Adaptive Replica Placement for Hot Embedding Vectors in Distributed Recommendation Serving Systems for E-Commerce Marketplaces

Andika Saputra[1] **and Fajar Wirawan**[2]

[1]Universitas Muhammadiyah Yogyakarta, Department of Informatics Engineering, Jalan Lingkar Selatan, Tamantirto, Bantul, Yogyakarta, Indonesia
[2]Universitas Mulawarman, Faculty of Computer Science and Information Technology, Jalan Kuaro, Samarinda Ulu, East Kalimantan, Indonesia

ABSTRACT Large-scale e-commerce marketplaces rely on distributed recommendation serving systems that retrieve and aggregate embedding vectors under strict tail-latency and availability targets. In these systems, a small fraction of embedding vectors often accounts for a disproportionate share of lookup traffic, and the identity of this hot set can shift rapidly with promotions, inventory changes, and seasonal demand. Replica placement for embedding vectors is therefore a central operational lever, yet it is constrained by memory budgets, network topology, update propagation costs, and multi-tenant isolation requirements. This paper studies adaptive replica placement for hot embedding vectors in distributed recommendation serving, focusing on the gap between static sharding approaches and the non-stationary, bursty workloads typical of marketplaces. We formulate replica placement as an online optimization problem that jointly minimizes expected tail latency and inter-rack traffic while enforcing per-node memory limits and bounded staleness for mutable embeddings. The proposed approach combines streaming hotness estimation, topology-aware placement, and a stability mechanism that prevents oscillations during demand surges. We develop a mathematical model linking request routing, replica multiplicity, and queueing delay, and we derive an adaptive control policy that updates placements using constrained gradients computed from live telemetry. The design addresses practical concerns including write amplification, cache warmup, partial failures, and fairness across embedding tables. The result is an implementable framework that improves latency robustness under shifting hot sets without relying on static overprovisioning or manual tuning.

1. Introduction

Modern recommendation models for e-commerce marketplaces represent users, items, queries, and contextual signals as dense vectors learned from historical interactions. During online serving, these embedding vectors are retrieved by key and combined by downstream neural components to produce ranking scores. The embedding lookup layer is frequently the highest fan-out stage in the inference pipeline because a single request can touch multiple embedding tables and multiple keys per table. While compute accelerators can reduce the marginal cost of dense layers, embedding retrieval remains sensitive to memory locality, network contention, and load imbalance. The operational reality is that embedding lookups must be fast, predictable, and resilient to spikes, because user-facing ranking is often a synchronous dependency of search, browse, ads, and personalization surfaces. A marketplace adds additional complexity: traffic is highly non-stationary; new items appear continuously; promotions induce abrupt redistributions of attention; and multiple business domains share a serving substrate with distinct service-level objectives.

A key empirical property of embedding access patterns is skew [1]. A small subset of keys in several tables, such as popular items, trending queries, and high-frequency

Article info:
Andika Saputra and Fajar Wirawan. *Adaptive Replica Placement for Hot Embedding Vectors in Distributed Recommendation Serving Systems for E-Commerce Marketplaces.* Center for Theoretical and Applied STEM Research (CTASR) . Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0)

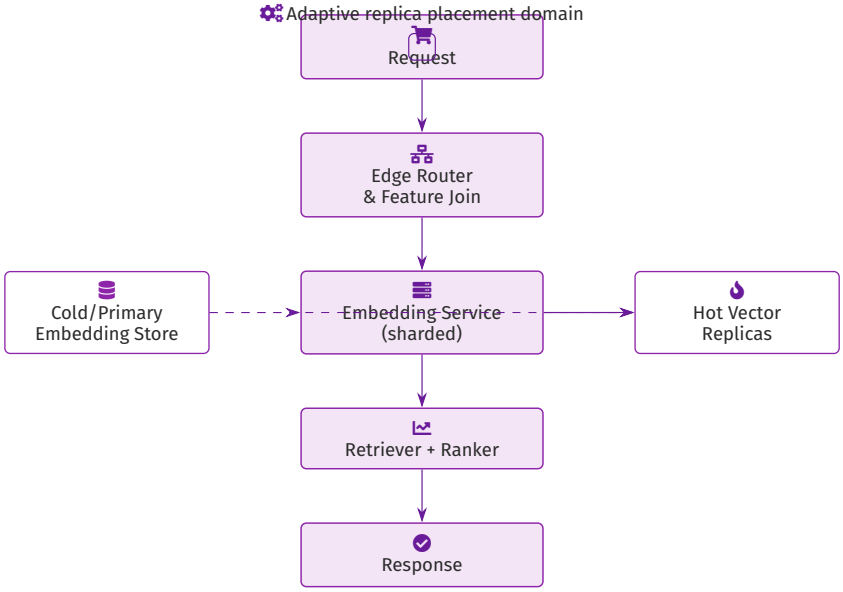


Figure 1 Serving-path architecture for marketplace recommendations. Hot embedding vectors are selectively replicated near embedding shards to reduce tail latency while keeping the primary embedding store as the durable source of truth.

Table 1 Online traffic characteristics across e-commerce verticals

Vertical	Requests/s	Unique IDs (M)	Hot-ID Fraction (%)
Fashion	120k	18	2.3
Electronics	95k	22	1.7
Home & Living	70k	14	2.0
Groceries	140k	10	3.5

sellers, can dominate request volume. This skew is not merely a static Zipf-like distribution; it can evolve on timescales of minutes. A flash sale, influencer-driven demand, or a sudden stockout can change which embeddings are most frequently queried. This dynamism makes static partitioning brittle. With fixed sharding, hot keys concentrate load on a subset of nodes, producing queueing delay, network hotspots, and degraded tail latency. Replication can mitigate imbalance by spreading hot keys across nodes, allowing request routing to select less loaded replicas. However, naive replication introduces its own costs: memory consumption grows; updates must be propagated to multiple locations; consistency and freshness guarantees become harder; and frequent changes can cause placement oscillations that thrash caches and saturate background bandwidth [2].

This paper addresses adaptive replica placement for hot embedding vectors in distributed recommendation serving systems tailored to e-commerce marketplaces. The focus is on embedding vectors as key-value objects that are read frequently and updated at lower but non-negligible rates. Updates may come from nearline training refreshes, incremental feature pipelines, or online learning components. Some embeddings are effectively immutable within a serving horizon, while others must reflect recent signals with bounded staleness. The system also has topology: nodes are arranged in racks and clusters; cross-rack traffic is more expensive and can affect tail behavior. Additionally, multiple embedding tables coexist, with different dimensionalities, access skew, and update rates.

The core problem is to decide, at runtime, which embedding vectors should be replicated, how many replicas each should have, and where those replicas should be placed, while also deciding how requests should be routed among available replicas [3]. The decision must respect memory budgets on each serving node and must avoid destabilizing the system by changing placements too aggressively. The objective is not only to minimize mean latency but also to reduce tail latency and interconnect congestion, because in practice the p99 or higher quantiles drive perceived performance and capacity planning.

We propose a framework that combines streaming hotness estimation with topology-aware replica placement and a control-theoretic stability mechanism. The key

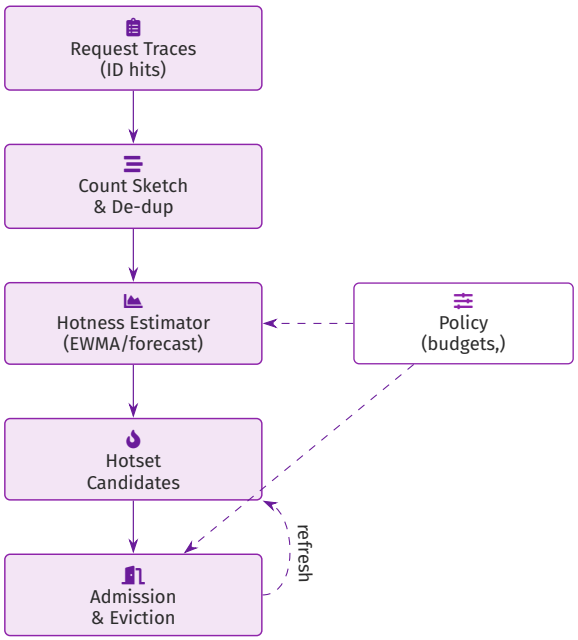


Figure 2 Telemetry-to-hotset pipeline. Streaming hit signals are compacted and forecast to maintain a stable hotset; policy-driven admission/eviction enforces memory budgets and SLA-sensitive priorities.

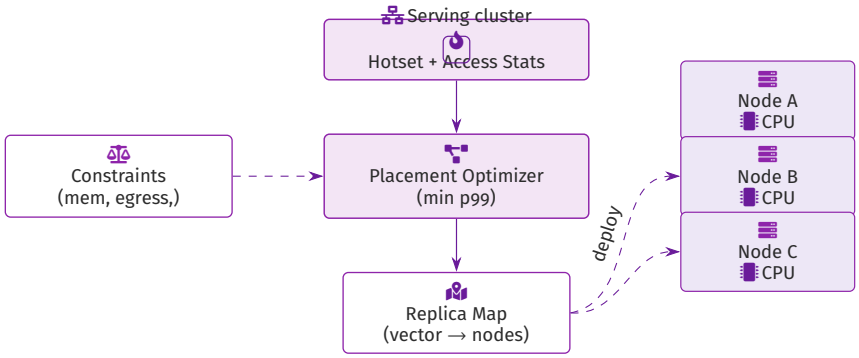


Figure 3 Adaptive replica placement as a constrained optimization step. Given hotness and access locality, the planner selects replica targets that reduce tail latency while respecting memory, network egress, and redundancy constraints across serving nodes.

idea is to treat replica placement as an online resource allocation problem with constraints, where the control variables are per-key replica multiplicities and replica locations. Telemetry from serving nodes provides per-key request rates (possibly approximated), per-node queueing metrics, and network utilization indicators. Using these signals, the system periodically computes placement adjustments that move replicas toward underutilized regions of the topology while increasing replica counts for keys whose marginal contribution to congestion is high. To prevent oscillations, the framework uses hysteresis, bounded step sizes, and placement inertia that accounts for warmup costs [4]. When updates must be propagated, the framework includes an explicit write cost term and a staleness constraint, ensuring that replication decisions remain compatible with freshness requirements.

