi} \quad \mathbb{E}\left[\sum_{f\in\mathcal{F}_q} w_f \, e_f \Delta_i^\pi k, f, t\right] \qquad \lambda_B \, \mathbb{E}\left[\text{bytes}\pi\right] \tag{3.3}$$

$$\text{s.t.} \quad \mathrm{p}_{99\%}\big(\text{latency}\pi\big) \le L_{\max}, \quad \Pr\big[\Delta_i^\pi k, f, t \le \tau_f\big] \ge 1 - \epsilon_f \;\; \forall f \in \mathcal{F}_q. \tag{3.4}$$

Here Δ_i^π denotes staleness induced by policy π, and λ_B encodes a cost on cross-region traffic. This is a multi-objective problem; the weighted sum is one scalarization, while a Pareto frontier can be estimated by sweeping λ_B or by imposing a bandwidth constraint. The controller can implement an online algorithm that updates π based on observed latency and freshness outcomes using stochastic approximation. For instance, with a policy parameter vector η, one may apply a mirror-descent style update:

$$\eta_{t1} = \arg\min_{\eta\in\Omega} \quad \langle \nabla \widehat{J}_t \eta_t, \eta\rangle \qquad \frac{1}{\gamma_t} D_\psi \eta, \eta_t, \tag{3.5}$$

where $\widehat{J}_t$ is an empirical objective incorporating latency violations and staleness penalties, D_ψ is a Bregman divergence, and Ω encodes feasible policy constraints. Such an update can be implemented without differentiating through the full system by using bandit feedback: sampling alternative policies on a small traffic fraction and estimating gradients from counterfactual outcomes [17]. Safety requires conservative constraints so that exploration does not violate SLOs; this can be enforced via Lagrange

multipliers that increase rapidly on violations:

$$\mathcal{L}\pi, \mu = \mathbb{E}\text{error}\pi \qquad \mu \, \max\left(0, \mathrm{p}_{99\%}\text{latency}\pi - L_{\max}\right), \quad \mu_{t1} = \left[\mu_t \;\; \beta_t \, v_t\right], \tag{3.6}$$

where v_t is an observed violation magnitude and $\cdot$ denotes projection onto nonnegative reals.

Consistency control also interacts with caching. A local cache can dramatically reduce latency, but caching stale values can amplify inconsistency if cache invalidation is delayed. The system can maintain per-feature cache TTLs tied to staleness budgets τ_f, and can invalidate or refresh based on causal frontier advances [18]. If the store exposes a compact frontier token, the cache can tag each entry with the token and validate it cheaply. When replication lag increases due to network congestion, the controller may reduce TTLs for sensitive features and increase them for insensitive ones to preserve overall latency while keeping error bounded.

An additional complexity arises from multi-feature queries. A query feature vector combines multiple features that may be stored in different partitions or even different systems (for example, a low-latency online store plus a graph store). Achieving a consistent cut across features is expensive if done by global snapshots [19]. Instead, the system can aim for a bounded skew constraint, where the difference in frontiers among features does not exceed a tolerance. Let u_f denote the frontier token for feature f returned by a read. A skew metric can be defined as $\max_{f,g \in \mathcal{F}_q} d u_f, u_g$, where d is a frontier distance such as event-time difference. The read policy can enforce a skew bound by selecting replicas or by waiting for lagging features up to a deadline. This resembles deadline-aware barrier synchronization, but with per-feature slack derived from sensitivity weights. The controller can choose which features to wait for and which to serve stale based on the marginal value of freshness relative to the marginal cost in latency [20].

4. Conflict-Free Resolution and Feature Semantics

Conflict resolution is the core mechanism that allows low-latency local writes while maintaining deterministic convergence across regions. The central principle is to represent each feature as a state in a join-semilattice when feasible, so that merging is associative, commutative, and idempotent. When a feature cannot be naturally expressed as a semilattice, the design expands the state to include sufficient metadata so that the merge becomes a semilattice join on the expanded space, or it uses a deterministic resolution rule that is stable under reordering and duplication. This section defines a typed feature algebra that supports common feature patterns in production and relates merge semantics to correctness and cost.

For monotone counters and sums, a classic construction is a grow-only counter or a PN-counter [21]. Each region maintains a component $c^r \in \mathbb{N}$ and the global count is $_r \, c^r$. The merge operation is componentwise maximum:

$$c \sqcup c' = \left(\max c^1, c'^1, \ldots, \max c^R, c'^R\right), \quad \text{value} c = \sum_{r=1}^{R} c^r. \tag{4.1}$$

This yields convergence without coordination and tolerates duplicate dissemination. For increments that may be negative, a PN-counter maintains two grow-only vectors p and n and computes $_r p^r - n^r$. This supports conflict-free aggregation for features like net purchase counts, net balance changes in a bounded domain, or certain time-window approximations when paired with decay mechanisms.

For sets of categorical tokens such as recently viewed items, wishlisted categories, or membership in cohorts, add-wins or remove-wins observed-remove sets can capture concurrent updates. However, features frequently require bounded-size sets such as top-K lists or recency-based truncations [22]. Pure CRDT sets do not directly encode top-K semantics because truncation is non-monotone. One approach is to represent a bounded recency set as a map from item to a timestamped score and define merge as pointwise maximum on scores, followed by deterministic truncation during read materialization. Let the state be $m : \mathcal{I} \to \mathbb{R}$ where $\mathcal{I}$ is the item universe and mi is the latest event-time timestamp observed for

item i. The merge is

$$m \sqcup m'i = \max mi, m'i, \tag{4.2}$$

which is a semilattice join. The bounded set returned to inference is then TopKm, K by timestamp. This separates convergence from presentation [23]. The cost is that the map can grow large; therefore, compaction and approximate representations are needed, such as retaining only items above a decaying threshold or using sketches to approximate membership with bounded false positives.

