Original Research

Incremental Community Detection on Streaming Graphs with Bounded Memory and Distributed Merge Guarantees

Suresh Bhatta[1] and Anil Kharel[2]

[1]Lumbini Buddhist University, Faculty of Science and Technology, Tansen Road, Butwal 32907, Nepal.
[2]Kailali Multiple Campus, Department of Computer Science, Campus Road, Dhangadhi 10900, Nepal.

Abstract
Streaming interaction data induces graphs whose vertex and edge sets evolve continuously, often at rates that make batch community detection impractical. In such settings, the detection objective is typically not only to track latent group structure, but also to do so under strict limits on memory, latency, and communication, while tolerating out-of-order updates and distributed ingestion. This paper studies incremental community detection on streaming graphs under bounded memory, emphasizing mergeable state representations that admit deterministic, distributed reconciliation with formal guarantees. The core premise is that community detection in a stream should be treated as an online optimization and inference problem whose sufficient statistics are carefully chosen to support approximate yet stable updates without retaining full adjacency. We formulate a model that unifies modularity-like partition quality, stochastic blockmodel-inspired likelihood, and embedding-based separation, and we derive incremental update rules that operate per edge in small amortized time. To respect memory budgets, we use bounded retention windows and sketch-based estimators for degrees, cuts, and inter-community affinities, with explicit approximation error propagation. For distributed ingestion, we define merge operators over compact community summaries and prove associativity and commutativity properties needed for eventual consistency, along with bounds on quality loss relative to a centralized baseline. We also address performance engineering in storage internals and distributed execution, and we outline evaluation protocols designed for reproducibility under streaming nondeterminism.

1. Introduction

Community detection is usually posed for a static graph, where the full adjacency is available and repeated passes over edges are feasible [1]. Many operational graphs are not static: they are induced by event streams such as messages, transactions, device telemetry, or social interactions, and they change at time scales that invalidate batch recomputation. A streaming view induces a time-indexed sequence of edge updates, possibly with weights, attributes, and deletions. The engineering constraints are typically as important as the statistical ones. State must be compact enough to fit in memory even when the graph contains billions of vertices. Updates must be processed with bounded latency, often near constant time per edge, and distributed ingestion requires merging partial states across workers without global coordination [2]. These constraints collide with the computational nature of community detection: many classical objectives are combinatorial, nonconvex, and, for widely used quality functions, computationally intractable in the worst case. The tension motivates algorithms that are online, approximate, and explicitly designed around mergeable summaries.

Two pragmatic observations shape the approach in this paper. The first is that in a stream, the notion of a "correct" partition is itself time-dependent, and it is more useful to optimize a surrogate that is stable under incremental updates than to chase a batch optimum that can only be approximated after the fact. The second is that, under memory bounds, the algorithm cannot treat the graph as a full data structure; it must treat the stream as the data, and the maintained state as a set of sufficient statistics and sketches. The resulting design is closer to online learning than to static graph partitioning: we maintain

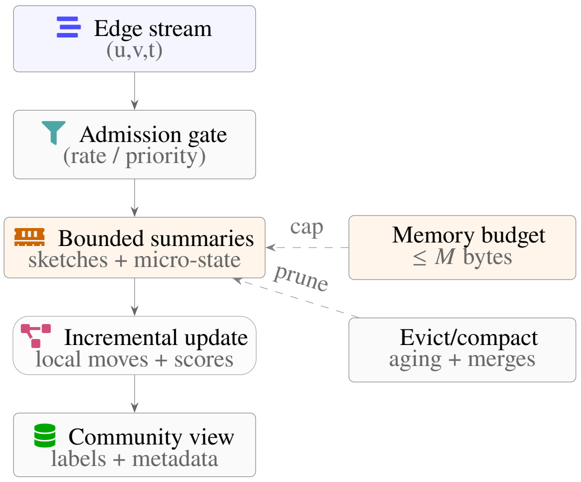

Figure 1: Streaming pipeline for incremental community detection under a strict memory cap. The system ingests edges, applies a lightweight admission policy, maintains bounded summaries (e.g., sketches and compact community state), and performs small local updates that continuously refresh the community view. Optional eviction/compaction mechanisms keep memory within M while preserving mergeable state.

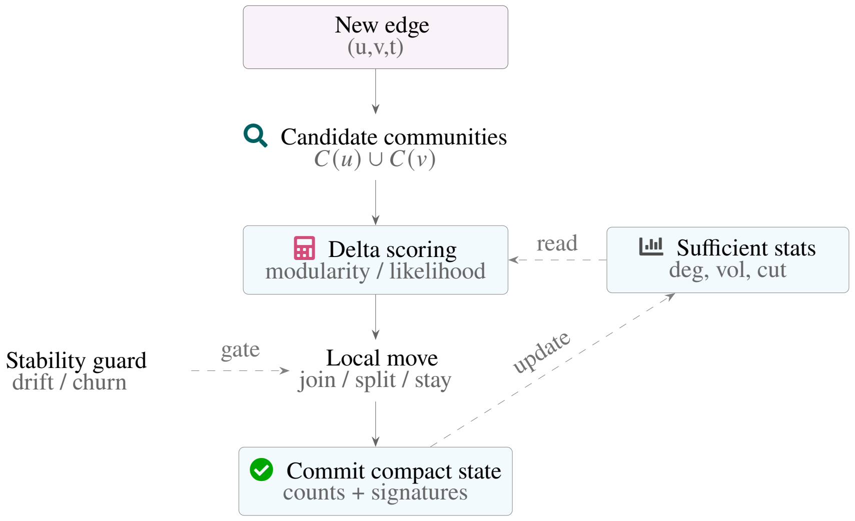

Figure 2: Edge-by-edge incremental update loop. Each arriving edge proposes a small set of candidate communities, evaluates a local objective change using compact sufficient statistics, and applies a bounded local move before committing mergeable state. Optional stability guards reduce churn under concept drift or bursty streams.

a parameterized partition model, update it with stochastic gradients or greedy local moves, and use sketches to approximate the quantities that would otherwise require access to full adjacency [3].

The distributed setting adds an additional requirement that is not automatically satisfied by online updates. When the stream is sharded across workers, each worker observes only a substream and updates a local state. Periodic reconciliation is needed to obtain a global partition or to answer queries that cross shards. Naively merging partitions is ambiguous because community labels are arbitrary and because local decisions can conflict. If reconciliation is done via centralized recomputation, the system loses the benefits of streaming and distribution [4]. The key requirement studied here is a distributed merge

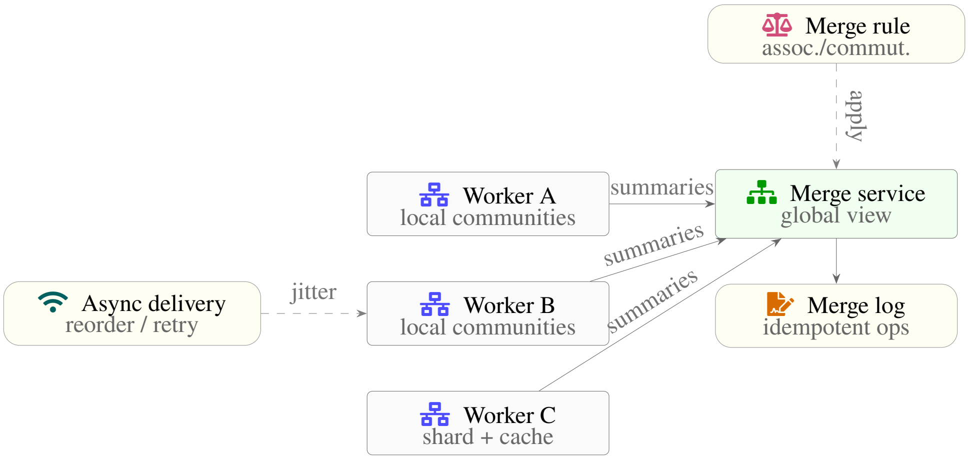

Figure 3: Distributed merge architecture with merge guarantees. Workers maintain shard-local community state and periodically emit mergeable summaries. A merge service applies associative/commutative rules recorded in an idempotent log, enabling consistent global views despite asynchronous delivery, reordering, and retries.

Dataset	$\lvert V\rvert$	$\lvert E\rvert$ (final)	Stream type
Twitter-Stream	41M	1.3B	Temporal interactions
Friendster-Live	65M	1.8B	Growing social graph
WebGraph-Click	12M	420M	Clickstream edges
Citation-Online	3.2M	32M	Incremental citations
Synth-ScaleFree	10M	200M	Generator (Barabási-Albert)

Table 1: Streaming graph datasets used in the evaluation.

Parameter	Symbol	Default	Role
Sliding window size (edges)	W	10^5	Temporal locality control
Update batch size	B	10^3	Amortization of maintenance
Memory budget per worker	M	512 MB	Bounded state constraint
Merge interval	τ	30 s	Frequency of distributed merge
Minimum community size	$s_{\min}$	5	Filtering of small clusters
Stability threshold	θ	0.8	Pruning unstable communities

Table 2: Key hyperparameters of the incremental detection algorithm.

guarantee: local states should be mergeable via operators that are deterministic and algebraically well-behaved, so that repeated merges in any order converge to the same result, and the loss relative to centralized processing is bounded.

The paper develops a framework that couples three ingredients. The first is a streaming objective that combines a modularity-inspired cut/volume term with a probabilistic regularizer akin to a degree-corrected block model, and an embedding-based term that captures higher-order similarity beyond immediate edges. The second is a bounded-memory state representation consisting of per-community additive statistics, per-vertex light metadata, and sketches for degrees and inter-community affinities. The third is a merge protocol that treats the state as a commutative monoid with conflict resolution rules

Variant	Memory budget (MB)	Relative F1 (%)	Throughput (k edges/s)
Strict-Bound-256	256	94.1	210
Strict-Bound-512	512	97.3	205
Strict-Bound-1024	1024	97.8	192
Adaptive-Bound	512	98.2	200
Unbounded-Baseline	∞	98.5	95

Table 3: Impact of bounded memory on accuracy and throughput.

Cluster setup	Workers	Merge interval τ (s)	Network overhead (%)
Single node	1	–	0.0
Small cluster	4	60	2.3
Medium cluster	8	45	3.8
Large cluster	16	30	5.9
Geo-distributed	16	90	8.7

Table 4: Distributed merge configurations and communication overhead.

Metric	Definition (sketch)	Range	Purpose
Modularity Q	Edge density vs random graph	$[-0.5, 1]$	Global structure quality
Conductance	Cut size / boundary volume	$[0, 1]$	Community separability
NMI	Mutual information of labelings	$[0, 1]$	Ground-truth agreement
F1-score	Harmonic mean of precision/recall	$[0, 1]$	Event-level detection
Stability	Jaccard across windows	$[0, 1]$	Temporal consistency

Table 5: Community quality metrics monitored on the stream.