A distinguishing aspect of marketplace serving is multi-tenancy. Different product surfaces and regions may share embedding infrastructure. Even within a single model, embedding tables may correspond to distinct entity types with very different access patterns. The proposed framework therefore includes a fairness-aware component that prevents one table's hot keys from consuming all memory, which would otherwise degrade other tables and potentially violate their objectives. The approach also explicitly supports partial failures by making placement decisions robust to node loss and by ensuring that every key retains a baseline availability level.

The remainder of this paper develops a formal model and an implementable algorithm [5]. We begin by describing a system architecture and a problem formulation that captures latency, network cost, memory constraints, and update propagation. We then introduce hotness esti-

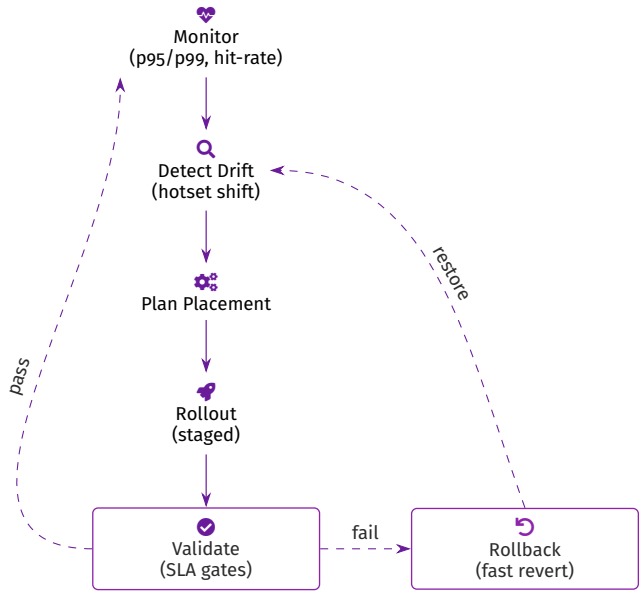


Figure 4 Closed-loop operation for replica placement. The system monitors tail latency and hotset drift, plans changes, rolls out in stages, validates against SLA gates, and quickly rolls back if regressions are detected.

Table 2 Embedding table statistics in the recommendation serving stack

Feature Group	IDs (M)	Dimension	Size (GB)
User	80	64	19.6
Item	160	128	78.1
Query	12	64	2.9
Context	6	32	0.8
Seller	9	64	2.2

mation and discuss the challenges of measuring per-key rates at scale. Next, we present an adaptive replica placement and routing policy, including stability mechanisms and a topology-aware objective. We analyze consistency and freshness under replication and propose a practical propagation design. Finally, we outline an evaluation methodology and discuss implementation considerations relevant to production environments, including failure handling, rollouts, and observability.

2. System Model and Problem Formulation

Consider a serving cluster consisting of N embedding servers connected by a data center network with a rack structure. Let nodes be partitioned into racks indexed by $r \in \{1, \ldots, R\}$, with N_r nodes per rack. Each embedding table $t \in \{1, \ldots, T\}$ stores vectors for keys in a keyspace $\mathcal{K}_t$. A request to the recommendation service triggers a set of embedding lookups across tables [6]. For each lookup, a key $k \in \mathcal{K}_t$ is resolved to one of its available replicas. The embedding servers maintain in-memory stores and optional SSD-backed extensions, but the latency target typically requires that the common hot path be memory-resident. We therefore model memory as the primary constrained resource and treat SSD as a fallback that should be avoided for hot keys.

Let s_t denote the size in bytes of an embedding vector in table t, including overhead. Each node i has a memory budget M_i available for embeddings. A replica placement specifies, for each key k, a set of nodes $\mathcal{P}(k) \subseteq \{1, \ldots, N\}$ that store k. The replica count is $m(k) = |\mathcal{P}(k)|$. The total memory used at node i is $\sum_t \sum_{k \in \mathcal{K}_t : i \in \mathcal{P}(k)} s_t$, which must not exceed M_i after accounting for system overhead and reserved headroom. In practice, embeddings may be compressed and stored in blocks, but the per-vector representation is a sufficient abstraction for placement control.

Requests arrive over time. For each key k, define $\lambda_k(t)$ as its read arrival rate at time t. This rate is non-stationary and may exhibit bursts [7]. We also consider updates. Let $\mu_k(t)$ be the update arrival rate for key k, representing embedding refresh events that must be ap-

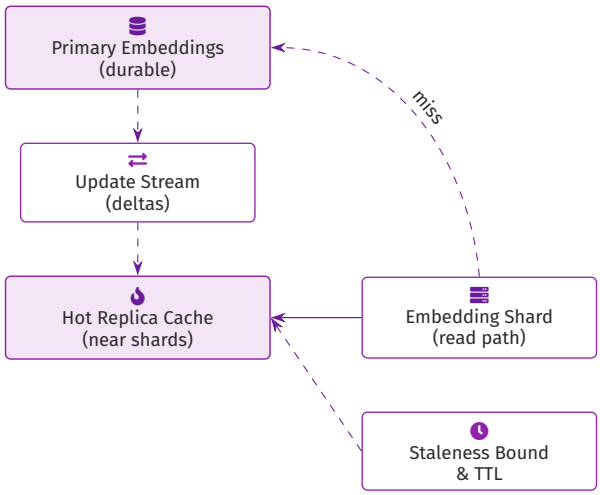


Figure 5 Replica consistency and freshness flow. Reads preferentially hit near-shard hot replicas; asynchronous deltas update replicas from the primary store, while TTL/staleness bounds cap the impact of lag during high-churn marketplace dynamics.

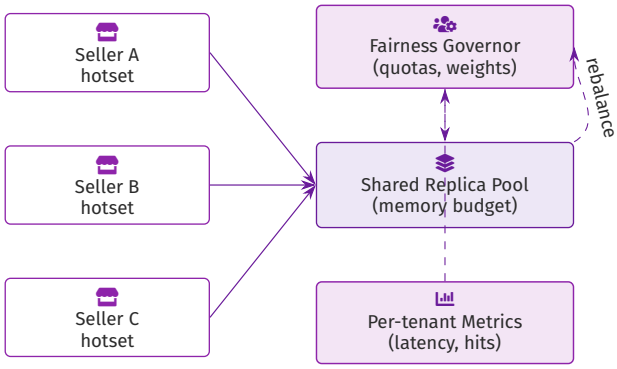


Figure 6 Multi-tenant replica allocation for e-commerce marketplaces. Tenant-specific hotsets compete for a shared replica pool; a fairness governor modulates quotas and weights using per-tenant performance signals to avoid starvation and stabilize tail latency.

plied to all replicas. Updates may be batched, but we model their effect through a cost term and staleness constraints.

Request routing is modeled as a mapping that selects a replica for each lookup. Let $p_{k,i}(t)$ be the fraction of reads for key k routed to node i at time t, with $p_{k,i}(t) = 0$ if $i \notin \mathcal{P}(k)$ and $\sum_{i \in \mathcal{P}(k)} p_{k,i}(t) = 1$. The effective arrival rate at node i is then $\Lambda_i(t) = \sum_t \sum_{k \in \mathcal{K}_t} \lambda_k(t) p_{k,i}(t)$. Each node has a service capacity that depends on memory bandwidth, CPU overhead, serialization, and network stack behavior. Let C_i denote the effective service rate in lookups per second for node i on the mix of tables. We model queueing delay using an $M/G/1$ approximation with utilization $\rho_i(t) = \Lambda_i(t)/C_i$ and assume that for stable operation $\rho_i(t) < 1$ with headroom [8].

Tail latency is influenced by queueing and by network. A lookup can be local to the rack or cross-rack depending on the requester's location and replica placement. Suppose the inference service frontends are co-located with embedding servers in the same cluster, potentially in each rack. Let $\alpha_{i,r}$ denote the fraction of requests originated from rack r that are routed to node i. Then network cost depends on whether i is in the same rack as the origin. We capture network contribution to latency as an additive term whose expected value depends on cross-rack probability and congestion. For tractability, define a topology cost $d(r, i)$ representing the base latency or cost of sending a lookup from rack r to node i, with d larger for cross-rack traffic. Congestion effects can be represented by a nonlinear penalty on aggregate cross-rack traffic, but in an online control loop it is useful to include a marginal cost term derived from measured network utilization [9].

The objective is to select $\mathcal{P}(k)$ and routing probabilities $p_{k,i}(t)$ to minimize an aggregate cost that captures tail latency and network usage, while satisfying memory constraints and freshness requirements. Because tail latency is not linear in utilization, we use a convex surrogate that increases sharply as ρ_i approaches 1. A common form is $\phi(\rho_i) = \rho_i/(1-\rho_i)$ or $\phi(\rho_i) = -\log(1-\rho_i)$, both of which penalize high utilization. Additionally, we include a network term that penalizes cross-rack routing.

Table 3 Hotness distribution and cache hit rates for embedding lookups

Popularity Percentile	Request Share (%)	Baseline Hit (%)	Adaptive Hit (%)
Top 1%	46.2	92.1	99.3
1–5%	24.8	75.4	93.7
5–20%	18.9	51.6	81.2
20–100%	10.1	12.5	29.4

Table 4 End-to-end latency breakdown (P99) before and after adaptive placement

Component	Baseline P99 (ms)	Adaptive P99 (ms)
Network RPC	18.4	13.2
Embedding Lookup	27.9	11.5
Ranking & Scoring	14.7	13.9
Feature Preprocessing	6.1	6.0
End-to-End	67.2	45.8

Let $o(k)$ denote the origin distribution of requests for key k across racks, which may be approximated by the distribution of the higher-level request fan-out. The expected network cost for key k is $\sum_r o_r(k) \sum_{i \in \mathcal{P}(k)} p_{k,i} d(r,i)$. Finally, replication induces update propagation cost. If updates for key k must be applied to all replicas, then the background write load is proportional to $\mu_k(t) m(k)$, with an additional factor for batching and protocol overhead. This cost can saturate bandwidth and interfere with reads if not controlled [10].