Time-windowed aggregates such as counts in the last hour or sums in the last day are common and are a source of subtle inconsistency. A naive representation as a scalar with increments and decrements is not conflict-free because window eviction is non-monotone and depends on time. A more robust representation is a ring buffer of bucketed counts keyed by time intervals. Let bucket width be Δ and number of buckets be B, representing a horizon $H = B\Delta$ [24]. The state is an array $a \in \mathbb{N}^{B\times R}$ where ab, r is the count for bucket b contributed by region r. Each update increments the component corresponding to its bucket and writer region. Merge is componentwise maximum as in counters:

$$a \sqcup a' \ : \ ab, r \leftarrow \max ab, r, a'b, r. \tag{4.3}$$

The read computes the window sum by selecting buckets whose time range overlaps the desired window and summing over r. Bucket rotation is monotone if bucket indices are derived from absolute time modulo B but with versioning to avoid reuse ambiguity; a common technique is to tag each bucket with an epoch number that increases when the bucket wraps [25]. The state then stores pairs epoch, count-vector and merges by taking the larger epoch, breaking ties by max on the vector. This yields deterministic convergence even under out-of-order delivery, at the expense of larger metadata.

For last-write-wins scalar features such as profile attributes or latest device type, the usual approach is to attach a timestamp and pick the max. However, timestamps can be inconsistent across regions due to clock skew. Hybrid logical clocks provide a partially ordered time that respects causality while approximating wall time. An HLC value is a pair p, l where p is physical time and l is a logical counter [26]. Comparison is lexicographic. Each update assigns an HLC based on local clock and last seen HLC. Merge selects the value with the higher HLC. This is deterministic and robust to moderate skew; under extreme skew, physical time can dominate and create anomalies, so the system can cap physical jumps and rely more on logical increments. In feature contexts, LWW semantics should be used only when it matches the meaning of the feature; otherwise, it can silently drop concurrent updates that should be aggregated [27].

Embedding-like vector features pose special challenges. Consider an embedding $v \in \mathbb{R}^d$ representing user preferences, updated by incremental learning steps or by aggregating item embeddings. Concurrent updates from different regions should ideally combine rather than overwrite. If updates are additive gradients, a natural conflict-free merge is to maintain a sum of gradient deltas and apply them to a base vector. Let the state be v_0, g where v_0 is a checkpoint vector and $g \in \mathbb{R}^{d\times R}$ stores per-region accumulated gradient sums. Merge is componentwise addition on g after ensuring idempotence via per-region sequence tracking. If each region maintains a monotonically increasing sequence number s_r for its gradient deltas, the state can store s_r, g_r per region and merge by taking the max s_r and associated g_r [28]. The materialized embedding is $v = v_0 \ _r g_r$. To control drift and numeric stability, periodic checkpoints can fold g into v_0 and reset components.

When vector features are maintained as exponential moving averages, conflict-free merges must respect weighting. Suppose each region produces updates $\Delta v, \Delta w$ where Δw is an effective weight, and the feature value is a weighted average $v = \frac{\Delta v}{\Delta w}$. Then the state is the pair of sufficient statistics s, w with $s \in \mathbb{R}^d$ and $w \in \mathbb{R}_{\geq 0}$, merged by addition with idempotence ensured by per-region sequence metadata. This yields a well-defined result independent of arrival order. Additionally, it provides a backprop-friendly representation: if $v = sw$, derivatives with respect to s and w are analytic, and the update plane

can incorporate learned adjustments without needing global coordination [29]. The system can also compress s via quantization or low-rank approximations when d is large. For example, maintain $s \approx U\Sigma z$ where $U \in \mathbb{R}^{d\times r}$ is a shared basis, $\Sigma \in \mathbb{R}^{r\times r}$ diagonal, and $z \in \mathbb{R}^r$ per key, with $r \ll d$. Updates then apply to z in the reduced space. A basis can be learned offline via PCA or randomized SVD over representative embeddings, then deployed to all regions. If U is fixed, merging reduced coefficients remains additive and conflict-free, and the reconstruction error is bounded by the truncated singular spectrum.

Approximate sketches provide conflict-free representations for high-cardinality features such as distinct counts, heavy hitters, or set similarity [30]. For approximate distinct counts, a HyperLogLog-like sketch can be merged via register-wise maximum, which is a semilattice join. For frequency estimation, count-min sketches can be merged by addition if idempotence is ensured at the delta level; alternatively, they can be merged via max for certain monotone formulations. The system must translate sketch error into model error. If a feature value $\hat{c}$ is an estimate of true count c with $\hat{c} \geq c$ and error bound $\hat{c} - c \leq \epsilon N$ with probability $1 - \delta$, then the contribution to inconsistency cost can incorporate this uncertainty. A Bayesian-ish treatment models c as a random variable with posterior interval and propagates it through g_θ using linearization:

$$\mathrm{Var}\big(g_\theta \hat{x}\big) \approx J_\theta \bar{x}\, \mathrm{Cov}\hat{x}\, J_\theta \bar{x}^\top, \tag{4.4}$$

where $\mathrm{Cov}\hat{x}$ includes sketch-induced variance and replication-induced staleness variance. This supports decisions that trade a small increase in uncertainty for significant latency savings.

Not all features admit pure conflict-free semantics [31]. Consider features derived from joins against mutable dimension tables, or features representing the maximum of a sliding window with deletions. In such cases, one can often define a deterministic resolution by carrying additional history or by encoding operations rather than state. An operation-based approach replicates updates as operations that are designed to commute, possibly after transforming them. For example, a sliding maximum can be represented by maintaining a multiset of candidate values with timestamps and using a deterministic rule that discards dominated candidates. The merge combines candidate sets via union and then prunes using a deterministic dominance relation [32]. Although pruning is non-monotone, if the dominance relation ensures that pruned elements can never become maximal in the future, the overall join can still be made monotone on an expanded state that includes all candidates up to a bounded horizon. This is an instance of storing a sufficient set of non-dominated elements, analogous to maintaining a Pareto frontier of candidates.