Method	F1 (%)	Modularity Q	Speedup vs batch ($\times$)
Batch-Louvain	100.0	0.62	1.0
Incremental-Louvain	97.5	0.61	12.3
Label-Propagation-Online	92.4	0.55	18.7
Proposed-Streaming	98.2	0.63	16.9
Proposed-Streaming+DM	98.0	0.63	28.4

Table 6: Comparison with batch and online baselines on Twitter-Stream.

that preserve determinism, allowing eventual consistency across distributed workers. The merge protocol is designed so that the “meaning” of a community is tied to its maintained statistics and hashed identifiers rather than to ephemeral local labels, reducing ambiguity [5].

The emphasis on bounded memory requires careful error accounting. Sketch-based approximations introduce bias and variance, and windowing introduces truncation. In streaming community detection, errors can compound because the partition update uses the approximate statistics to decide future moves. This feedback loop is similar to online learning with approximate gradients. The analysis therefore focuses on stability: sufficient conditions under which approximate update rules remain contractive in a suitable metric, and bounds on quality degradation as a function of sketch error, merge frequency, and stream disorder.

Window size W (edges)	F1 (%)	NMI	Median latency (ms)
10^4	94.3	0.81	4.8
5×10^4	96.8	0.86	7.2
10^5	98.2	0.88	9.5
5×10^5	98.4	0.89	15.7
10^6	98.5	0.89	27.4

Table 7: Sensitivity of incremental detection to sliding-window size.

Configuration	Incremental updates	Distributed merge	F1 (%)
Full model	✓	✓	98.2
No incremental updates	–	✓	94.7
No distributed merge	✓	–	97.6
No pruning	✓	✓	96.1
Static communities	–	–	89.3

Table 8: Ablation study highlighting the contribution of each component.

Operation	Time complexity	Memory footprint	Notes
Edge insertion	$O(\log d)$	$O(1)$	Local neighborhood update
Community reassignment	$O(k)$	$O(k)$	Candidates in vicinity
Window expiration	$O(\log W)$	$O(1)$	Lazy deletion index
Merge across workers	$O(c \log c)$	$O(c)$	c communities per worker
Periodical pruning	$O(c)$	$O(1)$	Threshold-based removal

Table 9: Asymptotic cost of major operations in the streaming algorithm.

The remainder of the paper is organized as follows [6]. We formalize the streaming graph model and objectives, including constraints and multi-objective trade-offs. We then derive bounded-memory incremental update rules and show how to implement them with sketches and low-rank structure. We present the distributed merge operators and prove determinism and consistency properties, as well as quality bounds. We discuss complexity and performance engineering, including storage internals, concurrency, and fault tolerance. Finally, we describe evaluation protocols suitable for streaming and distributed settings, with an emphasis on reproducibility under nondeterminism, and we conclude with limitations and directions [7].

2. Streaming Model, Objectives, and Constraints

Let the evolving graph at logical time t be $G_t = (V_t, E_t)$ with $|V_t|$ potentially increasing over time. The input is a stream of updates $\{x_t\}_{t\geq 1}$, where each update is an event $x_t = (u_t, v_t, s_t, \delta_t, \tau_t)$ representing an interaction between vertices u_t and v_t with strength $s_t \in \mathbb{R}_{>0}$, update type $\delta_t \in \{+1, -1\}$ for insertion or deletion, and event timestamp τ_t. The processing order may not coincide with timestamp order, and the system observes the stream in processing time. The effective edge weight at time t is the aggregation of updates, possibly with temporal decay. To capture time locality and to enforce bounded memory, we define an effective adjacency A_t via either a sliding window of width W in timestamp or an exponential decay with half-life H, both of which yield an implicit reweighting. For the sliding window case, one

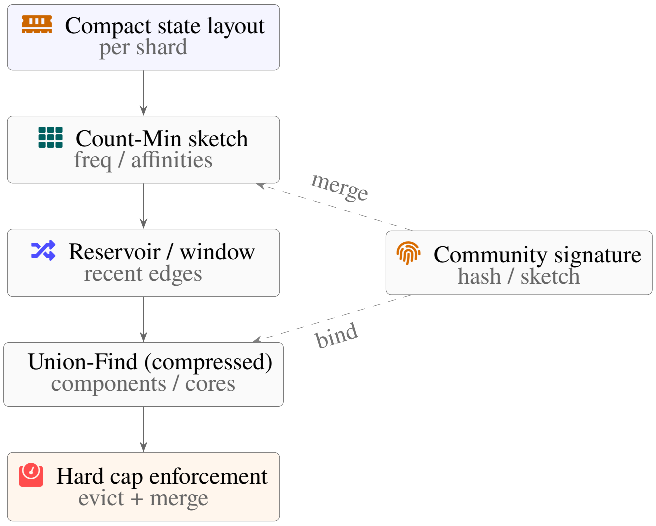

Figure 4: A bounded-memory toolkit for streaming graphs. Sketches approximate heavy hitters and affinities, reservoirs retain a small window of recent edges, and compressed union-find tracks coarse connectivity. Signatures (hashes/sketches) make community state mergeable and comparable across nodes under a hard memory cap.

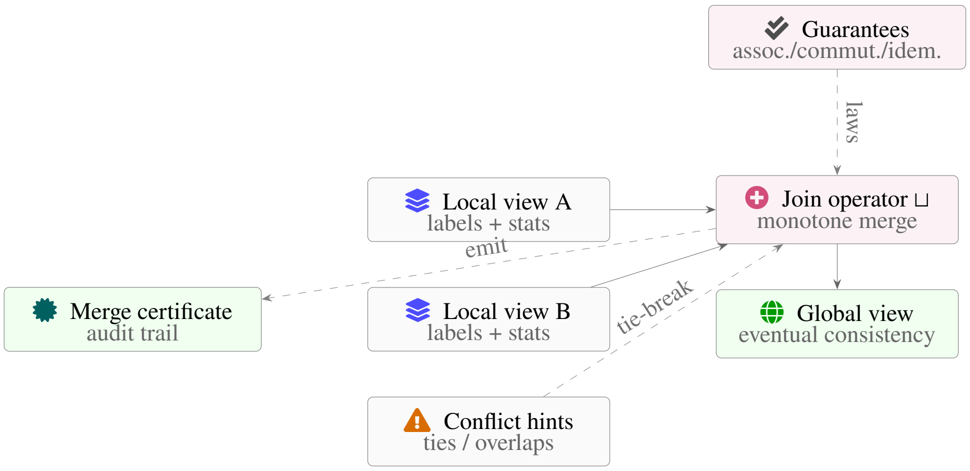

Figure 5: Merge guarantees as an algebraic join over community state. Two replicas contribute compact labeled summaries that are combined by a monotone operator ⊔, yielding an eventually consistent global view. Associativity, commutativity, and idempotence support reordering and retries; optional certificates make merges auditable.

may define

$$A_t(u,v) \;=\; \sum_{i\le t} s_i\,\delta_i\,\mathbf{1}\{(u_i,v_i)=(u,v)\}\,\mathbf{1}\{\tau_t-\tau_i\le W\} \quad . \tag{2.1}$$

For exponential decay with rate $\gamma > 0$, one may define [8]

$$A_t(u,v) \;=\; \sum_{i\le t} s_i\,\delta_i\,\mathbf{1}\{(u_i,v_i)=(u,v)\}\,\exp(-\gamma(\tau_t-\tau_i)) \quad . \tag{2.2}$$

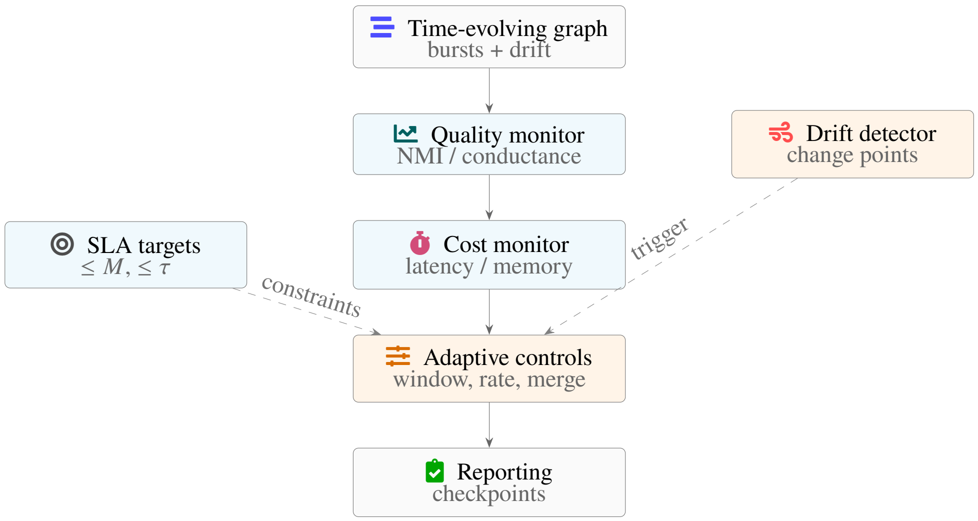

Figure 6: Online evaluation and control loop for streaming community detection. Quality and resource monitors track partition coherence and operational costs; drift detectors trigger parameter adjustments (e.g., window size, admission rate, or merge frequency) to maintain bounded memory and latency while preserving community quality.

The community assignment at time t is a mapping $c_t : V_t \to \{1, \dots, K_t\}$, where K_t may change as communities split and merge. In streaming, it is useful to allow community birth and death, and to consider c_t as a latent variable inferred from the event stream rather than from a static adjacency.

A common batch objective is modularity, which measures the excess of within-community edges over a null model based on degrees. In the streaming setting, degrees and the total edge mass evolve, and under bounded memory one cannot compute exact degrees from full adjacency. Moreover, modularity maximization is combinatorial and known to be computationally hard in worst case. These points motivate a surrogate objective that decomposes into per-community additive terms and can be updated online. We define the following streaming quality functional for an assignment c under effective weights A: [9]

$$Q(c; A) = \sum_{r=1}^{K} \left(m_r - \alpha \frac{\mathrm{vol}_r^2}{2m} \right) - \beta \Omega(c) + \eta \Phi(c) \quad . \tag{2.3}$$

Here $m = \frac{1}{2} \sum_{u,v} A(u,v)$ is the total edge mass, $m_r = \frac{1}{2} \sum_{u,v:c(u)=c(v)=r} A(u,v)$ is internal edge mass for community r, and $\mathrm{vol}_r = \sum_{u:c(u)=r} d(u)$ where $d(u) = \sum_v A(u,v)$ is the weighted degree. The parameter α controls the null-model strength. The term $\Omega(c)$ is a regularizer that penalizes pathological partitions such as extremely fragmented assignments or rapid label churn over time, and $\Phi(c)$ is an embedding-based cohesion term described below. The weights $\beta, \eta \geq 0$ trade off these terms.