A simplified time-sliced optimization at decision epoch τ can be written using estimated rates $\hat{\lambda}_k$ and $\hat{\mu}_k$ and measured node capacities. Let $x_{k,i} \in \{0,1\}$ indicate whether key k is stored at node i, and let $p_{k,i} \geq 0$ be routing fractions. The problem can be expressed as

$$\min_{\{x_{k,i}, p_{k,i}\}} \quad \sum_{i=1}^{N} w_i \, \phi\left(\frac{1}{C_i} \sum_k \hat{\lambda}_k p_{k,i} \right) + \beta \sum_k \sum_r o_r(k) \sum_i p_{k,i} d(r,i) + \gamma \sum_k \hat{\mu}_k \sum_i x_{k,i} \tag{1}$$

$$\text{s.t.} \quad \sum_i p_{k,i} = 1 \quad \forall k \qquad \text{and} \qquad 0 \leq p_{k,i} \leq x_{k,i} \quad \forall k, i \tag{2}$$

$$\sum_k s(k) x_{k,i} \leq M_i \quad \forall i \tag{3}$$

$$\mathbb{P}(\text{staleness}(k) \leq \Delta_k) \geq 1 - \epsilon_k \quad \forall k \tag{4}$$

where $s(k) = s_t$ for $k \in \mathcal{K}_t$, w_i weights nodes (possibly uniform), β trades off network cost, and γ trades off update overhead. Constraint (4) captures bounded staleness with threshold Δ_k and violation probability ϵ_k. The exact staleness distribution depends on propagation design and is addressed later, but it is included here to emphasize that replica count is not free when updates exist.

Problem (1)–(4) is a large-scale mixed-integer program. It is not directly solvable online for production-scale keyspaces, which can reach billions. The central challenge is therefore to develop a scalable approximation that focuses on the subset of keys where replication decisions matter most, while keeping placement stable and respecting constraints. We adopt a structure commonly observed in serving: most keys are cold and can be stored with a baseline replication and a fixed placement derived from sharding; a small hot set requires adaptive replication. This motivates a two-tier model: a static base layer and a dynamic hot layer.

Let $\mathcal{H}$ be the set of hot keys considered for adaptive replication at epoch τ. The rest of the keys $\mathcal{K} \setminus \mathcal{H}$ are placed by a base sharding function with a fixed replication factor m_0 for availability. The adaptive controller can increase replicas for $k \in \mathcal{H}$ beyond m_0 and can choose their locations within a candidate pool. The optimization is then restricted to $\mathcal{H}$, and the memory constraints incorporate the baseline footprint. Even with this restriction, the controller must operate under partial observability: per-key request rates are estimated; routing fractions may be influenced by client-side behavior; and node capacities vary with background load [11]. We therefore adopt an online control formulation in which decisions are adjusted incrementally based on measured gradients rather than solved from scratch.

To connect replica placement to tail latency, we use

Table 5 Throughput under a 50 ms P99 SLO for different serving strategies

System	Max Throughput (kQPS)	CPU Util. (%)	GPU Util. (%)
Static Partitioning	210	78	41
LRU Cache	245	81	44
Adaptive Placement	310	84	47
Oracle Upper Bound	335	86	49

Table 6 Replica usage and control-plane overhead

Metric	Static Scheme	Adaptive Placement
Avg. Replication Factor	2.0	1.6
Memory Overhead (%)	40	27
Control Traffic (MB/s)	5.2	7.9
Metadata Footprint (MB)	82	95

a queueing-based surrogate. For node i, with service time random variable S_i having mean $\mathbb{E}[S_i]$ and second moment $\mathbb{E}[S_i^2]$, the Pollaczek–Khinchine formula for an $M/G/1$ queue gives mean waiting time W_i as $\Lambda_i \mathbb{E}[S_i^2]/(2(1-\rho_i))$. Tail quantiles require more elaborate analysis, but in many operational settings reducing high utilization is strongly correlated with reducing high quantiles, because the waiting-time distribution becomes heavy as $\rho \to 1$. Therefore $\phi(\rho) = \rho/(1-\rho)$ serves as a reasonable control signal. We also incorporate a penalty for rapid changes in placement to account for warmup costs and to avoid oscillations. Let $x_{k,i}^{\text{prev}}$ be the previous placement. A stability term can be added as $\eta \sum_{k,i} |x_{k,i} - x_{k,i}^{\text{prev}}|$, approximated by a convex relaxation when needed.

The control loop operates at discrete epochs, for example every few minutes. At each epoch it ingests telemetry, updates hotness estimates, computes recommended changes, and executes them via a background replication mechanism [12]. During execution, routing is adjusted to gradually shift reads to new replicas once they are warmed. The design must be robust to the fact that telemetry itself can be delayed and noisy, and that workload can change within an epoch. These considerations shape the algorithmic choices described in later sections.

3. Workload Characterization and Hotness Estimation

Adaptive replica placement depends on accurate identification of hot embedding vectors and on forecasts of near-future demand. The fundamental constraint is that the keyspace is massive, making exact per-key counters infeasible. In addition, marketplace traffic is non-stationary and multi-modal, with daily cycles, weekly patterns, and event-driven spikes. A practical design must therefore combine approximate counting for long tails with precise tracking for the current hot set, and it must distinguish short transient bursts from sustained popularity to avoid unnecessary replica churn [13].

Let $\lambda_k(t)$ be the instantaneous request rate for key k. For adaptive decisions at epoch τ, the controller needs an estimate of the average rate over a recent window, such as $\bar{\lambda}_k(\tau) = \frac{1}{W}\int_{\tau-W}^{\tau} \lambda_k(u)\,du$, and potentially a short-horizon forecast $\tilde{\lambda}_k(\tau+\delta)$. A naive moving average is slow to react or too noisy depending on W. Exponential smoothing is often used, with $\hat{\lambda}_k(\tau) = (1-\alpha)\hat{\lambda}_k(\tau-\Delta) + \alpha\lambda_k(\tau)$ where Δ is the sampling period. However, exact $\lambda_k(\tau)$ is not observable for all keys. The system instead observes sampled requests and aggregated metrics.

A common approach uses a hierarchical measurement scheme. At the lowest level, each embedding server maintains approximate heavy-hitter structures for the keys it serves. Because base sharding maps each key to a primary node, the primary naturally observes the majority of reads in a non-replicated system [14]. With replication, reads are split across replicas, so relying solely on one node can undercount. To address this, measurement can be anchored on the frontend: the inference service that initiates lookups can sample key IDs and maintain approximate global counts. Alternatively, a distributed sketch can be maintained across embedding nodes and merged periodically. In a marketplace, where many services may share embeddings, measurement at the routing layer is often the most accurate representation of demand, because it captures the keys requested

Table 7 Ablation study on components of the replica placement policy

Variant	P99 Latency (ms)	Tail Drop Rate (%)
Full Policy	45.8	0.21
w/o Hotness Decay	49.3	0.34
w/o Load-Aware Balancing	52.6	0.51
w/o Cross-Rack Placement Awareness	48.7	0.29
Static Replication	67.2	1.08

Table 8 Reaction to workload shifts with adaptive replica reconfiguration

Scenario	Detection Time (s)	Convergence (s)	Max P99 Increase (%)
Flash Sale	6.3	42	18.4
Evening Traffic Spike	11.2	65	12.7
Cold-Start Item	8.7	37	9.3
Regional Outage	4.9	54	21.5

regardless of where they are served.

We model measurement as follows. Each lookup event for key k is sampled independently with probability q, producing a sampled stream. The controller maintains a sketch that estimates $\bar{\lambda}_k$ for keys above a threshold. For the heaviest keys, the controller can maintain exact counters after they enter the candidate set [15]. The sampling probability q can be adapted based on traffic volume to keep overhead bounded. The estimation variance scales inversely with q, so a moderate q can provide adequate precision for the hot set while remaining lightweight.

Hotness is not only about absolute request rate. A key's impact on system load depends on where it is served and what it costs to serve. Embedding vectors may have different sizes across tables; some tables may have higher per-lookup CPU overhead due to decompression or feature transformations; and some keys may be requested by more latency-sensitive surfaces. Therefore the controller uses a weighted hotness score. For key k in table t, define a per-lookup cost coefficient c_k that captures the expected service time contribution and a per-lookup network coefficient n_k capturing expected cross-rack cost if served from a non-local rack [16]. These coefficients can be learned from telemetry by regressing observed CPU time and network bytes on lookup counts, or can be approximated using table-level constants.

A useful score is the estimated marginal pressure a key exerts on a bottleneck. Let $\mathcal{B}$ denote the set of nodes currently operating at high utilization. For each key k, compute the fraction of its traffic served by nodes in $\mathcal{B}$, denoted $b_k = \sum_{i \in \mathcal{B}} p_{k,i}$. Then a pressure score is $h_k = \hat{\lambda}_k c_k b_k$. Intuitively, a key is more replication-worthy if it is both frequently accessed and concentrated on congested nodes. This differs from pure frequency ranking and can significantly reduce unnecessary replication of keys already well-distributed.

Non-stationarity requires forecasting or at least robust smoothing. Marketplace events can produce sudden spikes where the hot set changes in minutes. Replication, however, has a nonzero actuation delay, because copying a vector to a new node requires allocation, transfer, insertion into memory structures, and cache warmup [17]. Therefore the controller benefits from predicting near-future demand. In practice, short-horizon prediction can be built from leading indicators. For example, if the higher-level request is for an item page, the item ID appears in logs before the embedding lookup layer sees repeated accesses. Similarly, search query trends can be tracked at the query service. While an embedding system may not have direct access to all upstream signals, it can incorporate a simple autoregressive forecast based on its own observed counts. A robust option is to estimate a trend component using a second-order exponential smoothing, where both level and slope are tracked, allowing faster reaction to sustained changes.

To decide whether a key is hot enough to replicate, the system applies an admission policy [18]. Admission must balance sensitivity with stability. If admission is too sensitive, keys will frequently enter and leave the hot set, causing thrash. If it is too conservative, the system will not react to spikes. A common technique is hysteresis: use a higher threshold to admit keys than to evict them. Let H_{in} be the admission threshold and H_{out} the eviction threshold with $H_{\text{out}} < H_{\text{in}}$. A key enters $\mathcal{H}$ when h_k ex-

Table 9 Serving cost comparison at steady state

System	Nodes Used	Cost / 1M Req ($)	Relative Cost
Static Partitioning	220	15.8	1.00
LRU Cache	205	14.9	0.94
LFU Cache	198	14.6	0.92
Adaptive Placement	180	13.1	0.83

ceeds H_{in} for a persistence duration, and it exits when h_k stays below H_{out} for a longer duration. Persistence can be implemented by requiring the threshold to be met for L consecutive sampling intervals or by using an exponentially decayed occupancy score. This introduces controlled inertia and reduces replica churn.

The size of $\mathcal{H}$ is limited by memory overhead and control complexity. A typical design sets a cap $|\mathcal{H}| \le H_{\max}$ and selects the top keys by pressure score subject to table-level quotas. Table-level quotas are necessary because a single highly skewed table, such as item embeddings, might otherwise dominate the hot set, leaving insufficient adaptation capacity for user embeddings or context embeddings that may also become hot under certain conditions. Quotas can be dynamic, based on each table's contribution to total latency or on contractual service-level objectives [19].