Conflict-free resolution also must handle late-arriving events and backfills. Backfills can introduce updates with older event times that change aggregates. If the state is defined over event time with appropriate bucketing, late updates simply increment older buckets, and merges remain valid [33]. For features that are defined over processing time rather than event time, late updates may violate expectations; thus, the system should either avoid such definitions for replicated features or record both event time and processing time and define the feature semantics explicitly. A practical compromise is to define feature values for inference in processing time but to compute them via event-time state with a bounded lateness δ, thereby producing stable semantics and making staleness measurable.

5. Optimization Formulations, Complexity, and Approximation Guarantees

Latency-aware consistency and conflict-free resolution provide building blocks, but the overall system must choose replication topology, dissemination rates, compaction aggressiveness, and read strategies under resource constraints. These choices can be posed as optimization problems with multi-objective trade-offs among latency, inconsistency, bandwidth, and compute. Several subproblems are computationally hard, motivating approximation algorithms and heuristics with measurable bounds [34].

A basic design choice is the replication overlay $G = \mathcal{R}, E$ and per-edge replication rates. If each partition log produces a stream of deltas with rate λ_p, then choosing edges to minimize expected propagation delay subject to bandwidth constraints resembles a minimum-cost flow problem. Let x_{ij}^{p} be the rate allocated on edge i, j for partition p, and let c_{ij} be the per-byte cost. A simplified objective is to minimize total cost while ensuring that each region receives updates within a target delay. If delay is approximated by inverse rate for a queueing link, one obtains a convex-like relaxation, but the true system has discrete batching and stochastic variability. Even with simplifications, selecting a sparse overlay to reduce operational complexity while meeting delay targets is related to directed Steiner tree or facility location variants. For example, if one designates a subset of relay regions to forward updates, choosing the minimal set of relays that covers all destinations within latency bounds reduces to set cover [35]. This implies NP-hardness of optimal overlay selection in general. A reduction can be sketched by mapping each potential relay to a set of destinations it can reach within the bound; selecting relays to cover all destinations with minimum cost is set cover, which is NP-hard.

Read strategy selection can also be hard when queries span multiple partitions and features with different sensitivity weights. Consider a query that requires contacting a subset of partitions, each of which can be served from multiple replicas with different latency and freshness characteristics. If one must choose a replica per partition to minimize total latency while keeping aggregate inconsistency below a bound, the problem resembles a knapsack variant [36]. Let each choice a for a partition have cost t_a (latency contribution) and value u_a (freshness utility). Minimizing latency subject to utility constraints is NP-hard in general by reduction from knapsack. This motivates using greedy heuristics, Lagrangian relaxation, or dynamic programming approximations when the number of partitions per query is moderate.

A practical approach is to decompose optimization into per-feature and per-partition decisions with periodic global adjustments. The system can compute per-feature sensitivity weights w_f and per-feature staleness budgets τ_f offline, then enforce them online with local policies [37]. For example, a controller might assign each feature to one of a small set of consistency classes, such as local-only, local-with-fallback, nearest-remote, or quorum. If there are C classes, the decision space reduces to $\mathcal{F} \to \{1, \dots, C\}$. One can fit this mapping by minimizing an empirical objective over traces:

$$\min_{c \cdot} \sum_{\text{queries } q} \left(\widehat{\mathcal{E}}_q c \;\; \lambda_L \, \widehat{\mathcal{V}}_q c \;\; \lambda_B \, \widehat{\mathcal{B}}_q c \right), \tag{5.1}$$

where $\widehat{\mathcal{E}}_q$ estimates inconsistency cost, $\widehat{\mathcal{V}}_q$ penalizes latency SLO violations, and $\widehat{\mathcal{B}}_q$ estimates cross-region bandwidth. This is a discrete optimization; one can use coordinate descent, simulated annealing, or integer programming on a compressed feature set. Because features can be numerous, the system can cluster features by sensitivity and update patterns, reducing dimensionality. Embedding-based clustering of features can be computed from co-access statistics: build a feature co-access graph where vertices are features and edge weights reflect co-occurrence in queries, then embed the graph using spectral methods [38]. Let A be the adjacency matrix and compute a low-rank embedding via truncated SVD $A \approx U_r \Sigma_r V_r^{\top}$, then cluster rows of $U_r \Sigma_r^{12}$. This supports grouping features that tend to be fetched together, improving consistency planning and cache behavior.

When continuous control is desired, one may optimize dissemination rates. Suppose each partition p in region i produces deltas of size b_p bytes at rate λ_p, and these must be transmitted to other regions. Let r_{ij}^{p} be the fraction of deltas sent directly from i to j, with the remainder forwarded via intermediates. The expected propagation delay to j can be approximated by a path delay, and one can select paths by shortest path algorithms on expected delays. However, expected delay is not sufficient when tail behavior matters. One can define an effective edge weight as the α-quantile of delay, $\omega_{ij} = \mathrm{p}_{\alpha} D_{ij}$, and compute shortest paths on ω_{ij} to minimize tail propagation. This yields a deterministic plan, but it may overload

certain edges [39]. A min-cost flow formulation with edge capacities can distribute load:

$$\min \quad \sum_{i,j \in E} c_{ij}\, y_{ij} \tag{5.2}$$

$$\text{s.t.} \quad \sum_{j:i,j \in E} f_{ij}^{p} - \sum_{j:j,i \in E} f_{ji}^{p} = \begin{cases} \lambda_p b_p, & i = \text{source}p \\ -\lambda_p b_p, & i = \text{sink}p \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

$$\sum_{p} f_{ij}^{p} \le y_{ij}, \quad 0 \le y_{ij} \le C_{ij}. \tag{5.4}$$

Here f_{ij}^{p} is flow for partition p and y_{ij} is reserved capacity. In practice, partitions have multiple sinks, and replication is many-to-many; the formulation can be extended with super-sinks or by solving per-source distributions. The complexity is high, but periodic approximate solutions can guide rate limits and relay choices.