To connect to probabilistic modeling, consider a degree-corrected block model with latent community labels $c(u)$ and expected edge weight between u and v given by $\theta_u \theta_v \omega_{c(u),c(v)}$, where θ_u scales degrees and ω is a block affinity matrix. A streaming likelihood for observed edge weights can be written under a Poisson assumption as

$$\log p(A \mid c, \theta, \omega) = \sum_{u<v} \Big(A(u,v) \log(\theta_u \theta_v \omega_{c(u),c(v)}) - \theta_u \theta_v \omega_{c(u),c(v)} \Big) + \mathrm{const} \quad . \tag{2.4}$$

Directly optimizing this is expensive because it involves all pairs [10]. Under sparsity and streaming access, it is more natural to optimize a sampled negative log-likelihood, where non-edges are sampled according to a noise distribution. This yields an online objective resembling contrastive estimation. Let

(u_t, v_t) be an observed edge event and let $\{\tilde{v}_{t,j}\}_{j=1}^{J}$ be J negative samples drawn from a distribution proportional to degree sketches. Define a per-event loss

$$\ell_t(c,\psi) = -\log\sigma\big(\psi(u_t)^\top\psi(v_t)\big) - \sum_{j=1}^{J}\log\sigma\big(-\psi(u_t)^\top\psi(\tilde{v}_{t,j})\big) + \rho\,\mathbf{1}\{c(u_t) \neq c(v_t)\} \quad , \tag{2.5}$$

where $\psi(u) \in \mathbb{R}^k$ is an embedding vector and σ is a logistic function. The last term ties embeddings to communities by penalizing cross-community edges, while embeddings are learned online from the stream. The cohesion term $\Phi(c)$ can be defined using within-community embedding dispersion, for example

$$\Phi(c) = -\sum_{r=1}^{K}\sum_{u:c(u)=r}\big\|\psi(u) - \mu_r\big\|_2^2 \quad , \qquad \mu_r = \frac{1}{n_r}\sum_{u:c(u)=r}\psi(u) \quad . \tag{2.6}$$

This encourages communities to be compact in embedding space and provides a signal beyond immediate edges, which is useful when the stream is bursty or when only a small window is retained [11].

The bounded-memory and systems constraints are treated explicitly. Let M be the memory budget, measured in words, and let L be a latency budget measured as a target average processing time per update. Let B be a bandwidth budget for distributed merges, measured in bytes per unit time. We define a constrained optimization view in which the algorithm chooses a state representation S_t and an update rule to maximize expected quality subject to resource constraints. A convenient scalarization is a Lagrangian [12]

$$\max_{\pi} \quad \mathbb{E}\big[Q(c_t;A_t)\big] - \lambda\,\mathbb{E}\big[\mathrm{mem}(S_t)\big] - \mu\,\mathbb{E}\big[\mathrm{lat}(\pi)\big] - \nu\,\mathbb{E}\big[\mathrm{comm}(\pi)\big] \quad , \tag{2.7}$$

where π denotes the policy consisting of state, update, and merge decisions, and $(\lambda, \mu, \nu) \geq 0$ control the trade-off. This form emphasizes that the design is multi-objective; in practice one may trace a Pareto frontier by varying these multipliers. The theoretical analysis later uses λ, μ, ν to express how approximation parameters such as sketch width, embedding dimension, and merge frequency scale under resource constraints.

Two additional modeling choices are important for streaming. The first is handling deletions and temporal decay, which implies that community statistics must support subtraction and aging. The second is handling vertex churn, where vertices appear and disappear [13]. Under bounded memory, it is typical to keep only an active set of vertices whose recent activity exceeds a threshold, while relegating cold vertices to compressed summaries. The framework therefore distinguishes between active vertices V_t^{hot} and inactive vertices V_t^{cold}, with separate storage policies. The algorithm must still provide consistent merges in the presence of this tiering, which affects the merge design.

Finally, because community detection objectives such as modularity are NP-hard to optimize exactly, it is useful to state a hardness baseline to motivate approximation and heuristics. One can view modularity maximization as a quadratic form over assignments, and show that it generalizes cut objectives. A reduction from Max-Cut can be constructed by setting degrees and null-model terms so that increasing modularity corresponds to separating edges across two groups in a graph constructed from the instance. The key implication for streaming is not the reduction itself, but the practical conclusion that local, approximate moves are standard and that guarantees should be phrased in terms of stability, bounded regret relative to a reference policy, or bounded loss relative to the same heuristic run centrally [14].

3. Bounded-Memory Incremental Detection via Mergeable Sufficient Statistics

The algorithmic goal is to update c_t online as each event arrives, without retaining full adjacency and while supporting later distributed merges. The central design principle is to maintain a set of sufficient statistics that are additive across disjoint substreams and across community unions, so that they can be merged and can support local reassignment decisions. A second principle is to bound state by combining exact tracking for hot vertices with approximate sketches for global quantities. A third principle is to make reassignment decisions depend on local information that can be computed from the maintained summaries, avoiding neighborhood scans.

We represent the community state at time t as a collection $\mathcal{S}_t = \{S_t(r)\}_{r\in C_t}$, where C_t is the set of existing community identifiers and each $S_t(r)$ stores additive statistics. For a community r, the exact additive core statistics are [15]

$$n_r = \sum_u \mathbf{1}\{c(u) = r\} \quad , \qquad \mathrm{vol}_r = \sum_{u:c(u)=r} d(u) \quad , \qquad m_r = \frac{1}{2} \sum_{u,v:c(u)=c(v)=r} A(u,v) \quad . \tag{3.1}$$

Under streaming and bounded memory, degrees $d(u)$ are not stored exactly for all vertices; instead we maintain $d(u)$ exactly for hot vertices and approximately for cold vertices via sketches. Similarly, m_r cannot be updated by scanning all pairs; it is updated by processing edges as they arrive, using the current assignments of endpoints. When an update (u, v) arrives with effective weight increment Δw, we can update

$$m_{c(u)} \leftarrow m_{c(u)} + \tfrac{1}{2}\Delta w\, \mathbf{1}\{c(u) = c(v)\} \quad , \qquad \mathrm{vol}_{c(u)} \leftarrow \mathrm{vol}_{c(u)} + \Delta w \quad , \qquad \mathrm{vol}_{c(v)} \leftarrow \mathrm{vol}_{c(v)} + \Delta w \quad , \tag{3.2}$$

and the global mass $m \leftarrow m + \Delta w$ when appropriate for insertion, with corresponding subtraction for deletions or decay adjustments.

To incorporate embedding cohesion and to allow split/merge heuristics, we also maintain low-order moments of embeddings per community:

$$\Sigma_r^{(1)} = \sum_{u:c(u)=r} \psi(u) \quad , \qquad \Sigma_r^{(2)} = \sum_{u:c(u)=r} \psi(u)\psi(u)^\top \quad . \tag{3.3}$$

From these, the centroid $\mu_r = \Sigma_r^{(1)}/n_r$ and within-community scatter can be computed without storing all member embeddings. Storing $\Sigma_r^{(2)}$ is $O(k^2)$ per community, which may be too large; bounded memory suggests using a diagonal approximation, a low-rank factor, or a sketch such as a CountSketch of outer products. A practical compromise is to keep only $\Sigma_r^{(1)}$ and an exponentially weighted running estimate of average squared norm, enabling an approximate dispersion score

$$\mathrm{disp}_r \approx \sum_{u:c(u)=r} \|\psi(u)\|_2^2 - n_r \|\mu_r\|_2^2 \quad . \tag{3.4}$$

This scalar statistic is additive and supports a cohesion penalty [16].

The core incremental step is a reassignment decision triggered by each event. When an edge (u, v) arrives, the communities of its endpoints are candidates for update. A standard streaming heuristic is to consider moving one endpoint into the other's community if it improves a local objective gain. The gain must be computed using only maintained statistics. For the modularity-like part of Q, if we consider moving a vertex u from community a to community b, the change in internal masses and volumes can be expressed in terms of the weight from u to each community, denoted

$$w(u \to r) \;=\; \sum_{x:c(x)=r} A(u,x) \quad . \tag{3.5}$$

If $w(u \to r)$ were exact, the modularity change would be computable from $(m_a, m_b, \mathrm{vol}_a, \mathrm{vol}_b, m)$ and $w(u \to a), w(u \to b)$. In a stream, exact $w(u \to r)$ requires adjacency access. We therefore maintain an approximate estimator $\widehat{w}(u \to r)$ using a bounded window adjacency cache for hot vertices and a sketch for cold vertices. The cache stores the most recent T neighbors or the neighbors within time window W for hot vertices, allowing exact sums over recent interactions, while older interactions are captured approximately in community-level sketches [17].

A mergeable estimator for $w(u \to r)$ can be built from per-community hashed accumulators. For each community r, maintain a sketch $\mathcal{K}_r$ that maps vertex IDs to approximate incident weight from that vertex into the community. When processing edge (u, v) with Δw and current assignment $c(v) = r$, we update $\mathcal{K}_r[u] \leftarrow \mathcal{K}_r[u] + \Delta w$, and similarly update $\mathcal{K}_{c(u)}[v]$. With Count-Min sketches, this yields an overestimate with bounded additive error depending on sketch width. Because these sketches are additive, they are mergeable across workers. The estimate then is $\widehat{w}(u \to r) = \mathrm{CMSQuery}(\mathcal{K}_r, u)$, optionally corrected by subtracting cached exact contributions to reduce bias for hot vertices.

Given these estimates, an online reassignment rule can be written as selecting b to maximize an estimated gain. For the modularity-like term with coefficient α, an approximate gain for moving u from a to b is [18]

$$\widehat{\Delta Q}_{u:a\to b} \;=\; \widehat{w}(u \to b) \;-\; \widehat{w}(u \to a) \;-\; \alpha\,\frac{d(u)}{2\widehat{m}}\Big(\mathrm{vol}_b - \mathrm{vol}_a\Big) \;-\; \alpha\,\frac{d(u)^2}{2\widehat{m}} \;-\; \beta\,\Delta\Omega \;+\; \eta\,\Delta\Phi \quad . \tag{3.6}$$

The terms $\Delta\Omega$ and $\Delta\Phi$ are regularizer and cohesion changes. A churn regularizer can penalize changing a vertex's label too frequently by maintaining a per-vertex inertia parameter $h(u)$ that decays over time, and setting $\Delta\Omega = h(u)$ when a move occurs. The embedding cohesion change $\Delta\Phi$ can be approximated using community centroids and sizes:

$$\Delta\Phi \;\approx\; [19] - \big\|\psi(u) - \mu_b\big\|_2^2 \;+\; \big\|\psi(u) - \mu_a\big\|_2^2 \quad , \tag{3.7}$$

with μ_a, μ_b computed from $\Sigma^{(1)}$ and n. This term is differentiable in ψ and can be integrated into the embedding update.

The move decision is then performed with a threshold to avoid oscillations under sketch noise. One can use a margin $\gamma_{\mathrm{move}} > 0$ and accept a move only if $\widehat{\Delta Q}_{u:a\to b} > \gamma_{\mathrm{move}}$. The margin can be set based on an upper bound on sketch error so that moves are made only when the estimated gain exceeds potential estimation noise. If a Count-Min sketch with width w and depth d is used, and if the total mass into a community is M_r, then with probability at least $1 - \delta$ the overestimation error is at most ϵM_r where $\epsilon \approx 1/w$ and $\delta \approx e^{-d}$. The move margin can then be chosen proportional to ϵ times an upper bound on the relevant masses, yielding a stability guarantee that false-positive moves are rare [20].

