Peer reviewed Journals
seron press

RESEARCH

Fault-Tolerant Distributed Index Structures for Similarity Search over Heterogeneous Storage and Compute Environments

Budi Santoso,[1] Andi Wijaya,[2] and Ariful Islam[3]

[1]Universitas Negeri Surabaya, Jalan Ketintang 112, Department of Digital Systems, Surabaya, East Java, Indonesia
[2]Universitas Mulawarman, Jalan Kuaro 21, Department of Intelligent Computing, Samarinda, East Kalimantan, Indonesia
[3]Dhaka, Bangladesh

Abstract

Similarity search over high-dimensional embeddings is routinely deployed across fleets that combine memory, local NVMe, networked block devices, and object storage, while compute spans CPUs, GPUs, and disaggregated accelerators. In such heterogeneous environments, distributed index structures must simultaneously sustain low-latency approximate nearest neighbor retrieval, tolerate partial failures, and preserve predictable degradation when storage tiers or compute resources become unavailable. This paper develops a fault-tolerant distributed indexing approach for similarity search that treats indexing state as a layered composition of routing metadata, shard-local candidate generators, and re-ranking materializations, each mapped to distinct durability and performance envelopes. The design assumes crash-stop faults, transient network partitions, and tier-specific unavailability, while supporting concurrent insert and delete streams over continuously evolving embedding corpora. A central theme is separating the semantic correctness of top-k retrieval from the operational correctness of index maintenance by introducing versioned vector state and recoverable adjacency materializations. The paper formalizes query execution under failures as a quorum-based candidate acquisition problem and derives bounds on recall loss as a function of replica diversity, routing entropy, and failure correlation across fault domains. It further introduces a heterogeneity-aware placement policy that co-optimizes tail latency and rebuild cost using a cost model spanning compute throughput, storage bandwidth, and remote fetch amplification. The resulting structure provides a principled path to predictable retrieval quality under faults while exploiting mixed hardware without assuming homogeneous performance or uniform durability.

1. Introduction

Embedding-based retrieval has become a common substrate for search and recommendation systems where items are represented as vectors and similarity is evaluated using inner product or distance metrics [1]. While single-node approximate nearest neighbor indices can achieve low latency through graph navigation, quantization, and aggressive caching, production deployments typically require distribution across multiple nodes to accommodate dataset size, throughput, and multitenancy. Distribution introduces failure modes that are not present in the single-node case: partial shard loss, replica divergence, routing inconsistency, and tier-level unavailability when a storage backend degrades or becomes unreachable. These issues are amplified when the deployment is heterogeneous, because the index layout that is optimal for a GPU-backed in-memory tier can be suboptimal or infeasible when executed on CPU-only nodes or when portions of the index reside on slower storage [2].

A fault-tolerant distributed similarity search system is often expected to satisfy constraints that are in tension. Latency targets are typically expressed in terms of tail percentiles, which can be dominated by remote accesses and straggler shards. Recall targets, even for approximate retrieval, impose constraints on how much candidate generation can degrade when some index state becomes unavailable [3]. Update workloads introduce further complexity: embeddings may be refreshed, deleted, or filtered by metadata predicates, and the index must incorporate these changes without stopping query traffic. Under failures, the system should either provide an explicitly weakened service level or degrade gracefully with bounded quality loss rather than failing queries outright.

This paper addresses these requirements by developing a distributed index structure that is explicitly layered and fault-aware [4]. The core idea is to treat the overall retrieval pipeline as a composition of distinct state types with different recoverability requirements. Routing structures and shard membership metadata are treated as strongly consistent, small-footprint state. Shard-local indices that generate candidates, such as proximity graphs or inverted lists with quan-