Measurement noise and routing feedback can create estimation bias. When a key is replicated and routed across more nodes, any per-node estimator that only sees local traffic will observe reduced counts, potentially triggering eviction even if global demand remains high. This is a classic measurement-control coupling. The solution is to estimate demand at a point in the pipeline that is upstream of routing, or to correct local observations by scaling with routing fractions. If i observes $\lambda_{k,i}$ and the controller knows $p_{k,i}$, then an unbiased estimate of λ_k is $\lambda_{k,i}/p_{k,i}$, but this becomes unstable when $p_{k,i}$ is small. In practice, combining multiple replica observations and using a minimum routing fraction floor reduces variance. The framework described here assumes that the controller has access to upstream sampled counts that are not distorted by current routing, which is a common architectural choice.

Update rates also matter [20]. Some embedding vectors are refreshed frequently, especially those representing users with recent interactions or items with fast-changing attributes. Replicating such keys can increase write amplification and can threaten freshness guarantees if propagation becomes backlogged. Therefore the admission score can incorporate update cost by subtracting a term proportional to $\hat{\mu}_k$. One can define an adjusted score $h'_k = \hat{\lambda}_k c_k b_k - \kappa \hat{\mu}_k$, where κ reflects the cost ratio between write propagation and read savings. This adjustment prevents the controller from over-replicating keys where updates are expensive relative to the read benefit.

Finally, the controller must handle bursts that exceed the system's replication bandwidth. If many keys become hot simultaneously, copying them all is impossible. In such situations, the controller prioritizes keys with the highest marginal benefit and may temporarily rely on routing-based load balancing among existing replicas. For example, if a key has only baseline replication, routing can still distribute across those replicas, but the degree of freedom is limited [21]. Therefore the hotness estimator can also estimate the marginal benefit of adding one replica, which depends on current utilization and routing. This concept is used directly in the placement algorithm.

4. Adaptive Replica Placement and Routing Algorithm

The adaptive control problem consists of two coupled decisions: replica placement, which determines where keys reside, and routing, which determines how read traffic is split among replicas. Routing can be adjusted quickly, while placement changes are slower. A practical architecture treats routing as the fast inner loop and placement as the slower outer loop. The inner loop reacts to transient load fluctuations within the current placement, and the outer loop adjusts placement to reshape the feasible routing region over longer timescales.

We first describe a model for routing given a fixed placement [22]. Suppose key k has replicas on nodes in $\mathcal{P}(k)$. The controller seeks routing probabilities $p_{k,i}$ that reduce congestion and network cost. A simple congestion-aware routing policy is to allocate traffic inversely proportional to a node's estimated marginal delay. Let ψ_i be a congestion price for node i, increasing with utilization, such as $\psi_i = \phi'(\rho_i)/C_i$ where ϕ is the congestion penalty. The routing decision for key k can then be expressed as a softmin over nodes, balancing congestion and distance:

$$p_{k,i} \propto \exp\left(-\theta\left(\psi_i + \beta \sum_r o_r(k) d(r,i)\right)\right) \quad \text{for} \quad i \in \mathcal{P}(k) \tag{5}$$

with temperature parameter θ controlling how aggressively traffic shifts. This policy avoids hard switching, which can create oscillations and thundering herds, while still moving traffic away from congested nodes. It also incorporates topology preferences by penalizing distant replicas. In systems that require deterministic routing for cache locality, a consistent-hashing-based scheme can be modified by adding multiple choices and selecting among them based on load, preserving stickiness while adding flexibility [23].

Given routing, we consider the marginal benefit of adding or removing a replica for a hot key. The main benefit arises from reducing load on congested nodes by increasing the set of feasible destinations and allowing the routing policy to spread traffic. The marginal cost is memory and update propagation. We approximate the benefit of adding a replica of key k to node j by the decrease in objective (1) induced by allowing $p_{k,j} > 0$. For small changes, we can use a gradient approximation. Let $\mathcal{L}$ be the Lagrangian of (1) with multipliers v_i for memory constraints. Ignoring the discrete nature of $x_{k,i}$, the marginal value of storing key k at node j is approximately

$$\Delta_{k\to j} \approx -\hat{\lambda}_k \left(\psi_{\text{eff}}(k) - \psi_j\right) \quad - \quad \beta\hat{\lambda}_k \left(D_{\text{eff}}(k) - D_{k,j}\right) \quad - \quad \gamma\hat{\mu}_k \quad - \quad v_j s(k) \tag{6}$$

where $\psi_{\text{eff}}(k)$ is the current effective congestion price experienced by key k under its routing distribution, $D_{\text{eff}}(k)$ is the current expected distance cost, and $D_{k,j} = \sum_r o_r(k) d(r,j)$. The term $v_j s(k)$ reflects the opportunity cost of consuming memory at node j under tight constraints. If $\Delta_{k\to j}$ is positive, adding a replica at j is estimated to reduce total cost. This expression has an intuitive interpretation: it is beneficial to add a replica on a node that is less congested than the current serving set and that is topologically close to the request origins, but the benefit must outweigh update and memory costs.

Because the placement decision is discrete and constrained by per-node memory, the controller solves an allocation problem over candidate replicas [24]. A direct approach is to treat each potential key-node replica as an item with value $\Delta_{k\to j}$ and weight $s(k)$, then perform a constrained selection to maximize total value subject to memory budgets. However, the number of candidates can still be large. The framework reduces complexity by restricting candidates. For each hot key k, define a candidate set of nodes $C(k)$ consisting of nodes in racks with high origin probability $o_r(k)$ and nodes that have spare memory and low congestion price. The size of $C(k)$ can be bounded to a few dozen using heuristics such as selecting the best nodes per rack by ψ_j and selecting a small set of nearby racks.

The controller then performs a greedy or approximate knapsack allocation. Greedy selection by value density $\Delta_{k\to j}/s(k)$ is effective when item sizes are similar within a table, which is often the case. When sizes differ substantially across tables, the controller can allocate memory per table first and then do greedy selection within each table, which also aligns with fairness constraints.

Replica removal is similarly governed by marginal loss. A replica of key k at node j can be removed if its contribution is low, which may occur when demand drops or when node j becomes congested and is better used for other keys. The marginal loss of removing a replica is approximated by the increase in objective, which can be computed by considering how routing would redistribute [25]. Removal decisions must incorporate hysteresis to avoid immediate re-addition. Therefore replicas are assigned a minimum residency time unless emergency eviction is needed for memory safety.

A central production concern is stability. If the controller reacts too aggressively to short-term fluctuations, it can cause oscillatory placements where replicas migrate frequently. This thrashes caches and increases background bandwidth. Stability is enforced through multiple mechanisms. One is step-size limiting: at each epoch, only a bounded number of replica changes are allowed globally and per node [26]. Another is a switching cost term, effectively penalizing changes in $x_{k,i}$. The controller can implement this by subtracting a fixed penalty η from $\Delta_{k\to j}$ for adding a new replica and subtracting an analogous penalty for removals, making small marginal gains insufficient to trigger changes. A third mechanism is to use smoothed congestion prices ψ_i rather than instantaneous ones, reducing sensitivity to transient spikes.

The outer loop algorithm can be summarized conceptually as a constrained primal-dual method. Each node has a memory multiplier v_i that rises when memory is near capacity and falls when memory is plentiful. At each epoch, for each candidate replica $k \to j$, compute the adjusted gain in (6). Select additions with positive gain until memory budgets and change-rate limits are reached. Then update multipliers based on observed memory usage:

$$v_i(\tau + \Delta) = \left[v_i(\tau) + \xi\left(U_i(\tau) - M_i\right)\right]_+ \tag{7}$$

where U_i is used memory, ξ is a small learning rate, and $[\cdot]_+$ is projection to nonnegative values. This dual adjustment naturally discourages placing replicas on nodes that are memory-tight, without hardcoding per-node thresholds [27]. In practice, memory constraints must include safety buffers, so M_i in (7) is set to a conservative limit below physical capacity.

The inner loop routing continuously computes $p_{k,i}$ based on (5) using updated congestion prices. Congestion prices can be derived from queue length, service time, and CPU usage. A robust choice is to use an exponentially smoothed utilization estimate $\hat{\rho}_i$ and set $\psi_i = 1/(1 - \hat{\rho}_i)$ scaled by service rate. This is monotone and captures the rapid increase near saturation. To avoid routing instability, the routing updates are also rate-limited and can incorporate per-key stickiness so

that repeated lookups for the same request are consistent.

An important coupling arises because adding replicas changes routing, which changes observed congestion, which then affects the next placement decision. If the controller does not account for this, it can overcompensate. A useful technique is to evaluate marginal gains using a counterfactual congestion price that includes a model of how traffic would redistribute. One can approximate this by assuming that adding a replica for key k spreads its traffic evenly among $m(k)+1$ replicas, or more generally according to softmin weights [28]. The predicted change in node i's arrival rate is then $\Delta\Lambda_i = -\hat{\lambda}_k p_{k,i} + \hat{\lambda}_k p'_{k,i}$, where p' is the predicted routing after the addition. Using this, the controller can approximate the change in objective by a first-order expansion:

$$\Delta\mathcal{J} \approx \sum_i w_i \phi'(\rho_i)\frac{\Delta\Lambda_i}{C_i} \quad + \quad \beta\hat{\lambda}_k\left(D'_{\text{eff}}(k) - D_{\text{eff}}(k)\right) \quad + \quad \gamma\hat{\mu}_k \tag{8}$$

where $D'_{\text{eff}}(k)$ is the predicted expected distance after the change. This counterfactual evaluation reduces myopic choices that might shift congestion from one node to another without net benefit.

Marketplace serving often involves multiple clusters by region or availability zone. Replica placement within a cluster is the primary focus, but cross-cluster replication may exist for disaster recovery or geo-latency. The same framework can be extended by treating clusters as higher-level nodes with larger distance costs and stricter bandwidth constraints. However, the actuation delay is larger, so the controller should operate at slower timescales for cross-cluster decisions. Within a single data center cluster, the described approach is sufficiently responsive for minute-level dynamics [29].