Embedding learning is performed online to capture higher-order similarity and to support split decisions. We adopt a stream-based contrastive update with negative sampling. For each edge event (u, v), we update embeddings via stochastic gradient descent on ℓ_t. Let $z = \psi(u)^\top \psi(v)$ and $z_j = \psi(u)^\top \psi(\tilde{v}_j)$. The gradients are

$$\frac{\partial \ell_t}{\partial \psi(u)} \;=\; -(1 - \sigma(z))\,\psi(v) \;+\; \sum_{j=1}^{J} \sigma(z_j)\,\psi(\tilde{v}_j) \quad , \qquad [21]\,\frac{\partial \ell_t}{\partial \psi(v)} \;=\; -(1 - \sigma(z))\,\psi(u) \quad , \tag{3.8}$$

and similarly for negatives. Updates are

$$\psi(u) \leftarrow \psi(u) - \eta_t \frac{\partial \ell_t}{\partial \psi(u)} \quad , \qquad \psi(v) \leftarrow \psi(v) - \eta_t \frac{\partial \ell_t}{\partial \psi(v)} \quad , \tag{3.9}$$

with step size η_t possibly chosen by an adaptive optimizer. Because embeddings must be stored per vertex, bounded memory requires a policy for cold vertices. A typical policy is to store embeddings only for hot vertices and for community prototypes; when a cold vertex becomes active, its embedding can be reinitialized from its community centroid plus a small random perturbation, or reconstructed from a low-rank projection of feature sketches [22]. This introduces approximation but maintains bounded state.

To reduce embedding memory further, we can constrain embeddings to a low-rank subspace learned online by incremental PCA or Oja's algorithm. Let $U_t \in \mathbb{R}^{k\times r}$ be an orthonormal basis with $r \ll k$, and represent $\psi(u) = U_t y(u)$ with coordinates $y(u) \in \mathbb{R}^r$. Oja's update for U_t using a sample vector g_t derived from gradient directions can be written as

$$U_{t+1} \;=\; \mathrm{orth}\Big(U_t \;+\; \xi_t\, g_t\, y_t^{\top}\Big) \quad , \tag{3.10}$$

where $\mathrm{orth}(\cdot)$ orthonormalizes, and ξ_t is a small step. This yields a low-rank approximation that can be updated in a streaming fashion. The coordinate updates are then performed in $\mathbb{R}^r$ and projected back. Such projections are backprop-friendly because the mapping is linear for fixed U_t, and the orthonormalization can be implemented via incremental QR with bounded cost if r is small [23]. The benefit is a reduction in per-vertex embedding storage from k to r, at the cost of tracking U_t globally, which itself is mergeable if treated as a parameter averaged across workers with deterministic rules, discussed later.

Community birth, death, split, and merge decisions are needed for nonstationary streams. Under bounded memory, these decisions should be local and rely on maintained statistics. A split heuristic can be triggered when a community's dispersion disp_r grows relative to its internal mass m_r or when its conductance proxy degrades. Conductance requires boundary edges, which are not stored exactly, but can be approximated using volume and internal mass: boundary mass is approximately $\mathrm{vol}_r - 2m_r$. A proxy score is

$$\widehat{\phi}_r \;=\; \frac{\mathrm{vol}_r - 2m_r}{\min(\mathrm{vol}_r, 2m - \mathrm{vol}_r)} \quad . \tag{3.11}$$

When $\widehat{\phi}_r$ exceeds a threshold and dispersion is high, a split is considered. The split itself can be implemented by running a small number of k-means steps in embedding space on a sampled subset of member vertices, where the sample is maintained via reservoir sampling per community. Reservoir sampling is mergeable if each sample element is associated with a priority key derived from a hash of vertex ID and a random seed; merging two reservoirs keeps the top-s elements by priority, yielding a deterministic union [24]. This provides a bounded-memory representation of community membership distribution without storing all members.

A merge heuristic is also needed to avoid proliferation of small communities. Two communities r and q are candidates to merge if their embedding centroids are close and if their inter-community affinity is high relative to their volumes. Inter-community affinity can be estimated from sketches that track mass between communities. Maintain an inter-community sketch $\mathcal{M}$ that maps ordered pairs of community IDs to approximate edge mass between them, updated on each edge event by incrementing $(c(u), c(v))$ and $(c(v), c(u))$. Because the number of communities can still be large, the sketch must be compact, for example a CountSketch over pair-hashes. A merge score can be [25]

$$\mathrm{score}(r,q) \;=\; \frac{\widehat{m}_{r,q}}{\sqrt{\mathrm{vol}_r\,\mathrm{vol}_q}} \;-\; \kappa\,\|\mu_r - \mu_q\|_2^2 \quad , \tag{3.12}$$

and a merge is performed if the score exceeds a threshold. When merging, additive statistics are summed:

$$n_{r\cup q} = n_r + n_q \quad , \qquad \mathrm{vol}_{r\cup q} = \mathrm{vol}_r + \mathrm{vol}_q \quad , \qquad m_{r\cup q} = m_r + m_q + \widehat{m}_{r,q} \quad , \tag{3.13}$$

and embedding moments are summed similarly. This additivity is what later enables distributed merge operations.

The bounded-memory condition is enforced by explicitly limiting the number of active vertices, the number of communities, and the size of sketches [26]. Let N_{hot} be the maximum number of hot vertices, $K_{\max}$ be the maximum communities retained with full statistics, and let sketch parameters (w, d) be fixed by budget. When hot vertex capacity is exceeded, vertices are demoted to cold state; their exact neighbor cache is discarded, and only sketch-based degree and community affinity information is retained. When community capacity is exceeded, small communities may be merged into nearest neighbors by centroid, or collapsed into a background community that is not used for fine-grained detection but preserves mass accounting. These policies are expressed as projection operators on the state, applied periodically or when thresholds are crossed, and are designed to preserve mergeability by making the projection deterministic given the state.

Because the objective and the update rule use approximate quantities, error propagation must be considered. Let $\widehat{w}(u \to r) = w(u \to r) + \epsilon_{u,r}$, where $\epsilon_{u,r} \geq 0$ for Count-Min sketches. The move gain error is then a combination of these errors and degree estimate errors. Under a margin-based rule, a false move can occur only if the true gain is within a band around zero whose width is proportional to an error bound [27]. If the algorithm maintains an upper bound $\overline{\epsilon}_{u,r}$ on sketch error, then choosing γ_{move} larger than the maximum possible gain error yields a conservative update that avoids most erroneous moves at the expense of delayed adaptation. This yields a controllable trade-off between stability and responsiveness, which is relevant in nonstationary streams.

Finally, the incremental algorithm must be efficient per update. The reassignment decision should consider only a small set of candidate communities. Using only the endpoints' current communities is too restrictive; using all communities is too expensive. A bounded candidate set can be obtained via a combination of neighbor community caches and locality-sensitive hashing in embedding space. For each hot vertex, maintain a small cache of the most frequent communities among its recent neighbors, updated from the neighbor window [28]. Additionally, maintain an approximate nearest-centroid index using hashing of centroid projections. Let $R \in \mathbb{R}^{p\times r}$ be a random projection matrix with p small, and define signatures $h(r) = \mathrm{sign}(R\mu_r)$. Nearby centroids share signatures with high probability. On a move decision for vertex u, query candidate communities from buckets matching the signature of $\psi(u)$, intersect with neighbor-community cache, and evaluate gain for that small set. These operations are $O(p + C_{\text{cand}})$ with C_{cand} small, and the index is mergeable because it can be rebuilt deterministically from community centroids and a shared projection seed.

4. Distributed Merge Operators and Guarantees

In a distributed ingestion architecture, the stream is partitioned across workers, by vertex, by edge hash, by time, or by upstream sharding. Each worker processes its substream and maintains a local state $\mathcal{S}_t^{(i)}$. Periodically, states are merged to form a global view or to synchronize workers [29]. The merge must be deterministic and, ideally, associative and commutative so that merges can occur in arbitrary order with eventual consistency. The challenge is that community assignments depend on the sequence of updates and on local decisions, so naive merging of partitions yields ambiguity. The approach here is to decouple community identity from local labels by tying identity to mergeable summaries and deterministic tie-breaking, and to express merge as a combination of additive statistic union and reconciliation of vertex-to-community mappings.

We define the state on each worker as the tuple

$$\mathcal{S} = \left(C,\ \{S(r)\}_{r\in C},\ \mathcal{A},\ \mathcal{D},\ \mathcal{E},\ \mathcal{R}\right) \quad , \tag{4.1}$$

where C is the set of community IDs, $S(r)$ are per-community statistics, $\mathcal{A}$ is a mapping for active vertex assignments and metadata, $\mathcal{D}$ is a degree sketch, $\mathcal{E}$ is an inter-community affinity sketch, and $\mathcal{R}$ contains auxiliary structures such as reservoirs, centroid hash indices, and global seeds. For mergeability, the components are designed either to be additive structures (sketches, sums) or to be conflict-free replicated data types with deterministic resolution (maps with last-writer-wins based on timestamps, or sets with deterministic union).

A core requirement is that community IDs are globally meaningful [30]. Local labels like $1, 2, 3$ are not meaningful across workers. We therefore define community IDs as deterministic hashes of community summaries, updated when communities change. Because the true member set is not available under bounded memory, the ID must be derived from a compact fingerprint that is itself mergeable. One option is to maintain a community fingerprint as a commutative hash over member vertex IDs, for example

$$\mathrm{fp}(r) = \bigoplus_{u:c(u)=r} H(u) \quad , \tag{4.2}$$

where H is a fixed hash and $\oplus$ is bitwise XOR [31]. This fingerprint is additive under community union and supports merge across workers if the same vertices are assigned consistently. Inconsistency remains possible, so the fingerprint is not a perfect identifier, but it can be combined with other stable statistics such as centroid hash and volume to reduce collision probability. We define the community ID as

$$\mathrm{id}(r) = H'\big(\mathrm{fp}(r),\ \lfloor \mathrm{vol}_r/\Delta \rfloor,\ h(\mu_r)\big) \quad , \tag{4.3}$$

where H' is a hash, Δ is a volume quantization to stabilize IDs under small noise, and $h(\mu_r)$ is a centroid signature. The quantization is a deliberate bias toward stability, trading responsiveness for merge determinism.

Given two states $\mathcal{S}^{(i)}$ and $\mathcal{S}^{(j)}$, we define a merge operator $\sqcup$ producing $\mathcal{S}^{(i)} \sqcup \mathcal{S}^{(j)}$. For additive components, merge is straightforward: [32]

$$S_{\mathrm{merge}}(r) = S^{(i)}(r) + S^{(j)}(r) \quad , \qquad \mathcal{D}_{\mathrm{merge}} = \mathcal{D}^{(i)} + \mathcal{D}^{(j)} \quad , \qquad \mathcal{E}_{\mathrm{merge}} = \mathcal{E}^{(i)} + \mathcal{E}^{(j)} \quad , \tag{4.4}$$

where $+$ for sketches is coordinate-wise addition, and for statistics is sum of scalars and vectors. The complexity lies in reconciling the sets of communities and vertex assignments. Communities may have different IDs locally due to drift, and vertices may be assigned differently on different workers due to different substreams and approximate decisions.