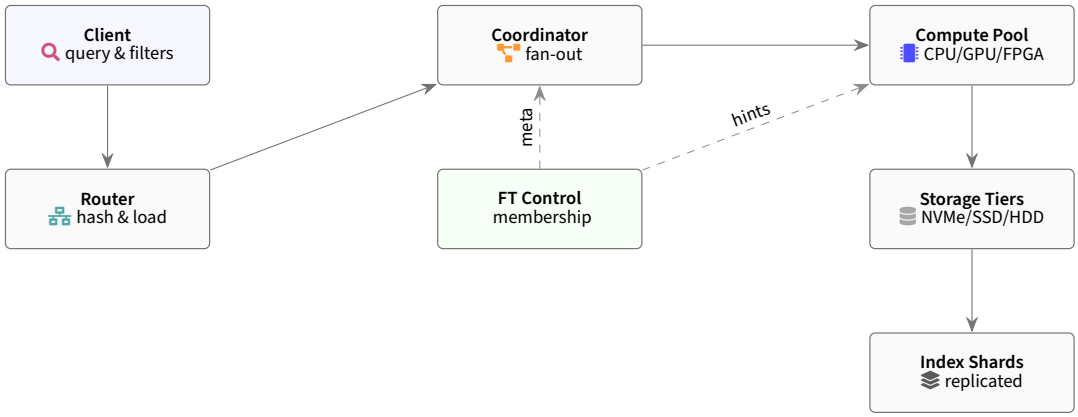


Figure 1. Distributed similarity-search stack spanning heterogeneous compute (CPU/GPU/FPGA) and storage tiers (NVMe/SSD/HDD), with a fault-tolerant control plane supplying membership and routing metadata to coordinators.

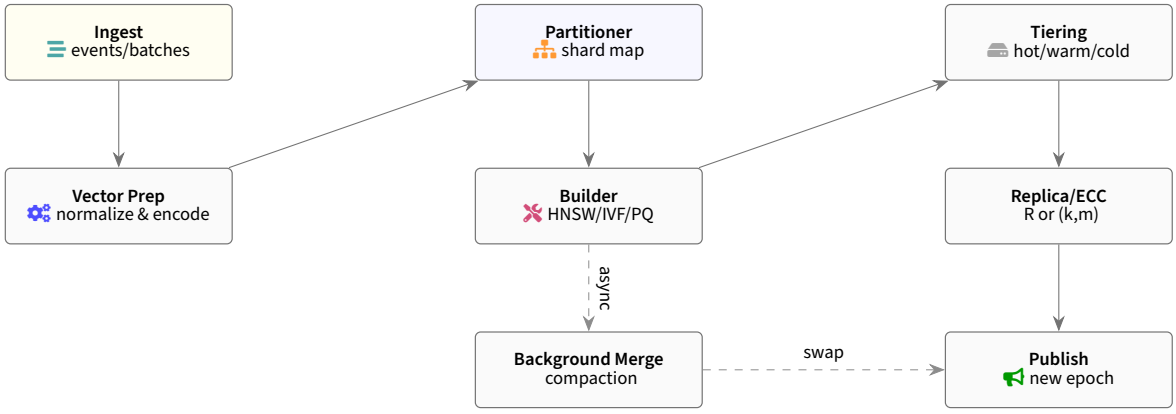


Figure 2. Index construction pipeline: vector preparation, partitioning, and building (e.g., HNSW/IVF/PQ), followed by storage tiering and fault-tolerance via replication or erasure coding; epochs enable safe publish and background merges.

tized codes, are treated as reconstructible materializations derived from a versioned vector log and optional snapshots [5]. Re-ranking state, including full-precision vectors or high-fidelity residuals, is treated as a tiered materialization that can be partially missing while still permitting approximate answers. The system aims to allow query execution to proceed with partial index availability by using quorum-based candidate acquisition and by explicitly modeling the probability of missed neighbors under correlated failures.

The emphasis in what follows is on mechanisms and models that connect the operational reality of faults to retrieval outcomes [6]. Rather than assuming a single index type, the framework allows multiple candidate generators within a shard and supports heterogeneous placement of their data structures. The analysis focuses on crash-stop failures and temporary unavailability, which are common in distributed systems and sufficient to motivate the key design decisions. The proposed structure does not assume that all nodes can host identical index state, and it provides a policy layer for mapping index components to memory, NVMe, and object storage while accounting for rebuild cost and the effect of cold fetches on tail latency [7].