Conflict-free state often increases payload size, which impacts bandwidth. Compression and coding are therefore part of optimization [40]. If updates are delta-encoded and entropy-coded, the expected bits per update is at least the entropy of the delta distribution. Let Δ be a random variable for the update payload after canonicalization. Shannon bounds imply that any prefix-free code has expected length $\mathbb{E}\ell\Delta \ge H\Delta$. Thus, improving canonicalization to reduce entropy is as important as selecting a code. For example, bucketed time-series states can produce sparse deltas; run-length encoding of zero buckets reduces entropy. For embedding deltas, quantization to q bits per component reduces size by a factor of $32q$ compared to float32, at the cost of quantization error [41]. The system can select q per feature based on sensitivity, framing it as a rate-distortion problem. If distortion is measured by mean squared error and rate is bits, one can define

$$\min_{q_f} \quad \sum_{f} w_f\, D_f q_f \quad \text{s.t.} \quad \sum_{f} R_f q_f \le B_{\text{budget}}, \tag{5.5}$$

where D_f is expected distortion and R_f is expected rate. Even if closed-form D_f is unknown, it can be empirically measured. A greedy allocation based on marginal distortion reduction per bit is a practical heuristic [42].

Approximate sketches introduce probabilistic error. The system must incorporate these errors into end-to-end guarantees. For count-min sketches with width w and depth d, the estimate $\hat{c}x$ satisfies $\hat{c}x \le cx \;\; \epsilon N$ with probability at least $1 - \delta$ where $\epsilon \approx ew$ and $\delta \approx e^{-d}$. This yields a tunable trade-off between memory and error. If the feature store uses sketches for replicated state, merge operations are efficient: addition for operation-based sketches or elementwise max for certain monotone variants. However, staleness compounds sketch error because missing updates reduce N and thus reduce the bias bound, but they also produce systematic undercount relative to the reference [43]. Therefore, the total discrepancy can be decomposed as

$$\tilde{c}x - c^{\star}x = \underbrace{\left(\tilde{c}x - c_{\text{local}}x\right)}_{\text{sketch error}} \qquad \underbrace{\left(c_{\text{local}}x - c^{\star}x\right)}_{\text{replication staleness}}, \tag{5.6}$$

where c_{local} is the exact count that would be obtained from the updates incorporated locally. Bounding each term separately enables conservative guarantees. For inference, conservative bounds may be too pessimistic; thus, the controller can use empirical distributions rather than worst-case bounds to allocate budgets.

Query planning across multi-region replicas can be framed as a dynamic program on a DAG representing dependencies among features and transformations. Suppose a query requires a set of raw features $\mathcal{F}_0$ and derived features computed by transformations $\mathcal{T}$, forming a DAG where nodes are intermediate feature sets. Each node u can be materialized at certain regions with associated latency and freshness profiles, or computed on the fly with compute cost [44]. The planner chooses where to read and where

to compute. If the DAG is acyclic and the number of transformation choices per node is bounded, one can compute an optimal plan by DP over topological order using a cost function that combines latency and inconsistency. Let Cu, r be the cost to obtain node u at region r. Then

$$Cu, r = \min_{\text{option } o \in \mathcal{O}u, r} \left(\text{lat}o \quad \eta \, \text{err}o \quad \sum_{v \in \text{pred}u} Cv, r_o \right), \tag{5.7}$$

where r_o is the region from which predecessors are obtained under option o. The planner can incorporate deadlines by truncating options whose latency exceeds remaining budget [45]. While this DP can be expensive for large graphs, queries typically involve tens to hundreds of features, and feature families can be grouped. Approximation can further reduce complexity by limiting the set of candidate regions to the nearest few and by caching DP results for common query templates.

6. Systems Engineering: Storage Internals, Distributed Execution, and Performance

Achieving predictable tail latency while maintaining mergeable semantics requires careful performance engineering across storage, networking, and execution. The system must support high write rates from streaming pipelines, high read QPS from inference, and background compaction and replication, all while avoiding interference that inflates p99 latency. This section describes a practical design centered on log-structured storage, structured state encoding, batching and backpressure, and multi-tenant isolation.

The primary storage engine is an LSM-tree or similar log-structured store, chosen for write throughput and compaction-based maintenance [46]. Keys are composed as partition, k, f, with optional sharding by feature family to separate hot and cold features. Values store the typed state plus merge metadata. To minimize read amplification, the engine maintains a materialized view for frequently accessed features, potentially in a separate column family with aggressive compaction. For large structured states such as ring buffers or per-item timestamp maps, the engine can store a compacted representation plus a small delta log, enabling fast updates by appending deltas and periodic folding. This resembles a mini-LSM per key: a sequence of recent deltas plus a base snapshot, with merging during reads or during background compaction. The design must ensure that read-time merging does not dominate latency; thus, it is crucial to cap the number of deltas to merge and to prioritize compaction for hot keys [47].

Encoding is a major determinant of both bandwidth and CPU. Typed states can be encoded using fixed-layout binary formats to avoid parse overhead. For vectors, use quantized formats with SIMD-friendly dequantization. For maps, use sorted arrays of key-value pairs with delta-encoded keys and variable-byte coding for timestamps. Compression can be applied at the block level in the LSM and at the replication message level [48]. To reduce CPU on the read path, compression choices should favor fast decompression. A common approach is to use lightweight compression for hot data and stronger compression for cold historical data. Since feature stores frequently maintain multi-version data for training, older versions can be compressed more aggressively without affecting inference latency.

Replication dissemination is implemented as streaming of log segments. Each partition maintains a local append-only log of updates (or merged deltas) with monotonically increasing offsets [49]. Replication consumers in other regions fetch segments and apply them to their local state. To support at-least-once delivery with idempotent application, each update carries an identifier such as source region, offset. For operation-based updates, idempotence is natural. For state-based CRDT merges, repeated merges are safe. For certain mixed cases, the receiver tracks the highest applied offset per source region, enabling duplicate suppression. This tracking can be maintained per partition to avoid large per-key metadata [50]. Late updates from backfills may arrive as special streams; they can be treated as separate sources with separate offsets, preserving idempotence.