We address reconciliation by defining a canonicalization step that maps local community IDs to canonical global IDs based on their summaries. Each worker computes $\mathrm{id}(r)$ for each community using the stable hash definition. In merge, communities with identical computed IDs are treated as the same entity and their statistics are added. Communities with different IDs are treated as distinct, but may later be merged by the merge heuristic based on affinity and centroid proximity, which is itself deterministic given merged statistics and shared seeds [33]. This approach avoids the need for label matching as a separate combinatorial step.

Vertex assignment reconciliation is handled by storing for each active vertex u a small record $(c(u), t_u, \pi_u)$ where $c(u)$ is the community ID, t_u is a logical timestamp of the assignment update, and π_u is a priority derived from a hash of $(u, c(u), t_u)$ under a shared seed. Merge resolves conflicts via last-writer-wins on t_u, with ties broken by π_u. This rule yields a deterministic outcome independent of merge order, assuming timestamps are from a consistent logical clock. If processing-time clocks differ, vector clocks are expensive; a practical alternative is to use stream offsets within each shard and include shard ID, producing a total order that is stable under merge but may not reflect real causality [34]. The goal is determinism and eventual convergence rather than perfect causal consistency.

To show algebraic properties, consider the merge operator for assignment records as

$$(u : (c_1, t_1, \pi_1)) \sqcup (u : (c_2, t_2, \pi_2)) = \begin{cases} (c_1, t_1, \pi_1) & \text{if } (t_1, \pi_1) \succ (t_2, \pi_2) \\ (c_2, t_2, \pi_2) & \text{otherwise} \end{cases} , \tag{4.5}$$

where $\succ$ is lexicographic order. This is associative and commutative because it is equivalent to taking the maximum under a total order [35]. Similar max-based merges are used for other per-vertex metadata such as inertia $h(u)$ and hot/cold status, provided they are encoded as monotone states or resolved by max with deterministic tie-break.

However, a subtle issue arises: per-community statistics like n_r and vol_r depend on assignments. If a vertex assignment changes under merge, community counts may become inconsistent if they were simply added. To address this, we distinguish between two layers of statistics. The first layer is event-derived additive statistics that do not depend on assignments, such as degree sketches and raw edge mass summaries. The second layer is assignment-derived statistics that must be recomputed or adjusted after reconciliation. Under bounded memory, full recomputation is impossible, but we can maintain a correction mechanism based on active vertices and approximate cold contributions [36]. Specifically, we maintain per-community assignment-derived statistics for the active set exactly, and for cold vertices approximately via background mass. After a merge, we apply a deterministic reconciliation pass over the merged active vertex map, updating community n_r^{hot}, $\mathrm{vol}_r^{\mathrm{hot}}$, and embedding moments based on the final assignment of each active vertex. Cold statistics are updated using additive sketches that map vertices to communities via their last known assignment; errors are bounded by the fraction of mass in cold vertices and by the frequency of cold assignment updates.

Formally, write $\mathrm{vol}_r = \mathrm{vol}_r^{\mathrm{hot}} + \mathrm{vol}_r^{\mathrm{cold}}$. The hot component is exact from active records. The cold component is approximated as

$$\widehat{\mathrm{vol}}_r^{\mathrm{cold}} = \sum_{u \in V^{\mathrm{cold}}} \widehat{d}(u)\, \mathbf{1}\{\widehat{c}(u) = r\} , \tag{4.6}$$

where $\widehat{d}(u)$ is from the degree sketch and $\widehat{c}(u)$ is a stored last-known community for cold vertices, which may lag. The merge operator for cold assignment maps is again max-based on timestamps [37]. Because the cold map can be large, it is itself stored as a sketch or as a compact dictionary for a bounded set of recently demoted vertices, with older cold vertices folded into per-community aggregates. This yields a bounded-memory approximation in which exact reconciliation is limited to hot vertices, consistent with the resource model.

The distributed merge guarantee is phrased as two properties. The first is state convergence: repeated merges of worker states in any order yield the same merged state, up to stochastic elements fixed by shared seeds. The second is quality stability: the partition induced by the merged state has objective value within a bounded gap of a reference partition produced by a centralized run of the same algorithmic policy on the union stream, under assumptions about sketch error and merge frequency. The first property is obtained by designing mergeable components with commutative monoid structure and deterministic conflict resolution [38]. The second requires bounding the effect of delayed synchronization and approximate reconciliation.

For convergence, define the state space $\mathbb{S}$ and a partial order $\preceq$ representing "information inclusion," where sketches and additive counters are monotone and max-based maps are monotone in their timestamp order. The merge operator $\sqcup$ is then a least upper bound in this semilattice. For additive sketches, the least upper bound is coordinate-wise addition, which is commutative and associative. For max-based maps, it is coordinate-wise maximum. For structures like reservoirs with top-s priorities, the merge is selecting the top-s by a total order, which is again associative and commutative [39]. Therefore, for these components, $\sqcup$ yields convergence.

The remaining nontrivial part is the community ID canonicalization, since IDs depend on summaries that change under merges. To make this stable, IDs are computed from quantized summaries and fingerprints. Under a merge, the summary changes by addition, which changes the quantized values predictably. If the ID is recomputed after merge, then a community's ID may change. To prevent instability, we treat the computed ID as an internal fingerprint and maintain a persistent canonical ID determined by a union-find-like structure over fingerprints [40]. Specifically, when two communities are merged by heuristic or by ID collision, we create a canonical representative ID as the minimum under a total order of the two IDs, and store a mapping from the other to the representative. This mapping is a grow-only set of alias relations, which is mergeable by union. Resolution of a community ID then follows alias links to the representative. Because unions only add alias relations and representatives are chosen deterministically, the resolution is eventually consistent. The cost is that community IDs may not reflect current statistics, but stability is improved and merges remain deterministic.

For quality stability, define the centralized reference policy $\pi^\star$ that processes the union stream in some canonical order and produces a partition $c_t^\star$. The distributed system processes shards and merges every T_{merge} events per worker, yielding c_t^{dist}. The difference arises from two sources: local decisions made without seeing remote edges until merge, and approximation error in sketches and cold reconciliation [41]. Let $\Delta_t = Q(c_t^\star; A_t) - Q(c_t^{\text{dist}}; A_t)$. A typical bound structure is

$$\Delta_t \;\le\; C_1\,\epsilon_{\text{sketch}}\,m_t \;+\; C_2\,\epsilon_{\text{cold}}\,m_t \;+\; C_3\,\epsilon_{\text{delay}}(T_{\text{merge}})\,m_t \;+\; C_4\,\epsilon_{\text{embed}}\,m_t \quad , \tag{4.7}$$

where m_t is total mass and the constants depend on objective parameters. The sketch term ϵ_{sketch} scales with sketch width, the cold term ϵ_{cold} with the fraction of mass involving cold vertices and the staleness of cold assignments, the delay term ϵ_{delay} with merge interval and the rate of cross-shard edges, and the embedding term with the rank truncation and SGD noise. The bound is qualitative unless assumptions are specified [42]. Under a stationary model such as a block model where cross-shard edges are random and merge intervals are moderate, one can treat delayed edges as a perturbation to local update decisions and show that the expected number of "wrong" moves per interval is proportional to the cross-shard mass not yet incorporated, leading to ϵ_{delay} proportional to that mass fraction. Under a margin-based move rule, many moves are conservative and thus less sensitive to perturbations.

A complementary view is regret relative to a class of local policies. Define a per-event surrogate gain g_t that the policy seeks to maximize, and let the distributed policy have approximate gain $\widehat{g}_t$ due to sketches and delay. Under Lipschitz conditions, one can bound the cumulative regret over T events:

$$\sum_{t=1}^{T} \big(g_t(\pi^\star) - g_t(\pi^{\text{dist}})\big) \;\le\; L_g \sum_{t=1}^{T} \big\|\widehat{\nabla} - \nabla\big\| \;+\; \text{stability terms} \quad , \tag{4.8}$$

where $\widehat{\nabla}$ denotes approximate gradients or approximate gain components. Here the stability terms reflect the fact that decisions affect future states; using an inertia regularizer and move margins can make the system more stable and reduce regret amplification.

Finally, the distributed merge protocol must support fault tolerance and repeated merges [43]. Because the merge operator is associative and commutative, it supports state replication and recovery: a worker can checkpoint its local state and recover by replaying or by merging with a replica. To avoid double-counting event-derived additive statistics when the same shard is merged twice, each additive component must be versioned by shard epochs. This is achieved by maintaining per-shard counters or sketches with identifiers and merging via a map from shard ID to its latest epoch state, resolved by max on epoch and then replaced rather than added. This transforms addition into a form of observed-remove semantics, which is still mergeable but requires careful design. In practice, a bounded number of recent epochs can be stored and older epochs compacted, consistent with bounded memory.

5. Complexity, Data Structures, and Performance Engineering

Streaming community detection under bounded memory is fundamentally a performance engineering problem as much as an algorithmic one [44]. The per-update critical path must fit a latency budget, memory accesses must be predictable, and distributed merges must not overwhelm bandwidth. This section analyzes complexity at the level of asymptotic bounds and at the level of storage layout and concurrency, since constant factors dominate in streaming systems.

Consider the incremental update per edge event. The minimal operations include updating degree estimates, updating per-community sketches for endpoint-to-community weights, updating internal mass if endpoints are co-assigned, updating embeddings for active endpoints, and optionally evaluating a move. Let C_{cand} be the number of candidate communities evaluated for a move. Each candidate evaluation uses a small number of sketch queries and arithmetic operations [45]. With Count-Min sketches of depth d, each query is $O(d)$ hash operations and memory reads. Therefore, the per-event time is

$$T_{\text{event}} = O\big(d + d\,C_{\text{cand}} + rJ + r\big) \quad , \tag{5.1}$$

where r is embedding dimension (possibly low-rank), and J is the number of negative samples. The rJ term comes from dot products with negatives. If embeddings are only updated for hot vertices, and if hotness is bounded, then the total embedding storage is $O(N_{\text{hot}}r)$. The sketch storage is $O(wd)$ per sketch. If we maintain per-community sketches $\mathcal{K}_r$, then storage would be $O(Kwd)$, which may be large; a more memory-efficient approach is to maintain a global sketch keyed by (r, u) pairs, effectively storing all per-community maps in a single sketch. This yields storage $O(wd)$ independent of K but increases collision noise [46]. A hybrid approach is to store exact maps for the largest communities and use a global sketch for the rest.

The bounded-memory design hinges on selecting which quantities are exact and which are sketched. Exact per-community scalars such as n_r and vol_r are small if K is bounded. Per-community centroids require storing r floats per community, giving $O(Kr)$ memory. If K is in the tens of thousands and r is tens, this is feasible; if K is larger, centroid storage becomes heavy and should be compressed, for example by quantizing centroids to 16-bit floats or by storing them in a product-quantized codebook. Quantization introduces error in centroid similarity and merge heuristics; it can be analyzed as an additional noise term in the merge score [47]. The impact is reduced if merge decisions use thresholds and if signatures are based on random projections that tolerate small perturbations.