We also address fairness across tables. Let B_t be a memory budget allocated to table t for adaptive replicas. These budgets can be proportional to each table's baseline footprint and its latency sensitivity. The allocation problem then includes constraints $\sum_{k\in\mathcal{H}\cap\mathcal{K}_t}\sum_i s_t x^{\text{extra}}_{k,i} \le B_t$, where x^{extra} indicates replicas beyond baseline. Dual variables can be used per table as well, creating a hierarchical primal-dual controller. This structure prevents one table's hot keys from consuming all adaptive memory and ensures that improvements are distributed in a way consistent with service priorities.

Finally, the controller must consider availability and failure domains. If replicas are placed too densely in a single rack, a rack failure can remove all replicas of a key [30]. Therefore placement includes diversity constraints, such as requiring that replicas for a key span at least two racks beyond baseline. This can be expressed as a constraint on rack-level placement indicators, but in practice it is enforced by candidate selection: when adding replicas, the controller prefers racks not already hosting the key's replicas, unless a latency-critical policy explicitly allows consolidation.

5. Consistency, Freshness, and Update Propagation Under Replication

Embedding replication is beneficial for reads, but it complicates updates. In marketplace systems, embeddings can be updated by periodic retraining pushes, incremental nearline updates, and occasionally by online learning. Some tables are effectively append-only or update infrequently, such as static item attributes encoded into embeddings; other tables, such as user embeddings, may update more frequently. The replica placement controller must be aware of these differences because update propagation can consume bandwidth, interfere with read latency, and violate freshness requirements if lag grows.

We model an update stream for key k with rate $\mu_k(t)$ [31]. Each update carries a new vector value and a version number. Replicas must converge to the newest version, but strict linearizability is rarely required for recommendation embeddings because the model is robust to small inconsistencies, and the downstream ranking typically tolerates minor staleness. Nevertheless, there are operational bounds: a user embedding that reflects recent clicks should not lag by hours, and a price-sensitive item embedding should not be too stale relative to inventory changes. Therefore serving systems typically implement eventual consistency with bounded staleness. The bounded staleness constraint in (4) can be interpreted as requiring the update propagation delay distribution to have most mass below Δ_k.

Update propagation can be designed in several ways. A common design uses a primary-replica scheme: each key has a designated primary node that receives updates and then forwards them to secondaries. Another design uses a publish-subscribe mechanism where updates are written to a log and replicas subscribe to updates for keys they store [32]. The latter decouples writers from the replica set and can simplify replica changes, because when a replica is added it can replay the log from a checkpoint. The tradeoff is that log-based propagation requires durable infrastructure and careful handling of backpressure.

We consider a log-based approach because it aligns well with adaptive placement. Updates for each table are appended to a partitioned log keyed by k. Each embedding server maintains subscriptions for the partitions corresponding to keys it stores. When a replica set changes, a node begins consuming updates for newly assigned keys. To avoid replaying an unbounded history, the system stores periodic snapshots or checkpoints of embeddings, or it uses a compacted log where only the latest value per key is retained beyond a horizon [33]. Because embeddings are large but manageable, compaction is often feasible. The main concern becomes propagation lag: if a node is overwhelmed, it may fall behind and

serve stale values.

Propagation lag can be modeled as a queue. Let node i receive update events for keys it stores at total rate $_i = \sum_{k:i\in\mathcal{P}(k)} \mu_k$. Let its update processing capacity be U_i updates per second, capturing network ingress, deserialization, and in-memory writes. The utilization for updates is $\rho_i^{(u)} =_i /U_i$. If reads and updates share resources, the effective capacity for each depends on scheduling. Many systems prioritize reads, meaning updates are processed opportunistically, which can increase lag under heavy read load. This introduces a coupling: replicating a frequently updated key increases $_i$ at multiple nodes and can increase lag if update capacity is limited [34].

To incorporate this in placement control, the update cost term $\gamma \sum_k \hat{\mu}_k m(k)$ is a coarse proxy. A more refined model accounts for per-node update utilization and its effect on read latency via interference. Suppose node i has total CPU budget normalized to 1, with read processing consuming fraction proportional to $\rho_i^{(r)}$ and update processing consuming fraction proportional to $\rho_i^{(u)}$. If the scheduler prioritizes reads, updates may be throttled when reads are high. A simple interference model sets effective read service rate as $C_i^{\text{eff}} = C_i(1-\delta\rho_i^{(u)})$ for some $\delta \in [0,1]$ capturing contention, and effective update service rate as $U_i^{\text{eff}} = U_i(1-\delta'\rho_i^{(r)})$. While approximate, this captures the qualitative behavior that too many replicas of frequently updated keys can degrade both freshness and latency.

Freshness can be expressed through the distribution of propagation delay. For log-based propagation, the delay is the time from log append to application at a replica. If updates are processed in order per partition, the lag can be measured as the log offset difference. Let $L_{i,p}$ be the lag in seconds for node i on partition p. For key k in partition $p(k)$, the staleness is approximately $L_{i,p(k)}$ at node i. The bounded staleness requirement can then be enforced by ensuring $L_{i,p}$ stays below thresholds. The controller can incorporate this by penalizing placements that would assign high-update-rate keys to nodes with high current lag or limited update capacity [35].

Replica addition creates a cold-start issue for freshness. When a new replica is created, it must obtain the latest value. If the system copies the current embedding value from an existing replica, it can then subscribe to incremental updates. If it relies purely on the log, it may need to catch up from a checkpoint. The time to warm a replica is therefore a function of transfer bandwidth and current lag. During warmup, routing should not send reads to the new replica, otherwise it would serve missing or stale values. The framework therefore separates replica installation from activation [36]. Installation transfers the current value and begins subscription, and activation occurs when the replica is caught up and has passed a validation check. Activation can be conditional on the replica's lag being below a threshold.

Consistency among replicas can be handled with version numbers. Each embedding value carries a monotonically increasing version, such as a timestamp or training step. Reads can accept any version within a staleness bound. For certain tables, it may be important to avoid serving values older than a global minimum version, for example when a model rollout expects embeddings to be compatible with a new schema. In that case, the system enforces a floor version and blocks or falls back when a replica lags. This interacts with placement because adding replicas increases the number of consumers that must keep up with the floor [37]. Therefore placement decisions during rollouts may be temporarily constrained, or update capacity may be provisioned.

A practical propagation design also needs to handle write amplification. If a key has $m(k)$ replicas, each update is applied $m(k)$ times. In a log-based system, the update is still published once, but it is consumed by $m(k)$ nodes. This increases network and CPU usage. The controller's update cost term can be made key-specific by incorporating the embedding size and update payload size. For table t, let update payload size be approximately s_t [38]. Then the total update bandwidth is roughly $\sum_k \mu_k s(k) m(k)$. This motivates a cost term $\gamma \sum_k \hat{\mu}_k s(k) m(k)$, which penalizes large frequently updated embeddings more strongly.

We now connect update propagation to placement stability. If placements change frequently, replicas are added and removed, which can cause spikes in update subscription changes and in state transfer. Even if updates are infrequent, the transfer of the full embedding value during installation can be expensive. Therefore the stability penalty in placement should account for state transfer size. A switching cost for adding a replica of key k can be set proportional to $s(k)$, reflecting bytes transferred, and also to an estimated warmup time. The controller can then avoid moving large embeddings unless the expected read benefit is significant [39].

The design must also address correctness under partial failures. If a node fails and loses in-memory state, replicas must be reconstructed. For cold keys, reconstruction can be lazy from persistent storage. For hot keys, reconstruction must be rapid to avoid overload on remaining replicas. Therefore the system maintains baseline replication $m_0 \geq 2$ and ensures that adaptive replicas are not all placed in a single failure domain. Additionally, the controller should treat failure events as triggers for temporary adaptation, increasing replicas for keys whose remaining replicas are under pressure. This requires that the placement controller integrate with cluster membership and failure detection [40].

Finally, marketplace systems often require multi-tenant isolation. Different business units may have separate update pipelines with different priorities. A global log infrastructure must enforce quotas so that one tenant's updates do not starve others. Similarly, adaptive replication must not amplify a noisy tenant's update traffic beyond budget. This can be enforced by per-tenant

update cost multipliers and by table-level budgets. Because these constraints are organization-specific, the framework emphasizes modular cost terms that can be tuned without changing the algorithmic core.

6. Analytical Modeling of Latency, Capacity, and Replica Multiplicity

To justify and tune adaptive replica placement, it is useful to develop a quantitative model connecting replica multiplicity to tail latency and to resource consumption [41]. While production systems require empirical validation, an analytical model provides guidance for controller design and helps avoid pathological regimes.

We consider a simplified setting with a single embedding table and homogeneous nodes, then extend to heterogeneous topology. Let N nodes each provide service rate C lookups per second. Requests for key k arrive as a Poisson process with rate λ_k, and each lookup has service time distribution with mean $1/C$ after normalization. Suppose key k is replicated on $m(k)$ nodes, and its traffic is split evenly among replicas, so each replica receives rate $\lambda_k/m(k)$. For a given node, the total arrival rate is the sum over keys assigned to it. In a baseline sharding scheme without adaptive replication, each key has a primary assignment, so node loads are determined by the sum of rates of keys mapped to the node [42]. With skew, a node that hosts a few hot keys can have much higher load than average.

Replica multiplicity affects the load variance across nodes. One way to model this is to treat key-to-node assignment as random and compute the expected maximum load. However, in practice assignments are deterministic and may be correlated with key popularity due to hashing, but randomness is still a useful approximation for average behavior. Suppose the base assignment is uniform random. The expected load at a node is $\bar{\Lambda} = \frac{1}{N}\sum_k \lambda_k$, but the variance depends on the second moment of λ_k. Replication effectively reduces the contribution of hot keys to variance by spreading them across multiple nodes.

We can formalize this by considering the load contribution of each key as a random variable [43]. If key k is replicated on $m(k)$ nodes and traffic is evenly split, then each node in its replica set receives $\lambda_k/m(k)$, and other nodes receive 0. If replica sets are chosen uniformly at random, then the probability a given node receives part of k is $m(k)/N$. The expected contribution of k to a given node's load is $(m(k)/N)\cdot(\lambda_k/m(k)) = \lambda_k/N$, independent of $m(k)$, so the mean load is unchanged. The variance contribution is

$$\mathrm{Var}\big[\mathrm{contrib}_k\big] = \frac{m(k)}{N}\left(\frac{\lambda_k}{m(k)}\right)^2 - \left(\frac{\lambda_k}{N}\right)^2 = \frac{\lambda_k^2}{N m(k)} - \frac{\lambda_k^2}{N^2}. \tag{9}$$

Summing over keys and ignoring small terms yields load variance approximately proportional to $\sum_k \lambda_k^2/(N m(k))$. This indicates that increasing $m(k)$ for large λ_k reduces variance significantly. A controller that targets hot keys is therefore aligned with variance reduction [44]. The diminishing returns are explicit: doubling replicas halves the variance contribution for that key.