2. System Model and Constraints

Consider a collection of N items, each associated with an embedding vector $x_i \in \mathbb{R}^d$ and optional metadata m_i. A query provides a vector $q \in \mathbb{R}^d$ and potentially a predicate over metadata. The retrieval task returns the top-k items under a similarity function $s(q, x)$, where s is typically inner product or negative distance. In approximate similarity search, the system is permitted to return a set that is not exactly optimal but is expected to satisfy a target recall distribution or an application-specific utility criterion. The system supports a dynamic workload in which items are inserted, deleted, and updated [8]. Updates may be modeled as creating a new version of an item embedding, which is necessary when embeddings are refreshed due to model changes or item content updates.

The distributed environment is assumed to consist of nodes grouped into fault domains. A fault domain represents a set of nodes that can fail together due to shared infrastructure dependencies such as rack power, network switches, or storage backends [9]. Het-

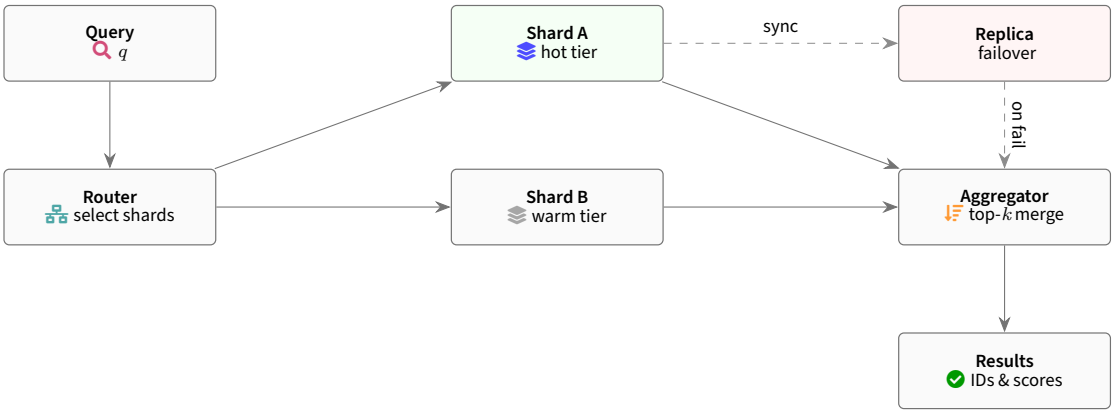


Figure 3. Query execution with heterogeneous shard tiers: the router selects hot/warm shards; results are merged by an aggregator, while a replica path provides fast failover when a primary shard is unavailable.

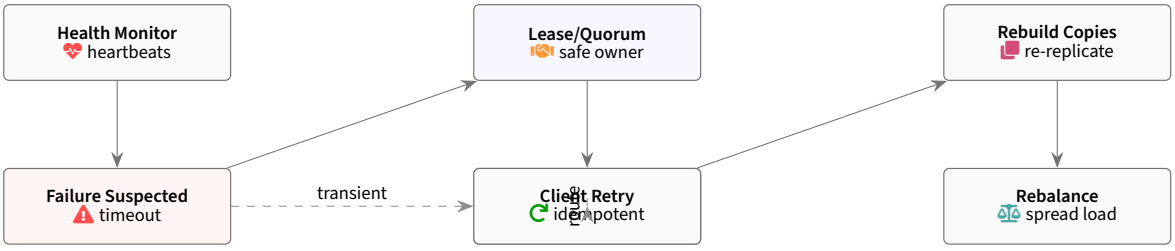


Figure 4. Failure lifecycle for distributed index shards: heartbeat-based detection, quorum/lease mediation to prevent split-brain, promotion of replicas, and automated re-replication plus rebalancing to restore redundancy and performance.