A key data structure is the active vertex map $\mathcal{A}$, which stores assignment, embedding coordinates, inertia, and small neighbor cache. This map must support frequent updates and queries by vertex ID. A hash table is typical, but high update rates can cause cache misses. Performance improves with open addressing and contiguous storage, or with a two-tier map where the hottest vertices are stored in a fixed-size array indexed by a compacted ID, while less active vertices are stored in a hash table. Vertex IDs from the input are arbitrary, so compacting requires maintaining an ID translation table, which itself consumes memory. Under bounded memory, one can instead use cuckoo hashing with a fixed capacity and evict on overflow, demoting evicted vertices to cold state [48]. Eviction must be deterministic to preserve distributed merge behavior; deterministic eviction is achieved by evicting the vertex with smallest activity score, with ties broken by hash priority under a shared seed.

The neighbor cache for hot vertices is another major memory consumer if stored as explicit adjacency lists. Bounded memory suggests keeping only a fixed-size ring buffer of recent neighbors per hot vertex. This yields $O(N_{\text{hot}}T)$ memory, where T is cache length. Each insertion pushes out an old neighbor. This cache supports local candidate community discovery and exact recent weight computation. For weighted edges, each cache entry stores neighbor ID, weight, and timestamp or decay factor [49]. To avoid per-entry timestamps, one can store only processing-time sequence numbers and approximate decay, which is cheaper but less accurate under disorder.

Sketch data structures must be engineered for high throughput. Hash computations are expensive; using a small number of fast hashes and deriving multiple indices from one 64-bit hash via tabulation reduces cost. Memory layout matters: storing sketch arrays in contiguous memory improves prefetching. Count-Min depth d is often small, such as $d = 3$ or $d = 4$, to balance error and cost [50]. The width w is chosen based on acceptable error. For inter-community pair sketches, hashing a pair (r, q) can be done via mixing the hashes of r and q, with care to avoid symmetry issues.

The embedding update is arithmetic-heavy but memory-local if embeddings are stored contiguously and if negative samples are drawn from a cache. Negative sampling requires access to a degree-proportional distribution. Under bounded memory, we cannot store exact alias tables for all vertices; instead we can sample negatives from a rolling reservoir of hot vertices weighted by approximate degree, or from a hash-based sampler that maps random numbers to vertices using a sketch of cumulative degree in buckets. A simple approach is to maintain buckets of vertices by hashed ID and maintain approximate total degree per bucket [51]. Sampling then picks a bucket proportional to its degree mass and then picks a vertex within the bucket uniformly or by a small local alias table. This yields approximate degree-proportional sampling with bounded memory and is mergeable if bucket masses are additive and if within-bucket selection uses deterministic seeds.

Distributed merge cost is driven by the size of state transmitted. Sending full active maps and all sketches frequently is expensive. Merge frequency therefore trades off delay error and bandwidth. A practical approach is hierarchical merging: workers send only deltas of additive sketches and only changes in active assignments since last merge [52]. Deltas are identified by version vectors or by per-worker epoch counters. The merge operator on deltas must avoid double counting; this is achieved by including epoch IDs and storing last-applied epoch per sender. For max-based assignment records, sending only changed records since last merge is natural. For additive sketches, sending deltas is possible by storing a copy of the sketch at last merge and subtracting, but subtraction requires signed counters; Count-Min is nonnegative, so deltas can be represented as signed sketches like CountSketch or as separate positive and negative delta sketches. Alternatively, one can use time-windowed sketches that naturally expire, reducing the need for deltas [53].

Concurrency control is crucial in a high-throughput streaming system. Updates arrive concurrently, and the state must be updated without locking bottlenecks. One strategy is to partition state by vertex hash and process updates in a sharded executor so that updates touching different shards proceed independently. Edges touch two vertices, potentially two shards. To avoid cross-shard locks, the system can assign each edge to a canonical shard based on a hash of its endpoints and update only that shard's state, treating the other endpoint as a remote reference whose metadata is accessed via a shared read-only snapshot. This introduces staleness but improves throughput [54]. Another strategy is to use lock-free atomic updates for sketches and additive counters while serializing assignment changes per vertex to avoid oscillations. Because the algorithm already tolerates approximation and delay, a limited amount of staleness is acceptable, and the merge guarantees focus on determinism given the observed update order within each shard.

Batching is a standard optimization. Instead of updating sketches and embeddings per edge, events are buffered for a short time and processed in micro-batches. Batching reduces overhead and allows vectorized operations on embeddings and sketch indices. However, batching increases latency [55]. The latency constraint in the Lagrangian can be interpreted as limiting batch size. A multi-objective selection can be expressed as choosing batch size b to maximize quality minus a latency penalty. If quality degrades with batch size because updates are delayed, and if processing cost per event decreases with batch size due to amortization, then one can choose b by minimizing a convex surrogate

$$\min_{b\geq 1} \; \mu\,\mathrm{lat}(b) + \lambda\,\mathrm{mem}(b) + \nu\,\mathrm{comm}(b) - \mathbb{E}[Q(b)] \quad , \tag{5.2}$$

where $\mathrm{lat}(b)$ increases with b and processing cost decreases. In practice, this is tuned empirically, but the formulation clarifies trade-offs.

Finally, a bounded-memory streaming system must address garbage collection of communities and vertices [56]. Communities that become inactive should be compacted or merged. Vertices that have not been seen recently should be demoted to cold and eventually folded into aggregate statistics. These operations must be deterministic to support merge consistency. Determinism is obtained by using timestamp thresholds and hash-based priorities. For example, if a community has not received mass above a threshold for a time window, it is marked inactive and merged into a background community in order of increasing ID. Such deterministic policies ensure that two replicas under the same observed stream converge even if cleanup happens at slightly different times, because the cleanup rule depends only on state and shared thresholds [57].

6. Error Analysis, Approximation Bounds, and Optimization Under Resource Budgets

Approximation is unavoidable under bounded memory: sketches introduce estimation error, windowing truncates history, low-rank embeddings approximate representation, and distributed merges introduce delay and reconciliation artifacts. This section provides an analytical account of how these errors enter the update rules and how resource parameters control them. The goal is not to claim tight worst-case bounds, which are typically pessimistic, but to provide explicit dependencies that guide parameter selection and to identify stability mechanisms.

Consider the modularity-like part of Q and a move decision for vertex u from a to b. Let the true gain in this part be Δ and the estimated gain be $\widehat{\Delta}$. The estimation error decomposes into edge-to-community weight errors and degree/volume errors. Write

$$\widehat{\Delta}-\Delta \;=\; (\widehat{w}(u\to b)-w(u\to b)) \;-\; (\widehat{w}(u\to a)-w(u\to a)) \;-\; \alpha\,\frac{\widehat{d}(u)}{2\widehat{m}}(\widehat{\mathrm{vol}}_b-\widehat{\mathrm{vol}}_a) \;+\; \alpha\,\frac{d(u)}{2m}(\mathrm{vol}_b-\mathrm{vol} \tag{6.1}$$

Count-Min sketch errors for $\widehat{w}$ are nonnegative and bounded in expectation by ϵ times the total mass inserted into the sketch, with high-probability tail bounds. Degree sketch errors have similar forms [58]. Volume errors arise from cold reconciliation and merge delay. Under small relative errors, the volume term can be linearized. Let $\widehat{d}(u)=d(u)(1+\delta_d)$, $\widehat{m}=m(1+\delta_m)$, and $\widehat{\mathrm{vol}}_r=\mathrm{vol}_r(1+\delta_r)$. Then the difference in the volume contribution is approximately

$$-\alpha\,\frac{d(u)}{2m}\Big((\delta_d-\delta_m)(\mathrm{vol}_b-\mathrm{vol}_a)+\delta_b\mathrm{vol}_b-\delta_a\mathrm{vol}_a\Big)\quad, \tag{6.2}$$

which shows that relative errors in global mass and volumes scale the gain error proportionally to $d(u)\mathrm{vol}/m$. In sparse graphs with heavy-tailed degrees, high-degree vertices are more sensitive; this motivates treating high-degree vertices as hot for longer, storing their degrees and recent neighbors more accurately.

Move stability is enforced by the margin γ_{move}. Let $E_{u,a,b}$ be an upper bound on $|\widehat{\Delta}-\Delta|$ for a candidate move. If $\gamma_{\mathrm{move}}\ge E_{u,a,b}$, then any accepted move has nonnegative true gain in the modularity part, excluding the effect of regularizers and cohesion terms. In practice $E_{u,a,b}$ is not known exactly, but can be bounded by sketch error bounds and by crude bounds on volume terms. For Count-Min with total inserted mass M, width w, and depth d, one has a bound of the form

$$\widehat{w}(u\to r)\;\le\;w(u\to r)\;+\;\epsilon M\quad\text{with probability at least }1-\delta\quad,\qquad \epsilon\approx\frac{e}{w}\quad,\qquad \delta\approx e^{-d}\quad. \tag{6.3}$$

Then $E_{u,a,b}$ can be set to a multiple of ϵM plus the estimated error in volume terms. This yields a conservative but explicit rule [59].

Windowing introduces truncation. In a sliding window of width W, edges older than W do not contribute. If the true underlying community structure changes over time, this can be beneficial because it focuses on recent structure. If structure is stable, truncation loses signal. The error can be modeled by comparing A_t under full history and under windowing. If edge weights decay with characteristic time $1/\gamma$, then using a window of width W approximates exponential decay when W is several half-lives [60]. The truncation error in total mass is bounded by the mass of discarded edges, which can be estimated from stream rate. This yields a resource-quality relationship: smaller W reduces memory and increases responsiveness, but increases variance and bias in degree and affinity estimates.

Low-rank embeddings introduce approximation error in representing similarity. If the embedding objective is effectively capturing a pointwise mutual information matrix factorization, then the best rank-r approximation error is governed by singular values. Let M be a matrix whose entries encode co-occurrence or random-walk similarity, and let M_r be its best rank-r approximation [61]. Then the Frobenius norm error is

$$\|M - M_r\|_F^2 = \sum_{i>r} \sigma_i^2 \quad , \tag{6.4}$$

where σ_i are singular values. In a streaming setting, one estimates M implicitly via gradients; the truncation error manifests as an inability to separate communities that differ in directions beyond rank r. Increasing r improves representation but increases memory and compute. Because community detection may not require fine-grained embedding detail, r can be smaller than typical representation learning embeddings, especially when combined with modularity-like statistics.

Distributed merge delay can be analyzed by treating the global state as a parameter vector updated asynchronously [62]. Suppose each worker performs local updates producing a local community summary vector θ_i, and merges perform a deterministic aggregation $\theta = \mathcal{M}(\{\theta_i\})$. If the update rule is a form of stochastic gradient descent on a smooth surrogate loss $L(\theta)$, then asynchronous SGD analyses suggest that delay introduces an error proportional to the maximum staleness and the gradient Lipschitz constant. While community detection updates are not purely gradient-based, the embedding updates are, and the partition updates can be seen as proximal steps on a discrete variable with inertia. A qualitative bound is that quality loss increases with merge interval and with the rate of cross-worker dependencies, and decreases with move margins and regularization that prevents rapid oscillations.