Variance reduction is related to tail latency because the most loaded node tends to dominate p99 latency when requests are routed to a node that may be overloaded. In a system where routing can choose among replicas based on load, the maximum load is less directly relevant, but reducing variance still reduces the frequency of saturation events.

We next incorporate queueing delay. For an $M/M/1$ approximation at each node, the response time distribution has mean $1/(C-\Lambda_i)$ and tail quantiles that increase sharply as Λ_i approaches C. For an $M/G/1$, the tail can be heavier depending on service variability. Nonetheless, the utilization $\rho_i = \Lambda_i/C$ remains the central parameter [45]. If replication reduces Λ_i for overloaded nodes, it reduces both mean and tail. The controller's congestion penalty $\phi(\rho)$ captures this sensitivity.

Consider a single hot key h with rate λ_h that is initially served by one node. If that node's baseline load from other keys is Λ_0, then utilization is $\rho = (\Lambda_0 + \lambda_h)/C$. If we add one replica on another node and split h evenly, then the first node's utilization becomes $\rho_1 = (\Lambda_0 + \lambda_h/2)/C$ and the second node's becomes $\rho_2 = (\Lambda_0' + \lambda_h/2)/C$, where Λ_0' is the baseline load on the second node. The reduction in congestion penalty is

$$\Delta \approx \phi\left(\frac{\Lambda_0 + \lambda_h}{C}\right) - \phi\left(\frac{\Lambda_0 + \lambda_h/2}{C}\right) - \phi\left(\frac{\Lambda_0'}{C}\right) + \phi\left(\frac{\Lambda_0' + \lambda_h/2}{C}\right). \tag{10}$$

When the original node is near saturation and the target node has headroom, Δ is large [46]. This motivates placing replicas on underutilized nodes rather than simply increasing replica count without regard to location. The marginal benefit depends on the slope of ϕ at the current utilizations, which is captured by congestion prices ψ_i in the algorithm.

Now incorporate network topology. Let there be R racks. Suppose requests originate from racks according to $o_r(k)$. If the inference frontends are rack-local, then routing to a replica in the same rack avoids cross-rack hops. For a key with concentrated origin distribution, placing at least one replica in the dominant rack can reduce network latency and cross-rack bandwidth [47]. For a key with dispersed origins, a single rack-local replica cannot satisfy all requests, so additional replicas across racks may be beneficial. This creates a tradeoff: distributing replicas across racks reduces network cost but may reduce the ability to balance load within a rack if nodes there are already busy. The cost term involving $d(r,i)$ captures this.

A more detailed model treats network as a shared resource with capacity constraints. Let $B_{r\to r'}$ be available bandwidth between racks r and r', and let $T_{r\to r'}$ be traffic induced by embedding lookups. Then one could impose constraints $T_{r\to r'} \le B_{r\to r'}$ and add queueing delay in network switches. Such modeling is complex, but the controller can approximate it through a congestion price on cross-rack traffic derived from measured utilization. Let $\chi_{r\to r'}$ be a price that increases when links approach saturation. The network term in (1) can then be modified to include χ:

$$\text{network cost} \approx \sum_k \sum_r \sum_i \hat{\lambda}_k o_r(k) p_{k,i} \left(d(r,i) + \chi_{r\to r(i)} \right) \tag{11}$$

where $r(i)$ is the rack of node i. This makes routing and placement responsive to network hotspots [48].

We now consider update propagation capacity. Suppose each node can process updates at rate U, and updates for key k have payload size $s(k)$. If $m(k)$ replicas exist, total update bytes processed is $\mu_k s(k) m(k)$. The aggregate update bandwidth required is $\sum_k \mu_k s(k) m(k)$. If this exceeds the provisioning, lags grow. A stability condition for updates at node i is $\sum_{k:i\in\mathcal{P}(k)} \mu_k < U_i$ and similarly for byte bandwidth. Because replica placement can concentrate updated keys on certain nodes, the controller should also balance update load. This can be done by introducing an update congestion price υ_i analogous to read congestion, based on measured update lag or update CPU usage [49]. The marginal gain formula (6) then includes a term $-\hat{\mu}_k \upsilon_j$, discouraging placement on nodes that are update-congested.

To integrate these considerations, we can view the system as having two resource dimensions per node: read capacity and update capacity, each with its own utilization and penalty. A combined objective can use $\phi_r(\rho_i^{(r)}) + \phi_u(\rho_i^{(u)})$ and a combined price. This adds complexity but aligns with observed production behavior where update backlogs can degrade quality even if read latency remains acceptable.

Finally, we model placement actuation delay and its effect on control stability. Let $T_{\text{copy}}(k)$ be the time to install a new replica of key k, which depends on payload size and available bandwidth. Let $T_{\text{warm}}(k)$ be the time to warm caches and validate. The total actuation delay is $T_{\text{act}}(k) = T_{\text{copy}}(k) + T_{\text{warm}}(k)$. If workload changes faster than T_{act}, then aggressive replication can chase a moving target and waste resources. Therefore the controller's admission and placement changes should be tuned to the actuation timescale. A useful heuristic is to admit keys whose hotness is expected to persist longer than T_{act} and to prefer routing adjustments for shorter spikes. This can be formalized by estimating a persistence probability for each key based on recent volatility, and scaling marginal gains by that probability. If π_k is the estimated probability that key k remains hot over the next epoch, then the effective marginal gain is $\pi_k \Delta_{k\to j}$. This reduces churn for volatile keys.

The analytical model thus supports several design principles: replicate keys in proportion to their squared demand to reduce variance, place replicas on nodes with low congestion prices to reduce tail, distribute replicas across racks according to origin distribution to reduce network cost, incorporate update capacity and lag into placement to maintain freshness, and stabilize control by accounting for actuation delays and switching costs [50]. The proposed algorithm implements these principles in an online, telemetry-driven manner.

7. Evaluation Methodology, Metrics, and Expected Behavioral Regimes

A rigorous evaluation of adaptive replica placement requires measuring not only average improvements but also robustness under non-stationary demand, operational safety under failures, and resource tradeoffs. In production environments, the primary objective is typically tail latency for end-to-end inference or for embedding lookup RPCs, along with availability and bounded staleness. Secondary objectives include network utilization, memory footprint, update lag, and controller overhead.

The evaluation can be structured using a combination of replay-driven simulation, testbed experimentation, and controlled production experiments. Replay-driven simulation uses recorded lookup traces from a marketplace to drive a model of the embedding serving cluster. The trace includes key IDs, timestamps, origin rack or frontend group, and table identity [51]. If the trace cannot include raw IDs due to privacy, it can be hashed consistently to preserve skew properties. Simulation then applies a placement and routing policy and measures predicted node loads and network traffic. Queueing delay can be approximated using a service time model calibrated from profiling. While simulation cannot perfectly capture all system effects, it is valuable for exploring parameter sensitivity and worst-case scenarios, such as flash sales that drive sudden skew changes.

A testbed provides higher fidelity by running the embedding service stack with synthetic or replayed traffic. The testbed can include a subset of nodes with realistic network topology emulation and can measure actual tail latencies and update propagation. The primary limitation is scale: heavy-hitter behavior and contention can differ at smaller scale [52]. Therefore, testbed experiments are complemented by production experiments with careful safeguards. Production evaluation can be done via canary deployment where a fraction of traffic is served by the adaptive policy, comparing to baseline using matched cohorts. Because recommendation systems can influence user behavior, careful experiment design is needed to avoid confounding. For embedding latency, however, the metric is infrastructure performance, which

can be evaluated more directly, assuming the policy does not change model outputs beyond freshness differences.

Key metrics include p50, p90, p99, and p99.9 lookup latency per table and aggregated across tables, as well as end-to-end inference latency. Network metrics include cross-rack bytes per second, top-of-rack switch utilization, and packet loss indicators. Memory metrics include total memory used for adaptive replicas, fragmentation overhead, and eviction rates [53]. Update metrics include propagation lag distribution per table, percentage of reads served with version older than a threshold, and update CPU utilization. Controller metrics include number of replica changes per epoch, state transfer bytes, and time to convergence after a demand shift.

The evaluation should cover distinct workload regimes characteristic of e-commerce marketplaces. One regime is steady-state diurnal cycles where the hot set changes gradually. In this regime, the adaptive controller should provide modest improvements by smoothing load and keeping p99 stable as traffic rises and falls. Another regime is event-driven spikes where a small set of items becomes extremely hot, such as during limited-time promotions. Here the key behavior is whether the controller can react within the actuation delay and whether it can prevent localized overload [54]. A third regime is distributed hotness where many keys are moderately hot, such as during broad seasonal shopping when attention is spread. In this case, replication of a small set may not be sufficient; the controller must allocate memory across more keys or rely more on routing. A fourth regime involves correlated hotness across tables, for example when a user cohort surges and both user embeddings and item embeddings are hot. Multi-table fairness becomes important here to avoid over-optimizing one table at the expense of another.

Behavior under failures is also critical. Rack failures, node restarts, and network partitions can suddenly reduce replica availability and increase load on remaining nodes. The controller should detect reduced capacity and either temporarily increase replicas for the most pressured keys or adjust routing to prevent saturation. Because placement changes take time, immediate mitigation relies on routing and on baseline replication [55]. The evaluation should therefore measure the time to recover to pre-failure p99 and the peak p99 during the incident.

Another important evaluation dimension is staleness. Adaptive replication should not materially increase stale reads beyond acceptable thresholds. If update rates are low for most replicated keys, staleness may remain unchanged, but in some workloads user embeddings can have frequent updates. Therefore the evaluation should stratify keys by update rate and measure staleness distributions separately. If the controller includes update-aware scoring, it should reduce replication for high-update keys unless read pressure is extreme. The resulting tradeoff can be quantified by plotting p99 latency improvement versus increase in update bandwidth and staleness [56].

The controller's stability parameters, such as hysteresis thresholds, switching penalties, and step-size limits, should be tuned using data. A common pitfall is setting thresholds based on average load rather than tail events. The evaluation should therefore include stress tests where traffic is scaled up and the skew is increased, to ensure that the controller does not thrash when the system is near capacity. It should also include low-load tests to ensure that the controller does not create unnecessary replicas simply because memory is abundant, since excess replication increases update overhead without commensurate latency benefit.