Distributed execution for updates must handle ordering and concurrency. Within a partition, updates can be applied in log order for each source, but different sources can interleave. Because merges are commutative for most feature types, strict ordering is unnecessary for correctness, though it can affect intermediate materializations. For LWW features, the ordering is determined by HLC, not by arrival [51]. For bucketed aggregates, ordering within buckets is irrelevant. Therefore, the update plane can process

updates in parallel and rely on merge logic to converge. This simplifies throughput scaling. Nevertheless, careful engineering is needed to avoid contention on hot keys. Techniques include sharding hot keys by feature family, applying updates in micro-batches, and using lock-free or striped-lock structures for in-memory memtables [52].

The read path must minimize network hops and avoid blocking on slow replicas. A typical implementation maintains a local replica in each region, with reads served locally when possible. The latency-aware controller may choose to consult a remote replica for sensitive features when local freshness is insufficient. To avoid paying multiple network round trips per feature, the system batches feature reads and routes them to replicas by partition. A query fan-out to many partitions can be expensive; thus, the system can collocate commonly co-accessed features, guided by the co-access graph embedding discussed earlier [53]. When collocation is not possible, the system uses asynchronous fan-out with a deadline: it issues requests in parallel and returns the best available values by the deadline, possibly with freshness annotations. For models that can tolerate missing features, the system can return partial vectors with defaults. However, defaults change model behavior; thus, the design prefers serving stale-but-present values over missing values for many features, unless missingness is explicitly handled in training.

Caching is integrated as a two-level mechanism: an in-process cache at inference services for extremely hot keys and a regional cache near the feature store for general hot sets. Cache entries store both values and frontier tokens. On a cache hit, the system can accept the value if the token satisfies the current staleness budget; otherwise it triggers a refresh [54]. This avoids coarse TTLs that either waste freshness or risk excessive staleness. Cache admission can be guided by a cost model: cache keys with high read frequency and high remote-read cost, and avoid caching large states that are expensive to store. For embedding vectors, caching quantized forms can reduce memory footprint, with optional dequantization at use time.

Isolation and multi-tenancy matter because feature stores serve multiple models and teams. Background compaction and replication can interfere with inference reads [55]. The system should implement priority scheduling: inference reads are high priority, update ingestion is medium priority, and compaction and repair are low priority but must make progress. Resource governance can be implemented via token buckets for CPU, disk I/O, and network. For example, replication bandwidth per edge can be rate-limited, and compaction can be paused under high tail latency. This interacts with freshness; thus, the controller must consider that throttling replication can increase staleness. A stability mechanism is to allocate a minimum replication bandwidth to sensitive feature partitions and allow less sensitive partitions to fall behind during overload [56].

Conflict-free state can bloat storage, so compaction and pruning policies are essential. For per-item timestamp maps, the system can apply deterministic pruning rules that preserve correctness for the intended feature semantics. For example, if the feature returns top-K most recent items, any item with timestamp below the K-th most recent can be discarded, provided that future merges cannot resurrect it with a higher timestamp unless a new update occurs, in which case it will reappear. This pruning can be applied per region and during merge; since merge takes max timestamps, discarding old timestamps is safe as long as the pruning threshold is computed conservatively. For ring buffers, epochs prevent ambiguity on wrap-around, and old epochs can be discarded once they fall beyond the lateness bound [57]. For vector sufficient statistics, periodic checkpointing folds sums into a base and clears per-region components, reducing metadata. Checkpointing must be coordinated carefully to preserve convergence; a safe method is to checkpoint by writing a new base with an associated frontier token and keeping the old components until all regions have acknowledged incorporation of the checkpoint marker, akin to a distributed garbage collection barrier.

Backpressure and overload handling are crucial. Under surges, replication lag increases, causing reads to require more remote fetches, which further increases load, risking a feedback loop. The controller should detect this regime via lag and tail latency metrics and shift to a degraded mode where it serves local stale values for lower-sensitivity features and reduces remote fetches [58]. This is a deliberate trade-off: protect latency SLOs while accepting increased inconsistency, but do so in a measured way

that keeps high-sensitivity features fresher. The design can incorporate per-feature criticality classes so that under overload, critical features retain remote fetch budgets while noncritical features are served locally. This is similar to load shedding but applied to consistency.

Security and compliance are also intertwined with multi-region replication. Some features may be restricted to certain regions [59]. The system must enforce placement constraints that limit where state can be stored and where reads can be served from. This can be expressed as constraints in the optimization: for each feature f, allowed regions are $\mathcal{R}_f \subseteq \mathcal{R}$, and replication edges must not violate data residency. Similarly, encryption and key management can add latency; thus, encryption should be performed with hardware acceleration where possible, and replication messages should be batched to amortize overhead. Auditability requires that the system logs policy decisions, frontier tokens served, and conflict resolutions in a way that supports later analysis without recording sensitive raw values unnecessarily.

7. Evaluation Methodology and Reproducibility

A system centered on latency-aware consistency and semantics-based conflict resolution must be evaluated not only on throughput and latency, but also on staleness distributions, conflict rates, convergence behavior, and model impact. Reproducible evaluation requires controlled workloads, trace-driven experiments, and fault injection that reflects real multi-region behavior [60]. This section outlines an evaluation plan that can be executed in staging environments and partially reproduced with open traces or synthetic generators.

Workloads should capture the heterogeneity of feature pipelines. Update workloads include high-rate streams with small deltas, periodic batch recomputations, and occasional large backfills. Read workloads include high-QPS inference queries with a stable feature subset, plus background reads for monitoring and training data extraction. Trace-driven replay is preferable: record update and read streams with timestamps, key distributions, and feature access patterns, then replay them into a testbed with configurable region latencies and failures [61]. When traces are unavailable, synthetic generators should match key popularity distributions (often heavy-tailed), temporal burstiness, and feature co-access patterns. A co-access graph can be generated with community structure to mimic model-specific feature bundles.