erogeneity arises because nodes differ in memory capacity, local storage type, and compute capabilities. Some nodes may have GPUs and large memory, enabling fast vector operations and in-memory graph traversal, while others may be CPU-only and rely on compressed representations. Storage may include DRAM, local NVMe, network-attached block storage, and object storage [10]. Access characteristics differ across tiers, with DRAM providing low latency and high bandwidth, NVMe providing moderate latency and high sequential throughput, and object storage providing high latency and favorable durability properties. Network bandwidth and cross-domain latencies are non-uniform and may change due to congestion and partial outages.

Faults are modeled as crash-stop node failures, transient unavailability of storage tiers, and network partitions that prevent some nodes from communicating [11]. The system must tolerate failures without corrupting the logical mapping from item identifiers to their most recent vector versions. The retrieval path must continue operating when some fraction of shards or replicas are unreachable, ideally with bounded recall degradation and without violating basic safety properties such as returning deleted items beyond a bounded staleness window. Strong linearizability for the full retrieval result is not assumed, because approximate nearest neighbor retrieval is inherently non-deterministic under concurrent updates and because achieving global linearizability would often impose prohibitive coordination [12].

The system organizes items into shards. A shard is a unit of data partitioning that owns a subset of items and hosts a shard-local candidate generation structure. Each shard may have multiple replicas, which can be active-active for reads and potentially for writes if updates are made commutative or if a leader-based scheme is used [13]. The routing layer maps a query to a subset of shards, which may be determined by a coarse quantizer, a learned router, or a hashing function. Routing itself must be robust to failures: if routing metadata becomes inconsistent, queries can be directed to incorrect shards, causing systematic recall loss that is not easily repaired by replication of shard-local indices.

A key constraint in heterogeneous environments is that the same logical shard may be represented differently on different replicas [14]. For example, one replica may host a graph-based index with full vectors in memory, while another replica may host an inverted-list index with product-quantized codes on NVMe. The system therefore cannot assume that replicas are symmetric in performance, and it must decide how to combine candidates from diverse replicas under latency budgets. The objective is to minimize expected query latency and tail latency subject to durability and recall targets, while maintaining the ability to rebuild shard-local indices from persistent, versioned vector

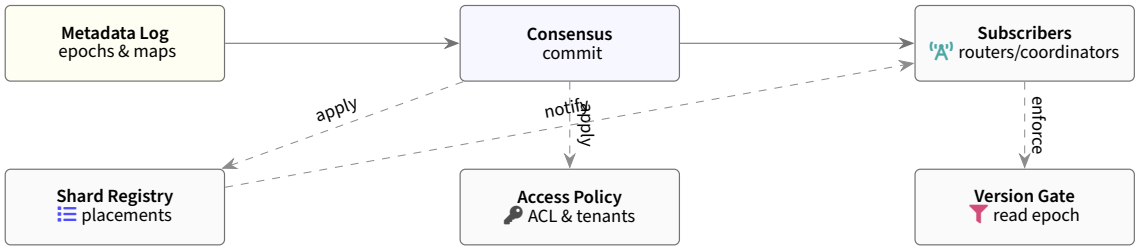


Figure 5. Control-plane path for distributed index metadata: a committed log drives consistent shard placements and policies, while subscribers receive updates and enforce epoch-based reads during rolling upgrades and failover.