Resource budgets are incorporated via the Lagrangian multipliers (λ, μ, ν). Parameter selection can be expressed as choosing sketch width w, embedding rank r, hot set size N_{hot}, cache length T, and merge interval T_{merge} to minimize a surrogate of the bound on quality loss plus penalties. A stylized optimization is

$$\min_{w,r,N_{\text{hot}},T,T_{\text{merge}}} \quad C_1 \frac{1}{w} \; + \; C_2 e^{-c_2 N_{\text{hot}}} \; + \; C_3 \frac{1}{r^{\zeta}} \; + \; C_4 \, T_{\text{merge}} \; + \; \lambda \left(N_{\text{hot}} r + K r + w d + N_{\text{hot}} T \right) \; + \; \mu \left(d C_{\text{cand}} + r J \right) \tag{6.5}$$

where the first group models error terms with illustrative scaling, and the second group models resource costs [63]. The term $e^{-c_2 N_{\text{hot}}}$ is a placeholder for the notion that keeping more vertices hot reduces cold assignment staleness and improves accuracy. The exponent ζ captures singular value decay. While these functional forms are approximate, the expression clarifies how increasing w reduces sketch error at linear memory cost, increasing r reduces representation error at linear memory and compute cost, increasing N_{hot} improves accuracy but increases memory and cache cost, and decreasing T_{merge} reduces delay error but increases communication overhead.

The optimization can be solved heuristically by selecting a feasible region given hard memory and latency constraints, then tuning parameters to balance error contributions. Multi-objective optimization can also be expressed via Pareto optimality: a configuration is Pareto optimal if no other configuration improves quality without worsening at least one resource. In practice, one can explore a small set of configurations by varying w and r logarithmically and choosing N_{hot} and T from memory, then selecting T_{merge} from bandwidth. The framework here provides a principled vocabulary for that tuning and a way to interpret results.

7. Evaluation Methodology and Reproducibility in Streaming and Distributed Settings

Evaluating streaming community detection differs from batch evaluation because the target is time-dependent, the algorithm is approximate, and distributed execution introduces nondeterminism [64]. A methodology must therefore define what is measured, at what times, and under what execution conditions. The objective is to characterize accuracy-resource trade-offs and merge behavior rather than to report a single static score.

A first decision is the notion of ground truth. In many real streams, ground truth communities are not available. Synthetic generators with evolving block structure provide controllable ground truth. A useful generator is a time-varying degree-corrected block model where community memberships change according to a Markov process, edge rates depend on current memberships, and degrees follow a heavy-tailed distribution [65]. The stream is generated by sampling interactions over time, with optional bursts and deletions. Ground-truth labels are then the latent memberships at each time. For real data, proxy ground truth can come from known attributes or from downstream task performance, but such proxies should be interpreted cautiously.

A second decision is the evaluation metric. Partition similarity metrics such as normalized mutual information and adjusted Rand index compare detected labels to ground truth, but they require aligning labels across time [66]. In streaming, label churn is itself a metric: a partition that changes excessively may be undesirable even if instantaneous similarity is high. Therefore, evaluation should include temporal smoothness measures, such as the fraction of vertices whose community assignment changes per unit time, and the stability of large communities. Additionally, objective-based measures such as modularity proxy Q can be tracked over time to assess whether the algorithm maintains quality under drift and merges. For distributed merges, one should measure divergence between worker-local partitions and the merged partition, and convergence speed after merges.

Resource usage must be measured alongside accuracy. Memory usage is measured as the size of active maps, caches, and sketches, including overhead. Latency is measured as processing time per event, including tail latency under bursts [67]. Communication is measured as bytes per merge and bytes per unit time. Because streaming systems experience backpressure, it is important to measure throughput at a fixed latency budget and to report degradation under overload conditions. The Lagrangian formulation suggests reporting a family of configurations along a Pareto curve rather than a single configuration.

Distributed evaluation requires defining sharding strategies and merge schedules. Different sharding patterns affect cross-worker dependencies. Vertex-based partitioning tends to localize edges but can be skewed under power-law degrees [68]. Edge-hash partitioning balances load but increases cross-worker vertex overlap. Time-based partitioning induces strong delay effects. Evaluation should therefore include multiple sharding patterns to test merge robustness. Merge interval T_{merge} should be varied to observe how quality degrades with delay and how bandwidth scales. Fault injection can test merge idempotence and double-counting defenses by simulating worker restarts and repeated merges.

Reproducibility is challenging because sketches, hashing, and asynchronous execution can introduce nondeterminism [69]. The approach in this paper is to enforce determinism through shared seeds and deterministic tie-breaking, but micro-level nondeterminism can still occur due to thread scheduling and floating point ordering. To improve reproducibility, one should fix random seeds for hashing, negative sampling, and reservoir priorities; use deterministic floating point reduction order where feasible; and log merge epochs and event offsets. Checkpointing should capture all state components, including sketch arrays, centroid indices, alias maps for community IDs, and hot/cold thresholds. Replay should be supported: given the same event stream order and the same seeds, the algorithm should produce identical results up to floating point rounding. In practice, slight differences can occur, so evaluation should include confidence intervals over multiple runs with controlled perturbations.

A careful evaluation should also separate the contributions of different components [70]. Because the framework combines modularity-like statistics, embedding cohesion, sketches, and distributed merges, ablation studies clarify which parts contribute under which conditions. Under bounded memory, ablations must be resource-matched; for example, if embeddings are removed, the freed memory might be allocated to larger sketches, changing error behavior. Resource-matched comparisons are therefore more informative than feature-matched ones.

Finally, evaluation should include queries that the system is intended to support. Community detection is often used to answer queries such as “which community does this vertex belong to now,” “which communities are strongly connected," or "which communities are emerging." Under bounded memory, the system may return approximate answers. Therefore, evaluation should include query accuracy and query latency, and error should be measured relative to a centralized baseline [71]. For inter-community connectivity queries, sketch error bounds can be empirically measured by comparing sketch estimates to exact counts on a sampled subset where adjacency is retained. For membership queries, the rate of mis-assignment for hot vertices can be measured separately from cold vertices, reflecting the tiered storage policy.

8. Conclusion

This paper has presented a framework for incremental community detection on streaming graphs under bounded memory, with an emphasis on distributed merge guarantees. The central idea is to treat community detection as an online inference and optimization problem driven by event streams, where the maintained state is a set of mergeable sufficient statistics rather than an explicit graph. A modularity-inspired quality functional was combined with a probabilistic regularization perspective and an embedding-based cohesion term, yielding update rules that can be implemented with per-event bounded computation [72]. Bounded memory was achieved by tiering vertices into hot and cold sets, using fixed-size neighbor caches for recent exact locality, and using mergeable sketches to approximate degrees, endpoint-to-community affinities, and inter-community mass. Low-rank embedding maintenance and reservoir sampling were used to support split and merge heuristics without storing full membership lists.

For distributed ingestion, the design focused on deterministic, associative, and commutative merge operators over state components, using additive statistics, max-based conflict resolution, and stable community identifiers derived from mergeable fingerprints and quantized summaries. Because assignment-derived statistics can become inconsistent under naive addition, a reconciliation pass over the bounded active set was incorporated, with approximate handling for cold vertices. Merge guarantees were expressed as eventual convergence of state and bounded quality loss relative to a centralized reference policy, with dependencies on sketch error, merge delay, and cold-state staleness. The analysis emphasized stability mechanisms such as move margins and inertia penalties, which reduce sensitivity to approximation and delay [73].

The framework leaves open practical and theoretical limitations. Under extreme churn, cold-state approximation can dominate. Under adversarial streams, sketch collisions and delayed merges can induce persistent errors despite margins. Embedding learning introduces additional tuning and can be sensitive to negative sampling approximations. These limitations suggest further work on adaptive sketch sizing under detected error, on more robust merge reconciliation for highly overlapping shards, and on tighter stability analyses for discrete assignment updates coupled to approximate statistics. Within the bounded-memory and distributed constraints, the presented approach provides a coherent set of modeling, algorithmic, and systems mechanisms for tracking community structure in evolving graphs while supporting deterministic distributed merges [74].

References

[1] V. D. Maio, A. Aral, and I. Brandic, ''A roadmap to post-moore era for distributed systems,'' in *Proceedings of the 2022 Workshop on Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*, pp. 30–34, ACM, 7 2022.

[2] S. Zhu, ''Chat application with distributed system,'' 5 2020.

[3] A. Rogan, A. Kolar-PoŻun, and G. Kosec, ''A numerical study of combining rbf interpolation and finite differences to approximate differential operators,'' in *2025 MIPRO 48th ICT and Electronics Convention*, pp. 1177–1182, IEEE, 6 2025.

[4] K. R. V. Dame, T. B. Bergmann, M. Aichouri, and M. Pantoja, *A Comparative Study of Consensus Algorithms for Distributed Systems*, pp. 120–130. Germany: Springer International Publishing, 4 2022.

[5] Y. Song, Z. Mi, H. Xie, and H. Chen, ‘‘Powerinfer: Fast large language model serving with a consumer-grade gpu,’’ in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 590–606, ACM, 11 2024.

[6] M. K. Aguilera, *ApPLIED@PODC - Apply or Perish.* ACM, 7 2018.

[7] F. Loisel, G. Zeqo, A. Morichetta, A. Lackinger, and S. Dustdar, ‘‘Raincloud: Decentralized coordination and communication in heterogeneous iot swarms,’’ in *2024 International Symposium on Parallel Computing and Distributed Systems (PCDS)*, pp. 1–10, IEEE, 9 2024.

[8] A. Davydenko, ‘‘Ensuring the order of message processing in distributed systems,’’ *NaUKMA Research Papers. Computer Science*, vol. 7, pp. 58–62, 5 2025.

[9] 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.

[10] 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.

[11] H. M. Zangana, N. Y. Ali, and S. R. M. Zeebaree, ‘‘Transforming public management,’’ *Indonesian Journal of Education and Social Sciences*, vol. 4, pp. 36–46, 1 2025.

[12] C. Herbert, V. Marschin, B. Erb, D. Meißner, M. Aufheimer, and C. Bösch, ‘‘Are you willing to self-disclose for science? effects of privacy awareness and trust in privacy on self-disclosure of personal and health data in online scientific studies-an experimental study.,’’ *Frontiers in big data*, vol. 4, pp. 763196–, 12 2021.

[13] G. Hu, Z. Wang, C. Tang, J. Shen, Z. Dong, S. Yao, and H. Chen, ‘‘Webridge: Synthesizing stored procedures for large-scale real-world web applications,’’ *Proceedings of the ACM on Management of Data*, vol. 2, pp. 1–29, 3 2024.

[14] F. Lai, P. Zhang, R. Cheng, and P. Xu, ‘‘Distributed systems anomaly detection based on log,’’ in *2021 International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, vol. 14, pp. 72–79, IEEE, 12 2021.