To analyze controller decisions, one can compute the marginal gains estimated by (6) and compare them to realized improvements after changes. This validation helps calibrate congestion prices and cost weights. For example, if the controller consistently overestimates benefits of adding replicas to certain racks, it may indicate that network cost is underweighted or that congestion prices are misestimated due to measurement lag. Similarly, if the controller avoids replicating keys that later cause overload, it may indicate that admission thresholds are too conservative or that the hotness estimator lags [57]. These diagnostics are important for production hardening.

In an e-commerce marketplace, there are also business-driven constraints. Certain entities may be subject to policy controls, and the system may need to ensure that adaptive behavior does not create unfairness across sellers or regions. The embedding layer is typically not directly responsible for such business logic, but adaptive placement can indirectly affect latency and thus user experience across segments. Therefore evaluation should include segment-level latency metrics, such as by region or by frontend, to ensure that improvements are not concentrated in one segment while others degrade. If the controller includes topology-aware placement, it may preferentially optimize for dominant racks. Fairness constraints in the objective can mitigate this by including weights for different origin groups [58].

While exact numerical results depend on deployment specifics, the expected qualitative outcomes are as follows. Under shifting hot sets, adaptive replication reduces the frequency and severity of hotspots, lowering p99 latency and reducing the variance of node utilization. Cross-rack traffic can decrease when replicas are placed closer to request origins, though it may increase if the controller prioritizes load balancing over locality in extreme situations. Update overhead increases with replication, but update-aware scoring and per-node update capacity pricing keep lags within bounds. Stability mechanisms reduce thrash and keep replica change rates manageable, at the cost of slower reaction to very short spikes. The tradeoff is acceptable when actuation delay is nontrivial, and routing can address short spikes.

8. Implementation Considerations in Production Recommendation Serving

An adaptive replica placement system must integrate with the realities of production serving, including heterogeneous hardware, evolving model schemas, multi-tenant resource governance, and operational safety [59]. This section discusses implementation considerations that influence algorithm design and the boundary between what is handled by the controller versus what is delegated to underlying storage and orchestration layers.

Embedding servers often store multiple tables, each with different vector sizes and access patterns. Memory allocation must therefore handle fragmentation. If embeddings are stored in a hash map with per-entry overhead, frequent insertions and deletions caused by replica churn can fragment memory and degrade performance. A production implementation typically uses a slab allocator or an arena per table, where embeddings are stored in fixed-size blocks to reduce fragmentation. When a replica is removed, its block can be returned to a free list, but immediate reuse may not be safe if readers can still access it. Therefore the system requires safe reclamation, such as epoch-based reclamation or reference counting [60]. This influences how quickly replicas can be removed and may introduce a delay between decision and actual memory recovery. The controller should account for this by treating memory usage as including pending-free blocks and by avoiding decisions that would exceed hard limits.

The placement controller's actuation must be reliable. Copying an embedding to a new node can fail due to network errors or node overload. Therefore replica installation should be idempotent and retriable. A common approach is to represent placement state in a control-plane store, such as a distributed key-value store, where the desired replica set is recorded. Each node runs an agent that watches desired assignments and performs local actions to converge [61]. This separates decision logic from execution and allows the system to tolerate controller restarts. The agent can report progress, such as bytes transferred and catch-up lag, enabling the controller to activate replicas only when ready.

Routing integration is another challenge. If routing is done by the inference frontends, they need up-to-date knowledge of replica sets. Disseminating frequent changes to many frontends can be expensive and can introduce inconsistent views, leading to failed requests. Therefore the system may use an indirection layer: a routing service or a consistent hashing ring that can be updated centrally. Alternatively, the embedding service can provide a redirect mechanism, where a frontend sends a lookup to a default node which then forwards to an appropriate replica [62]. Forwarding adds latency, so it is typically avoided on the steady state, but it can be acceptable during transitions. A hybrid approach uses frontends for normal routing and fallback forwarding for misses. The controller's stability mechanisms reduce the frequency of changes, making dissemination easier.

Batching is central in recommendation serving because each request includes many embedding lookups. Embedding services often support batch RPCs where multiple keys and tables are requested at once. Batch performance depends on how keys are grouped by destination node. If replica placement increases the number of destinations, it can increase fan-out, potentially increasing latency due to more RPCs [63]. Therefore placement must consider not only per-lookup latency but also batching efficiency. One approach is to co-locate replicas for keys that commonly co-occur in the same request. For example, item embeddings for items shown together in a carousel may be requested together. Co-locating them can reduce fan-out. This is related to graph partitioning based on co-access patterns, which is complex at scale. A practical compromise is to avoid scattering replicas excessively by limiting the number of racks used for a key and by preferring nodes that already serve many keys from the same origin group. The controller can include a fan-out penalty proportional to the expected number of distinct destination nodes per request [64]. While estimating this exactly is difficult, it can be approximated at the table level by measuring average destinations per batch and penalizing changes that increase it.

Compression and quantization are often used to reduce memory. If embeddings are stored in compressed form and decompressed on read, the service time increases. Replication then interacts with CPU load differently than with pure memory bandwidth. The controller's cost coefficients c_k can reflect this by table. Additionally, some systems store embeddings on GPUs for faster access, especially for very hot vectors. GPU memory is limited and expensive, so adaptive replication could be extended to decide which hot embeddings are promoted to GPU memory [65]. This introduces a tiering dimension beyond the scope of a pure CPU memory placement, but the same principles apply: treat GPU as a constrained resource with high benefit for the hottest keys, and use an online controller with switching costs. In marketplaces, GPU tiering may be used sparingly due to cost, but it is increasingly relevant.

Schema evolution is another production reality. Embedding dimensions may change, tables may be added or removed, and key encoding may evolve. Replica placement logic must be robust to such changes. This argues for table-level metadata that describes vector size, update semantics, and compatibility requirements. During schema transitions, mixed versions may coexist [66]. The controller must avoid replicating incompatible versions across nodes or must ensure that routing selects the correct version for the requesting model. This is typically handled by versioned table names or namespaces. The placement controller then operates within a namespace and can treat version rollouts as changes in table identity, with separate budgets and gradual migration.

Observability is essential. The controller relies on telemetry: per-node utilization, queueing metrics, network metrics, update lag, and approximate per-key rates. Instrumentation must be efficient and must not introduce significant overhead. For per-key telemetry, the system can report only the heavy hitters and aggregate the rest [67]. Sampling must be carefully implemented to avoid bias. Additionally, the controller's decisions should be explainable in operational terms: why a key was replicated, why it was removed, and what benefit was expected. This can be achieved by logging marginal gain components, including congestion price differences, distance costs, update costs, and memory prices. Such logs help diagnose misbehavior and support safe tuning.

Safety mechanisms are required to prevent the controller from causing outages. One mechanism is a global kill switch that reverts to baseline placement and routing. Another is guardrails on memory: the agent must refuse placements that exceed hard capacity, regardless of controller desire [68]. Similarly, guardrails on update lag can prevent adding replicas for frequently updated keys when lag is high. Rate limiting on replica changes ensures that the controller cannot saturate the network with state transfers. Canarying is also important: the controller can be deployed to a subset of tables or a subset of keys and gradually expanded.

Multi-tenancy and governance require quotas. Adaptive replicas consume memory that might otherwise be used for baseline tables or other tenants. Therefore budgets B_t should be enforced and reported. In some organizations, budgets are negotiated and tracked [69]. The controller can support this by computing per-tenant memory usage and providing levers to adjust weights. In addition, fairness can be enforced by including per-tenant weights in the objective or by limiting the fraction of adaptive memory any tenant can use.

Failure handling requires coordination with orchestration. When nodes are drained for maintenance, replicas on those nodes must be moved. The controller can treat drains as planned failures and proactively replicate hot keys elsewhere before the drain begins. This is a key operational benefit: planned maintenance can otherwise cause transient overload if hot keys lose replicas. The controller can use maintenance schedules from orchestration to precompute changes [70]. When unplanned failures occur, immediate mitigation is handled by routing and baseline replication; the controller then restores redundancy by adding replicas as capacity allows.

Finally, it is important to consider the interaction between embedding replication and upstream caching. Many systems include per-frontend caches for embeddings or for partial model results. If a key is cached upstream, embedding server load may drop, changing hotness. Conversely, caching policies may depend on embedding latency. Adaptive replication can therefore change cache hit rates. While modeling this precisely is complex, a production deployment should monitor cache metrics and consider them in hotness estimation [71]. One approach is to estimate demand at the point after cache, since that is what loads the embedding servers, and treat cache policy as exogenous. Another approach is to jointly tune caching and replication, which is beyond scope but compatible with the framework: caching is another form of replication at a different layer with different costs.

Overall, implementing adaptive replica placement requires careful integration with storage primitives, routing dissemination, update propagation, and operational safety. The algorithmic core must be simple enough to reason about and robust to noisy telemetry. The primal-dual structure with marginal gain evaluation, candidate restriction, and stability penalties is well-suited to these constraints.

9. Conclusion

Distributed recommendation serving systems in e-commerce marketplaces must deliver embedding vectors with low and predictable tail latency under highly skewed and rapidly changing workloads. Static sharding and fixed replication are often insufficient because hot keys can overload individual nodes and network links, while naive replication can waste memory and increase update overhead [72]. This paper presented an adaptive replica placement framework for hot embedding vectors that treats replication as an online, constrained optimization problem. The approach combines streaming hotness estimation with topology-aware placement and congestion-aware routing, using marginal gain approximations and primal-dual resource prices to scale decisions under memory and freshness constraints. Stability is addressed through hysteresis, switching costs, and rate limits that account for actuation delay and warmup behavior. Consistency and freshness considerations are incorporated via update-aware scoring and propagation-aware penalties, recognizing that replication increases write amplification and can affect staleness.

The resulting design emphasizes implementability in production: it restricts decisions to a managed hot set, uses telemetry readily available in serving stacks, and integrates with control-plane execution mechanisms that ensure idempotent replica installation and safe activation. Analytical modeling links replica multiplicity to load variance and queueing sensitivity, providing guidance for tuning and for understanding diminishing returns. The evaluation methodology highlights how to assess improvements across workload regimes characteristic of marketplaces, including event-driven spikes and correlated hotness across tables, while monitoring update lag and network utilization.

Adaptive replica placement is not a substitute for capacity planning or model-level optimization, but it is a practical control lever that can reduce hotspot-driven tail latency and improve robustness to non-stationary de-

mand when carefully constrained by memory, topology, and freshness requirements [73].