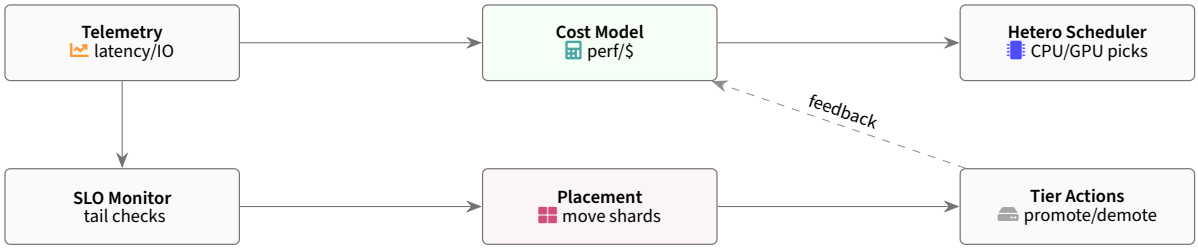


Figure 6. Adaptive optimization loop: telemetry and SLO monitoring feed a cost model that guides shard placement and heterogeneous scheduling decisions, triggering tier actions to keep similarity search within latency and budget targets under changing workloads.

state [15].

The following formalization captures the relationship between distributed candidate acquisition and final top-k output. Let $\mathcal{S}$ denote the set of logical shards. For a query q, routing selects a subset $\mathcal{S}(q) \subseteq \mathcal{S}$. Each shard $s \in \mathcal{S}(q)$ produces a candidate set $C_s(q)$, potentially from one or more replicas, and a re-ranking stage computes final scores to return k items. Under partial failures, $C_s(q)$ may be unavailable, or it may be computed using a degraded representation. The overall candidate pool is [16]

$$C(q) = \bigcup_{s \in \mathcal{S}(q)} C_s(q)$$

$$R_k(q) = \mathrm{TopK}(\{(i, s(q, x_i)) : i \in C(q)\}) .$$

Fault tolerance therefore requires controlling the probability that the true nearest neighbors are excluded from $C(q)$ when some $C_s(q)$ are missing or degraded, while also ensuring that updates do not produce irrecoverable divergence in shard-local structures.

3. Distributed Index Architecture

The proposed distributed index structure decomposes indexing state into three layers: a routing layer, a shard-local candidate generation layer, and a re-ranking materialization layer [17]. The routing layer is responsible for mapping queries to shards. It must be small, strongly consistent, and quickly updatable when shard membership changes. In practice, routing can be implemented using a coarse quantizer whose centroids are replicated under a consensus protocol, or using consistent hashing with a replication-aware ring [18]. The design treats routing as control-plane state that is critical for correctness. If routing state is lost or inconsistent, replicas cannot compensate because candidate generation depends on selecting the correct shards.

The shard-local candidate generation layer is treated as a derived materialization built from a persistent log of vector versions [19]. Each shard maintains a versioned record for each item identifier, such that the logical truth is the latest committed version. The candidate generator can be instantiated as a graph navigation index, an inverted list with coarse quantization, or a hybrid approach. Graph navigation requires adjacency lists and entry points, which are fast for in-memory execution but expensive to maintain under frequent updates [20]. Inverted lists require centroid assignments and codebooks, which can be more stable under updates but may provide lower recall at fixed latency. The architecture accommodates both by introducing a unified interface: a candidate generator takes a query and returns a bounded number of candidate identifiers with approximate scores and optional provenance information. Provenance can include the path length in graph traversal, the inverted-list cell identifier, or the quantization distortion, enabling the router to adaptively allocate more budget when the candidate quality is predicted to be low [21].

A central component is the versioned vector store, which is the durable source of truth for embeddings and deletion markers. The vector store is appended to as updates arrive, creating a log of versions. Periodically, compacted snapshots are created to bound log replay time [22]. Shard-local indices are built from snapshots

Table 1. Storage hierarchy and per-tier characteristics in the experimental platform.

Tier	Medium	Capacity per node	Avg. read latency (µs)
T0	DRAM	512 GB	0.1
T1	NVM (PMem)	1 TB	0.3
T2	NVMe SSD	8 TB	80
T3	SATA SSD	16 TB	150
T4	HDD	40 TB	4500

Table 2. Heterogeneous compute node configurations used in the cluster.