[15] W. B. Daszczuk, *Integrated Model of Distributed Systems*, pp. 31–48. Germany: Springer International Publishing, 3 2019.

[16] S. Sakr and A. Y. Zomaya, *Distributed Systems*, pp. 690–690. Springer International Publishing, 2 2019.

[17] E. P. G. Correia, ‘‘Distributed system tests framework,’’ 7 2019.

[18] M. Wu, L. Mao, Y. Lin, Y. Jin, Z. Li, H. Lyu, J. Tang, X. Lu, H. Tang, D. Dong, H. Chen, and B. Zang, ‘‘Jade: A high-throughput concurrent copying garbage collector,’’ in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 1160–1174, ACM, 4 2024.

[19] H. Zhang, R. Chen, Z. Tang, K. Cheng, and H. Chen, ‘‘Accelerating million-scale in-network lock management using lock fission,’’ *ACM Transactions on Computer Systems*, 11 2025.

[20] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, ‘‘Greybox fuzzing of distributed systems,’’ in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1615–1629, ACM, 11 2023.

[21] T. Pusztai, S. Nastic, P. Raith, S. Dustdar, D. Vij, and Y. Xiong, ‘‘Vela: A 3-phase distributed scheduler for the edge-cloud continuum,’’ in *2023 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 161–172, IEEE, 9 2023.

[22] P. Raith, T. Rausch, A. Furutanpey, and S. Dustdar, ‘‘<i>faas-sim</i>: A trace-driven simulation framework for serverless edge computing platforms,’’ *Software: Practice and Experience*, vol. 53, pp. 2327–2361, 10 2023.

[23] S. Shankar, ‘‘The canonical controller for distributed systems,’’ *Multidimensional Systems and Signal Processing*, vol. 32, pp. 303–311, 8 2020.

[24] A. K. Chopra, S. H. C. V, and M. P. Singh, *Multiagent Foundations for Distributed Systems: A Vision*, pp. 62–71. Germany: Springer International Publishing, 3 2022.

[25] 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.

[26] I. Schagaev and S. Farrell, *Distributed Systems: Resilience, Desperation*, pp. 343–368. Springer International Publishing, 7 2024.

[27] R. Gorantla, S. S. P. D. Pantham, and H. Venkatesh, *Treasure Hunt on Web-Predicated Distributed Systems*, pp. 89–99. Book Publisher International (a part of SCIENCEDOMAIN International), 5 2022.

[28] L. Pick, A. Desai, and A. Gupta, ‘‘Psym: Efficient symbolic exploration of distributed systems,’’ *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 660–685, 6 2023.

[29] S. Deng, H. Zhao, B. Huang, C. Zhang, F. Chen, Y. Deng, J. Yin, S. Dustdar, and A. Y. Zomaya, ‘‘Cloud-native computing: A survey from the perspective of services,’’ 6 2023.

[30] ‘‘Security, algorithms and distributed systems,’’ 9 2020.

[31] A. M. Ahmed, M. A. Saeed, A. A. Hamood, A. A. Alazab, and K. A. Ahmed, ‘‘Comparative study of static analysis and machine learning approaches for detecting android banking malware,’’ in *2023 3rd International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pp. 1–8, IEEE, 10 2023.

[32] D. Schneider, M. K. Yurt, F. Simonis, and B. Uekermann, ‘‘Aste: An artificial solver testing environment for partitioned coupling with precice,’’ *Journal of Open Source Software*, vol. 9, pp. 7127–7127, 11 2024.

[33] 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.

[34] F. Kargl, B. Erb, and C. Bösch, *Defining Privacy*, pp. 461–463. Springer International Publishing, 7 2022.

[35] K. Erciyes, *Distributed Graph Algorithms*, pp. 117–136. Springer International Publishing, 4 2018.

[36] G. B. C, A. J. S, S. S, V. M, and R. M, ‘‘Water theft and leakage identification in distributed system,’’ in *2022 Smart Technologies, Communication and Robotics (STCR)*, pp. 1–4, IEEE, 12 2022.

[37] D. F. Blanco and F. L. Mouël, ‘‘Infrastructure de services cloud faas sur noeuds iot,’’ 6 2020.

[38] R. P. M, ‘‘Dynamic load balancing in distributed systems,’’ *International Journal for Research in Applied Science and Engineering Technology*, vol. 13, pp. 694–701, 9 2025.

[39] G. Sun, T. Alpcan, B. I. P. Rubinstein, and S. Camtepe, ‘‘A communication security game on switched systems for autonomous vehicle platoons,’’ in *2021 60th IEEE Conference on Decision and Control (CDC)*, pp. 2690–2695, IEEE, 12 2021.

[40] I. Schagaev, H. Cai, and S. Monkman, *On Performance: From Hardware up to Distributed Systems*, pp. 221–247. Springer International Publishing, 7 2019.

[41] 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.

[42] W. Wu, ‘‘Accelerating distributed systems by in-network computing,’’ in *Proceedings of the ACM CoNEXT Workshop on In-Network Computing and AI for Distributed Systems*, pp. 1–2, ACM, 11 2025.

[43] A. A. Klishin, D. J. Singer, and G. van Anders, ‘‘Avoidance, adjacency, and association in distributed systems design,’’ *Journal of Physics: Complexity*, vol. 2, pp. 025015–, 4 2021.

[44] Y. Xia, D. Du, Z. Hua, B. Zang, H. Chen, and H. Guan, ‘‘Boosting inter-process communication with architectural support,’’ *ACM Transactions on Computer Systems*, vol. 39, pp. 1–35, 11 2021.

[45] D. Sukhoplyuev and A. Nazarov, ‘‘Methods for analyse performance in distributed systems,’’ *E3S Web of Conferences*, vol. 419, pp. 1029–01029, 8 2023.

[46] L. Guegan, B. L. Amersho, A.-C. Orgerie, and M. Quinson, *AINA - A Large-Scale Wired Network Energy Model for Flow-Level Simulations*, vol. 926, pp. 1047–1058. Springer International Publishing, 3 2019.

[47] Z. J. Hamad and S. R. M. Zeebaree, ‘‘Recourses utilization in a distributed system: A review,’’ 1 2021.

[48] C. Dajun, Q. Zhangquan, Z. Qi, and L. Li, ‘‘Spark distributed system and gpu parallel computing,’’ in *International Conference on Electronic Information Engineering and Computer Science (EIECS 2022)*, pp. 64–64, SPIE, 4 2023.

[49] F. Neubauer, B. Uekermann, and J. Pleiss, ‘‘Ai-assisted json schema creation and mapping,’’ in *2025 ACM/IEEE 28th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 79–83, IEEE, 10 2025.

[50] S. Gregersen, ‘‘Aneris: A mechanised logic for modular reasoning about distributed systems,’’ 3 2021.

[51] S. Priyadarshini and S. Rodda, ‘‘Frequent subgraph mining by giraph distributed system,’’ *International Journal of Engineering and Advanced Technology*, vol. 9, pp. 1267–1275, 6 2020.

[52] 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.

[53] N. Parlavantzas, ‘‘Automated application and resource management in the cloud,’’ 6 2020.

[54] 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.

[55] S. Akbari and M. Hauswirth, ‘‘Sampling in cloud benchmarking: A critical review and methodological guidelines,’’ in *2024 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 160–167, IEEE, 12 2024.

[56] 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.

[57] N. Nischal, ‘‘Automated evaluation for distributed system assignments,’’ 1 2023.

[58] I. Murturi, P. K. Donta, and S. Dustdar, ‘‘Community ai: Towards community-based federated learning,’’ in *2023 IEEE 5th International Conference on Cognitive Machine Intelligence (CogMI)*, pp. 1–9, IEEE, 11 2023.

[59] A. Moumen, L. Zahiri, M. Jammoukh, and K. Mansouri, *Numerical Modeling of the Thermomechanical Behavior of Polypropylene Reinforced by Snail Shell Particles as a Sustainable and Ecological Biocomposite*, pp. 359–369. Springer International Publishing, 2 2022.

[60] I. Colonnelli and M. Aldinucci, ‘‘Workflow models for heterogeneous distributed systems,’’ 5 2022.

[61] L. Bradatsch, M. Haeberle, B. Steinert, F. Kargl, and M. Menth, ‘‘Secure service function chaining in the context of zero trust security,’’ in *2022 IEEE 47th Conference on Local Computer Networks (LCN)*, pp. 123–131, IEEE, 9 2022.

[62] M. Maresch and S. Nastic, ‘‘Vate: Edge-cloud system for object detection in real-time video streams,’’ in *2024 IEEE 8th International Conference on Fog and Edge Computing (ICFEC)*, pp. 27–34, IEEE, 5 2024.

[63] Y. Tan, C. Tan, Z. Mi, and H. Chen, ‘‘Pipellm: Fast and confidential large language model services with speculative pipelined encryption,’’ in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 843–857, ACM, 3 2025.

[64] A. Mahida, ‘‘Enhancing observability in distributed systems-a comprehensive review,’’ *Journal of Mathematical & Computer Applications*, vol. 2, pp. 1–4, 9 2023.

[65] P. D. Crutcher, N. K. Singh, and P. Tiegs, *Computer Networks and Distributed Systems*, pp. 133–164. Apress, 6 2021.

[66] S. Soori, B. Can, M. Gurbuzbalaban, and M. M. Dehnavi, ‘‘Async: Asynchronous machine learning on distributed systems.,’’ 7 2019.

[67] J. F. Herculano, L. O. S. de Andrade, W. D. P. Pereira, and A. S. de Sá, ‘‘E-theta: An efficient and adaptive tdma approach for wireless body area networks,’’ in *2024 XIV Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 1–6, IEEE, 11 2024.

[68] P. Chakraborty, *Distributed Systems*, pp. 437–557. Chapman and Hall/CRC, 10 2023.

[69] T. Pusztai, J. Hisberger, C. Marcelino, and S. Nastic, ‘‘Stardust: A scalable and extensible simulator for the 3d continuum,’’ in *2025 IEEE International Conference on Edge Computing and Communications (EDGE)*, pp. 44–53, IEEE, 7 2025.

[70] R. K. Ghosh and H. Ghosh, *Distributed Systems*. Wiley, 2 2023.

[71] 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.

[72] J. Pennekamp, J. Lohmöller, E. Vlad, J. Loos, N. Rodemann, P. Sapel, I. B. Fink, S. Schmitz, C. Hopmann, M. Jarke, G. Schuh, K. Wehrle, and M. Henze, *Designing Secure and Privacy-Preserving Information Systems for Industry Benchmarking*, pp. 489–505. Germany: Springer Nature Switzerland, 6 2023.

[73] N. Benmoussa, A. Elyamami, K. Mansouri, M. Qbadou, and E. Illoussamen, ‘‘A multi-criteria decision making approach for enhancing university accreditation process,’’ *Zenodo (CERN European Organization for Nuclear Research)*, 2 2019.

[74] U. Ogiela, M. Takizawa, and L. Ogiela, *Cybersecurity of Distributed Systems and Dispersed Computing*, pp. 434–438. Springer International Publishing, 3 2023.