References

[1] *Clock Synchronization in Distributed Systems*, pp. 281–292. CRC Press, 10 2018.

[2] I.-A. Secara, *Challenges and Considerations in Developing and Architecting Large-scale Distributed Systems*, pp. 1–15. B P International (a part of SCIENCEDOMAIN International), 4 2023.

[3] R. Fronteddu, U. Ardinghi, L. Colombi, S. Dahdal, A. Morelli, M. Tortonesi, C. Stefanelli, and N. Suri, "Semantic information management systems," in *2025 International Conference on Military Communication and Information Systems (ICMCIS)*, pp. 1–9, IEEE, 5 2025.

[4] L. Sorokin, "Towards auditable distributed systems," 1 2022.

[5] J. Shi, Y. Yu, J. Gu, and Y. Xia, "A hardware-software co-design for efficient secure containers," in *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1229–1245, ACM, 3 2025.

[6] H. M. Shukur, S. R. M. Zeebaree, R. R. Zebari, O. M. Ahmed, L. M. Haji, and D. Abdulqader, "Cache coherence protocols in distributed systems," 6 2020.

[7] 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.

[8] F. Checconi, J. J. Tithi, and F. Petrini, "Ridgeline: A 2d roofline model for distributed systems," 1 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] K. Johari, G. K. Agarwal, and Y. D. Arya, "Server load balancing analysis for client task assignment in distributed systems," *ECS Transactions*, vol. 107, pp. 17605–17621, 4 2022.

[11] A. S. M. Noor, N. F. M. Zian, and F. N. M. S. Bahri, "Survey on replication techniques for distributed system," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, pp. 1298–1303, 4 2019.

[12] 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.

[13] I. Schagaev, *Distributed Systems: Maximizing Resilience*, pp. 325–342. Springer International Publishing, 7 2024.

[14] K. Habibi, "Web service replica selection analysis using a multiagent-based simulator," 5 2021.

[15] "Distributed system based on correlation sequencing algorithm," *Distributed Processing System*, vol. 2, 9 2021.

[16] X. Yuan and J. Yang, "Asplos - effective concurrency testing for distributed systems," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 1141–1156, ACM, 3 2020.

[17] 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.

[18] R. Ross, *The Business of Managing Distributed Systems*, pp. 10–1–10–14. Auerbach Publications, 7 2019.

[19] 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.

[20] J. Yuan, A. Le-Tuan, M. Hauswirth, and D. Le-Phuoc, "Cooperative students: Navigating unsupervised domain adaptation in nighttime object detection," in *2024 IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1–6, IEEE, 7 2024.

[21] J. D. L. García and S. Imle, "A novel diagnosis system concept for safe distributed systems," *Tecnología en Marcha*, vol. 34, pp. 19–27, 11 2021.

[22] C. Melo, G. Gonçalves, F. A. Silva, L. Feitosa, I. Fé, A. Soares, E. Choi, T. A. Nguyen, and D. Min, "Optimal resource utilization in hyperledger fabric: A comprehensive spn-based performance evaluation paradigm," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, pp. 1–7, IEEE, 5 2024.

[23] "Cryptographic foundation for secure and scalable distributed systems," 9 2023.

[24] A. Prasadan, "Sketching algorithms in distributed systems," 10 2021.

[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] A. Arman, P. Bellini, D. Bologna, P. Nesi, G. Pantaleo, and M. Paolucci, "Automating iot data ingestion enabling visual representation.," *Sensors (Basel, Switzerland)*, vol. 21, pp. 8429–8429, 12 2021.

[27] A. Kolar-Požun and G. Kosec, "Computational electromagnetics with the rbf-fd method," in *2023 8th International Conference on Smart and Sustainable Technologies (SpliTech)*, pp. 1–4, IEEE, 6 2023.

[28] A. Kumar, P. Hegde, R. Vaze, A. Alloum, and C. Adjih, "Breaking the unit throughput barrier in distributed systems.," 10 2020.

[29] L. Nenov, "Reinforcement learning for key management in distributed systems," in *2024 32nd National Conference with International Participation (TELECOM)*, pp. 1–5, IEEE, 11 2024.

[30] J. C. G. Vargas, R. Fabregat, A. Carrillo-Ramos, and T. Jové, "Survey: Using augmented reality to improve learning motivation in cultural heritage studies," *Applied Sciences*, vol. 10, pp. 897–, 1 2020.

[31] M. Hörmann, M. Marx, M. Segata, M. Zucchelli, and F. Kargl, "Poster: Mbhws — a mountainbike-to-hiker warning system," in *2025 IEEE Vehicular Networking Conference (VNC)*, pp. 1–2, IEEE, 6 2025.

[32] N. V. Patel, "Adaptive experimentation in marketplaces: Balancing exploration and revenue optimization," *Frontiers in Health Informatics*, vol. 11, pp. 760–786, 2022.

[33] C. J. Underhill, J. A. A'Vant, and K. T. H. Cao, "Monitoring distributed systems with open-source software," 8 2019.

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

[36] I. Siddique, "Libguides: Distributed systems: Home," 7 2018.

[37] *Scalable Distributed Systems*, pp. 2290–2290. Springer New York, 6 2018.

[38] T. Simeonova, "Peculiarities of distributed systems, scada and iot," *Yearbook Telecommunications*, vol. 7, pp. 65–70, 8 2021.

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

[40] 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.

[41] W. N. S. W. Nik, "A survey on data replication and resource management in distributed systems," *International Journal of Advanced Trends in Computer Science and Engineering*, pp. 2638–2646, 10 2019.

[42] B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, and E.-R. Olderog, "Global winning conditions in synthesis of distributed systems with causal memory.," 7 2021.

[43] 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.

[44] X. Wu, J. He, L. Huang, C. Fu, and W. Wang, "Wbsan: Webassembly bug detection for sanitization and binary-only fuzzing," in *Proceedings of the ACM on Web Conference 2025*, pp. 3311–3322, ACM, 4 2025.

[45] E. P. G. Correia, "Distributed system tests framework," 7 2019.

[46] 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.

[47] P. A. Sarvari, S. Martin, D. Khadraoui, and G. Baskurt, *Distributed Systems and Surge of Entrepreneurial Activities*, pp. 50–62. Springer Nature Switzerland, 3 2024.

[48] N. V. Patel, "Estimating heterogeneous treatment effects of driver incentives in ride-hailing platforms," *Northern Reviews on Algorithmic Research, Theoretical Computation, and Complexity*, vol. 9, no. 11, pp. 16–43, 2024.

[49] M. Dahlmanns, C. Sander, R. Decker, and K. Wehrle, "Secrets revealed in container images: An internet-wide study on occurrence and impact," in *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pp. 797–811, ACM, 7 2023.

[50] K. Schlesinger, "3d models of distributed systems," 1 2020.

[51] 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.

[52] 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.

[53] 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.

[54] M. Han, R. Chen, W. Shen, H. Zhang, J. Yang, and H. Chen, "Real-time, work-conserving gpu scheduling for concurrent dnn inference," *ACM Transactions on Computer Systems*, vol. 44, pp. 1–42, 11 2025.

[55] 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.

[56] B. Sedlak, V. C. Pujol, P. K. Donta, and S. Dustdar, "Markov blanket composition of slos," in *2024 IEEE International Conference on Edge Computing and Communications (EDGE)*, pp. 128–138, IEEE, 7 2024.

[57] A. Bolfing, *Distributed Systems*, pp. 143–198. Oxford University PressOxford, 9 2020.

[58] J. Stój, A. Ziębiński, and R. Cupek, "Fpga based industrial ethernet network analyser for real-time systems providing openness for industry 4.0," *Enterprise Information Systems*, vol. 16, pp. 1711–1731, 7 2021.

[59] A. A. Naji, M. A. Saeed, A. H. Almagashi, A. M. Ahmed, H. S. Al-Atefi, M. N. Khaled, and O. M. Rashed, "Enhancing manets security against black hole attacks through time-based node analysis in aomdv protocol," in *2024 4th International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pp. 1–6, IEEE, 8 2024.

[60] 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.

[61] G. P. Pinto, P. K. Donta, S. Dustdar, and C. Prazeres, "A systematic review on privacy-aware iot personal data stores.," *Sensors (Basel, Switzerland)*, vol. 24, pp. 2197–2197, 3 2024.

[62] J. Flak, T. Skowron, R. Cupek, M. Fojcik, D. Caban, and A. Domański, "Zigbee network for agv communication in industrial environments," in *2023 IEEE 10th International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 1–9, IEEE, 10 2023.

[63] N. V. Patel, "Estimating the causal effects of pricing interventions in two-sided marketplaces under interference and spillovers," *Frontiers in Health Informatics*, vol. 13, no. 8, pp. 7572–7601, 2024.

[64] 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.

[65] I. Čilić, A. Lackinger, P. A. Frangoudis, I. P. Žarko, A. Furutanpey, I. Murturi, and S. Dustdar, "Reactive orchestration for hierarchical federated learning under a communication cost budget," in *2025 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pp. 1–7, IEEE, 5 2025.

[66] A. Poshtkohi and M. B. Ghaznavi-Ghoushchi, *IoT and Distributed Systems*, pp. 15–22. Auerbach Publications, 2 2023.

[67] J. Jaskolka, "Evaluating the exploitability of implicit interactions in distributed systems.," 6 2020.

[68] M. A. Guimarães and R. J. D. A. Macêdo, "Energy-efficient ehealth monitoring with lpwan," in *2024 XIV Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 1–6, IEEE, 11 2024.

[69] A. Q. M. Algaolahi, A. A. Hasan, A. A. Sallam, A. M. Sharaf, A. A. Abdu, and A. A. Alqadi, "Port-scanning attack detection using supervised machine learning classifiers," in *2021 1st International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pp. 1–5, IEEE, 8 2021.

[70] 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.

[71] R. Glebke, M. Winck, S. Lenz, M. Stoffers, and K. Wehrle, "Scalable supply and demand matching involving up-to-date information from individual prosumers," in *Proceedings of the 16th ACM International Conference on Future and Sustainable Energy Systems*, pp. 577–589, ACM, 6 2025.

[72] V. A. Kolesnikov, *CMIS - Autonomous distributed systems and their simulation.* 6 2020.

[73] Y. Wang, X. Zhu, R. Mahajan, and S. Wang, "Programmable and adaptive scheduling for distributed systems," in *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, pp. 193–200, ACM, 11 2025.