Metrics must include end-to-end inference latency percentiles, especially 99% and 99.9%, and must separate the contributions of network, storage, and compute. Freshness metrics include event-time and wall-clock staleness distributions per feature, measured as differences in watermarks or missing update counts. Conflict metrics include the rate of concurrent updates detected by causality metadata, the distribution of conflict types by feature class, and the cost of merge operations in CPU and memory. Convergence metrics include time-to-converge under normal conditions and under partitions, and divergence magnitude during partitions [62]. Because the system aims to relate staleness to model impact, evaluation should measure prediction drift. One approach is to take a fixed model and compare outputs under reference features versus served features. The reference can be computed by running a centralized merge with a generous lateness bound and full update incorporation. Prediction drift can be summarized by mean absolute difference, KL divergence for probabilistic outputs, or ranking changes for recommendation outputs. Importantly, drift should be reported by traffic slices, because sensitivity may vary by user segment or context [63].

Fault injection is necessary to test multi-region behavior. Scenarios include increased latency on certain inter-region links, partial packet loss, one region becoming unavailable, and partitions that isolate subsets of regions. The replication plane should be tested for repair behavior: after a partition heals, how quickly do replicas converge, and what is the impact on read latency as catch-up traffic competes for resources. In addition, clock skew scenarios should be tested to validate HLC behavior for LWW features. Backfill scenarios should be tested by injecting historical updates that overlap with current streams and measuring whether aggregates and derived features remain stable [64]. The evaluation should also include overload scenarios where update rates spike and replication lag grows, to test the controller's degraded modes and ensure that latency SLO protection behaves as intended.

Reproducibility requires that experiments are parameterized and that all relevant configurations are logged. This includes replication overlay, bandwidth limits, compaction settings, cache policies, controller parameters, and feature typing definitions. The system should provide a deterministic mode for debugging where randomness is controlled and scheduling is simplified, allowing repeated runs to be comparable. For conflict-free state, deterministic convergence should be validated by replaying the same update stream in different orders and verifying that final states match bit-for-bit for semilattice-based features and match within defined tolerances for approximate features and quantized vectors [65]. To support this, each feature type should have a conformance test suite that validates associativity, commutativity, and idempotence of merge, as well as semantic invariants such as monotonicity of certain derived values. For approximate sketches, reproducibility should specify random seeds for hash functions and confirm that merge results are independent of dissemination order.

A critical part of evaluation is separating improvements due to semantic merges from improvements due to policy control. Ablation can be performed by fixing conflict resolution semantics and varying consistency policies, and then fixing policies and varying feature typing and merge rules. Another axis is comparing the latency-aware controller to static consistency classes [66]. Static baselines might include local-only eventual reads, always-quorum reads, and nearest-region reads without freshness checks. The controller's benefit should be reflected in reduced tail latency for the same prediction drift, or reduced drift for the same latency. Because model impact can be noisy, evaluation should include confidence intervals derived from repeated trace segments or bootstrap sampling. Where possible, online A/B experiments can validate trace-derived results, but they must be carefully limited to avoid user harm; staging or shadow evaluation is typically safer for early validation.

Finally, the evaluation should report resource costs: cross-region bandwidth, storage overhead from typed state, CPU overhead from merge operations, and energy proxies such as CPU time and network bytes [67]. The system's design aims to allocate these costs where they buy the most reduction in inconsistency error, so reporting per-feature or per-feature-family breakdowns is informative. Even without external references, results should include sufficient detail for an informed reader to replicate the testbed: region topology, RTT distributions, bandwidth caps, machine types, dataset sizes, and workload rates.

8. Conclusion

Multi-region replication for feature pipelines requires more than choosing between strong and weak consistency. Feature stores sit at the boundary between distributed systems and statistical learning, where the cost of staleness depends on feature semantics and model sensitivity, and where tail latency dominates user experience. This paper presented a framework that couples latency-aware consistency control with conflict-free, semantics-grounded resolution [68]. The system model treats features as typed mergeable state, enabling deterministic convergence under concurrent updates, late arrivals, and replication reordering. Consistency is expressed as probabilistic freshness constraints and skew tolerances that can be selected per feature and per workload slice, guided by sensitivity estimates that connect staleness to prediction drift. Optimization formulations capture multi-objective trade-offs among latency, inconsistency, bandwidth, and compute, while acknowledging NP-hardness in overlay and plan selection and motivating practical approximations. Systems engineering details emphasize log-structured storage, structured encoding, batching, caching with frontier tokens, and resource isolation to protect inference tail latency. The evaluation methodology focuses on reproducible measurement of latency, freshness, conflict behavior, convergence, and model impact under trace replay and fault injection. The overall approach supports predictable multi-region feature serving without requiring global coordination on the critical path, while keeping inconsistency measurable and bounded in ways that align with the semantics of features and the requirements of inference workloads [69].

References

[1] N. Zbitnev, D. Shishlyannikov, and D. Gridin, "Probabilistic block cipher for distributed systems," *Journal of Physics: Conference Series*, vol. 1117, pp. 012011–, 11 2018.

[2] A. R. Pratama, W. Widyawan, A. Lazovik, and M. Aiello, "Multi-user low intrusive occupancy detection," *Sensors (Basel, Switzerland)*, vol. 18, pp. 796–, 3 2018.

[3] P. Foytik and S. Shetty, *Blockchain for Distributed Systems Security - Blockchain Evaluation Platform*. Wiley, 3 2019.

[4] E. Collini, F. I. Kurniadi, P. Nesi, and G. Pantaleo, "Context-aware retrieval augmented generation using similarity validation to handle context inconsistencies in large language models," *IEEE Access*, pp. 1–1, 1 2025.

[5] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Programming open distributed systems in maude," in *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, pp. 1–12, ACM, 9 2024.

[6] C. Marcelino, S. Gollhofer-Berger, T. Pusztai, and S. Nastic, "Cosmos: A cost model for serverless workflows in the 3d compute continuum," in *2025 IEEE International Conference on Smart Computing (SMARTCOMP)*, pp. 106–113, IEEE, 6 2025.

[7] F. Hackett, S. Hosseini, R. Costa, M. Do, and I. Beschastnikh, "Compiling distributed system models with pgo," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 159–175, ACM, 1 2023.

[8] M. Bisen, "Integration technology of distributed system based on multi-objective hotopy algorithm," *Distributed Processing System*, vol. 3, 7 2022.

[9] R. Malik, S. Kim, X. Jin, C. Ramachandran, J. Han, I. Gupta, and K. Nahrstedt, "Mlr-index: An index structure for fast and scalable similarity search in high dimensions," in *International Conference on Scientific and Statistical Database Management*, pp. 167–184, Springer, 2009.

[10] A. V. Vesa, S. Vlad, R. Rus, M. Antal, C. Pop, I. Anghel, T. Cioara, and I. Salomie, "Iccp - human activity recognition using smartphone sensors and beacon-based indoor localization for ambient assisted living systems," in *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pp. 205–212, IEEE, 9 2020.

[11] C. Zhu, S. Wang, X. Fan, X. Deng, S. Liu, Y. He, and C. Wu, "Blockchain-enhanced federated learning for secure and intelligent consumer electronics : An overview," *IEEE Consumer Electronics Magazine*, pp. 1–12, 1 2025.

[12] R. Gajanin, A. Danilenka, A. Morichetta, and S. Nastic, "Towards adaptive asynchronous federated learning for human activity recognition," in *Proceedings of the 14th International Conference on the Internet of Things*, pp. 38–46, ACM, 11 2024.

[13] O. Pavliuk, M. Medykovskyy, R. Cupek, and M. Mishchuk, "Modern methods of data preprocessing to increase the accuracy of agv battery discharge forecast," in *2024 IEEE 19th International Conference on Computer Science and Information Technologies (CSIT)*, pp. 1–4, IEEE, 10 2024.

[14] Y. Tan and Z. Mi, "Performance analysis and optimization of nvidia h100 confidential computing for ai workloads," in *2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pp. 1426–1432, IEEE, 10 2024.

[15] J. Pennekamp, P. Weiler, M. Bodenbenner, M. Sudmann, I. Koren, I. Kunze, M. Fey, D. Wolfschläger, C. Brecher, R. H. Schmitt, and K. Wehrle, "Confmod: A simple modeling of confidentiality requirements for inter-organizational data sharing," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, pp. 1–6, IEEE, 5 2025.

[16] I. Schagaev, *Distributed Systems: Maximizing Resilience*, pp. 249–266. Springer International Publishing, 7 2019.

[17] S. C. Voinea, S. Vladov, and F. Rensing, "Coronaz: another distributed systems project.," 2 2021.

[18] A. Desai, A. Phanishayee, S. Qadeer, and S. A. Seshia, "Compositional programming and testing of dynamic distributed systems," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 159–30, 10 2018.

[19] N. Rathore and J. Rathore, "Efficient checkpoint algorithm for distributed system," *International Journal of Engineering in Computer Science*, vol. 1, pp. 59–66, 7 2019.

[20] S. Avasthi and S. L. Tripathi, “Distributed system architecture and computing models,” 11 2025.

[21] C. Gao and T. Braun, “Two-stage hybrid edge caching framework for 360° vr video,” in *2025 IEEE 26th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 1–10, IEEE, 5 2025.

[22] M. Rafailescu, “Fault tolerant leader election in distributed systems,” *Zenodo (CERN European Organization for Nuclear Research)*, 12 2022.

[23] L. Hunhold and J. Quinlan, “Evaluation of bfloat16, posit, and takum arithmetics in sparse linear solvers,” in *2025 IEEE 32nd Symposium on Computer Arithmetic (ARITH)*, pp. 61–68, IEEE, 5 2025.

[24] R. Chandrasekar, R. Suresh, and S. Ponnambalam, “Evaluating an obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs,” in *2006 International Conference on Advanced Computing and Communications*, pp. 628–629, IEEE, 2006.

[25] R. Sukharev, O. Lukyanchikov, E. Nikulchev, D. Biryukov, and I. Ryadchikov, “Methods and tools for profiling and control of distributed systems,” *IOP Conference Series: Materials Science and Engineering*, vol. 312, pp. 012024–, 3 2018.

[26] “Proceedings of the 1st international symposium on parallel computing and distributed systems,” in *2024 International Symposium on Parallel Computing and Distributed Systems (PCDS)*, pp. 1–1, IEEE, 9 2024.

[27] Z. Zhao, M. Wu, H. Chen, and B. Zang, “Characterization and reclamation of frozen garbage in managed faas workloads,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 281–297, ACM, 4 2024.

[28] P. Raith, S. Nastic, and S. Dustdar, “Simuscale: Optimizing parameters for autoscaling of serverless edge functions through co-simulation,” in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pp. 305–315, IEEE, 7 2024.

[29] F. Fernández-Bravo Peñuela, J. Arjona Aroca, F. Muñoz-Escoí, Y. Yatsyk Gavrylyak, I. Illán García, and J. Bernabéu-Aubán, “Delta: A modular, transparent, and efficient synchronization of dlts and databases,” *International Journal of Network Management*, vol. 34, 8 2024.

[30] P. Bellini, P. Nesi, and G. Pantaleo, “Iot-enabled smart cities: A review of concepts, frameworks and key technologies,” *Applied Sciences*, vol. 12, pp. 1607–1607, 2 2022.

[31] T. Srinivasan, R. Chandrasekar, V. Vijaykumar, V. Mahadevan, A. Meyyappan, and A. Manikandan, “Localized tree change multicast protocol for mobile ad hoc networks,” in *2006 International Conference on Wireless and Mobile Communications (ICWMC’06)*, pp. 44–44, IEEE, 2006.

[32] O. V. Talaver and T. A. Vakaliuk, “Reliable distributed systems: review of modern approaches,” *Journal of Edge Computing*, vol. 2, pp. 84–101, 5 2023.

[33] L. Su, X. Wang, and L. Wang, “A resilience analysis method for distributed system based on complex network,” in *2021 IEEE International Conference on Unmanned Systems (ICUS)*, pp. 238–243, IEEE, 10 2021.

[34] P. K. Donta, I. Murturi, V. C. Pujol, B. Sedlak, and S. Dustdar, “Exploring the potential of distributed computing continuum systems,” *Computers*, vol. 12, pp. 198–198, 10 2023.

[35] Y. Braidiz, D. Efimov, A. Polyakov, and W. Perruquetti, “On robustness of finite-time stability of homogeneous affine nonlinear systems and cascade interconnections,” *International Journal of Control*, pp. 1–11, 9 2020.

[36] K. Gorokhovskyi, O. Zhylenko, and O. Franchuk, “Distributed system technical audit,” *NaUKMA Research Papers. Computer Science*, vol. 3, pp. 69–74, 12 2020.

[37] T. Kharkovskaia, “Design of interval observers for uncertain distributed systems,” 12 2019.

[38] R. Chandrasekar and T. Srinivasan, “An improved probabilistic ant based clustering for distributed databases,” in *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI*, pp. 2701–2706, 2007.

[39] M. Broy, “Concurrent distributed systems beyond monotonicity,” 1 2024.

[40] E. B. Gulcan, J. Neto, and B. K. Ozkan, “Generalized concurrency testing tool for distributed systems,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1861–1865, ACM, 9 2024.

[41] S. Gorai, *Decentralization without Blockchains*, pp. 24–32. Productivity Press, 8 2024.

[42] Y. Yang, D. Du, H. Song, and Y. Xia, "On-demand and parallel checkpoint/restore for gpu applications," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 415–433, ACM, 11 2024.

[43] N. Benmoussa, M. F. Amr, S. Ahriz, K. Mansouri, and E. Illoussamen, "Outlining a model of an intelligent decision support system based on multi agents," *Zenodo (CERN European Organization for Nuclear Research)*, 6 2018.

[44] S. H. Haeri and P. V. Roy, "Purely functional distributed systems programming," 10 2020.

[45] C. Akmut, "6.824 'distributed systems' (special 4)," 12 2023.

[46] V. Matkovic, M. Josten, and T. Weis, "Optical token transmission for secure iot device discovery and control in ubiquitous environments," in *Companion of the 2025 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pp. 1530–1533, ACM, 10 2025.

[47] J. Raffety, B. Stone, J. Svacina, C. Woodahl, T. Cerny, and P. Tisnovsky, *Multi-source Log Clustering in Distributed Systems*, pp. 31–41. Germany: Springer Singapore, 4 2021.

[48] A. Labutkina, A. Selezneva, T. John, and D. Hausheer, "Multiobjective path optimization for deadline-aware multipath over scion," in *2024 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pp. 56–62, IEEE, 5 2024.

[49] "Three-layer distributed system based on bayesian classifier," *Distributed Processing System*, vol. 3, 10 2022.

[50] B. Sedlak, V. C. Pujol, P. K. Donta, and S. Dustdar, "Active inference on the edge: A design study," in *2024 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 550–555, IEEE, 3 2024.

[51] T. Maalouf, "Performance optimizations of nosql databases in distributed systems," 12 2020.

[52] V. Vijaykumar, R. Chandrasekar, and T. Srinivasan, "An obstacle avoidance strategy to ant colony optimization algorithm for classification in event logs," in *2006 IEEE Conference on Cybernetics and Intelligent Systems*, pp. 1–6, IEEE, 2006.

[53] X. Song, R. Chen, H. Song, Y. Zhang, and H. Chen, "Unified and near-optimal multi-gpu cache for embedding-based deep learning," *ACM Transactions on Computer Systems*, vol. 44, pp. 1–32, 11 2025.

[54] M. Goudarzi, Q. Deng, and R. Buyya, *Resource management in edge and fog computing using FogBus2 framework*, pp. 17–52. The Institution of Engineering and Technology, 5 2024.

[55] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu, *HotOS - Metastable failures in distributed systems*. ACM, 6 2021.

[56] V. K. Yadav, "Improve the scalability of system through distributed system," 3 2021.

[57] J. Köstler, H. P. Reiser, F. J. Hauck, and G. Habiger, "Fluidity: Location-awareness in replicated state machines," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pp. 192–201, ACM, 3 2023.

[58] C. Ankapanaidu, "Distributed system in cloud audit data storage," *International Journal of Big Data Security Intelligence*, vol. 4, pp. 1–6, 6 2018.

[59] F. Ansari and A. Afzal, *A Grid-Connected Distributed System for PV System*, pp. 919–924. Springer Singapore, 1 2021.

[60] J. M. Diament, "Com3800: Intro to distributed systems," 9 2019.

[61] P. M. Moretti, *Approximations for Distributed Systems*, pp. 55–68. CRC Press, 9 2024.

[62] K. Karthikeyan, S. Hemalatha, and S. Vignesh, "Introduction to distributed systems," 11 2025.

[63] M. Itmi and A. E. Hami, "Distributed system approach to experiment regional competitiveness," *Computer Science & Information Technology*, pp. 103–109, 2 2018.

[64] Y. Liu, T. Xu, Z. Mi, Z. Hua, B. Zang, and H. Chen, "Cps: A cooperative para-virtualized scheduling framework for manycore machines," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pp. 43–56, ACM, 3 2023.

[65] T. Pollinger, A. V. Craen, C. Niethammer, M. Breyer, and D. Pflüger, "Leveraging the compute power of two hpc systems for higher-dimensional grid-based simulations with the widely-distributed sparse grid combination technique," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14, ACM, 11 2023.

[66] R. Malik, C. Ramachandran, I. Gupta, and K. Nahrstedt, "Samera: a scalable and memory-efficient feature extraction algorithm for short 3d video segments.," in *IMMERSCOM*, p. 18, 2009.

[67] G. R. S. Martinez, "Soa distributed systems architecture," *Scientia et Technica*, vol. 26, pp. 328–334, 9 2021.

[68] F. N. Al-Wesabi, H. G. Iskandar, and M. M. Ghilan, "Improving performance in component based distributed systems," *ICST Transactions on Scalable Information Systems*, vol. 6, pp. 159357–, 7 2019.

[69] "Evaluation of safety and survivability of distributed system using complexity proportionality assessment (copras) method," *Journal on Innovations in Teaching and Learning*, vol. 4, pp. 24–33, 9 2025.