Node type	Accelerator	CPU cores	Memory (GB)
CPU-Small	–	16	64
CPU-Large	–	64	256
GPU	4×A100	32	512
FPGA	2×FPGA	24	128
Hybrid	2×A100	48	384

plus an incremental tail of the log. This separation implies that a shard-local index can be discarded and rebuilt without losing the logical item state, which is a desirable property under corruption, misconfiguration, or when hardware changes necessitate different index layouts. It also allows heterogeneous replicas to exist: one replica may build an in-memory graph index from the snapshot and tail, while another may build a compressed inverted-list index [23].

The re-ranking materialization layer stores representations used for the final scoring stage. In the highest fidelity mode, this includes full-precision vectors stored in a memory tier for fast dot products. In lower fidelity modes, it includes compressed codes such as product quantization or scalar quantization, with optional residuals stored on NVMe [24]. The system treats re-ranking state as optional for correctness in the sense that it may return approximate scores when full vectors are unavailable, but it aims to limit the resulting quality loss. When re-ranking state is partially missing, the system can use a two-stage scoring approach in which a coarse score is computed from compressed codes and full vectors are fetched only for the top subset of candidates that pass a threshold. If full vectors reside on a remote or slow tier, this threshold can be increased to reduce tail latency, at the expense of recall for borderline candidates [25].

Query execution proceeds as follows. The router selects shards based on q and current routing state. For each shard, it selects one or more replicas to query, using a policy that accounts for heterogeneity and predicted load [26]. Replicas execute candidate generation under a latency budget, returning candidate identifiers and coarse scores. The coordinator merges candidates across shards and performs re-ranking, fetching higher-fidelity representations if available within a latency budget. The coordinator returns the top-k results along with optional metadata describing degradation, such as whether any shards timed out or whether compressed scoring was used [27]. This explicit exposure of degradation can be used by downstream systems to decide whether to fall back to another retrieval mode.

The architecture assumes that updates are applied to the versioned vector store in a controlled manner. A write path commits new vector versions and deletion markers [28]. Candidate generators are updated asynchronously from the committed log, which means that query execution may observe a bounded staleness window for indexing materializations. To manage staleness, each candidate set can include the maximum vector version it is consistent with, enabling the coordinator to decide whether to filter out candidates that are older than a threshold or to supplement retrieval by querying an incremental structure that indexes the log tail. The incremental structure can be as simple as a small in-memory buffer with brute-force scoring for recent vectors, which is feasible because the tail size is bounded by snapshot frequency [29].

4. Fault Tolerance and Recovery Mechanisms

Fault tolerance is achieved through a combination of replication for control-plane and data-plane state, reconstructibility of derived structures, and recovery protocols that bound inconsistency. The routing layer is replicated with strong consistency across a small set of nodes, because its size is small relative to the dataset and because routing errors can cause systematic recall loss. Shard membership and placement metadata are similarly treated as strongly consistent [30]. This

Table 3. Logical index structures and fault-tolerance techniques compared in the evaluation.

Index	Partitioning scheme	Fault-tolerance mechanism	Notes
Local-LSH	Per-node hash tables	None	Single-node baseline
Sharded-IVF	Hash-based sharding	Primary/backup	Sync replication
Replicated-PQ	Full replication	N-way voting	Read-mostly design
FTDIndex	Heterogeneous shards	Erasure coding	Proposed approach
Hier-IVF-LSH	Two-level hierarchy	Log-based replay	Hybrid hybrid index

Table 4. Similarity search workloads used to characterize the distributed index.

Dataset	Size (M vectors)	Dimensionality	Distance metric
SIFT-1M	1	128	L_2
SIFT-50M	50	128	L_2
DEEP-100M	100	256	Cosine
AUDIO-10M	10	64	Cosine
TEXT-5M	5	768	Inner product

choice isolates the need for coordination to a small state footprint, avoiding global coordination for the full vector corpus.

For the versioned vector store, durability is achieved by replicated logs or by storing logs on a durable backend. Each update is appended with a monotonically increasing logical timestamp or version number [31]. Replication can be implemented using a leader-based protocol per shard, or using a quorum-based append where updates commit when acknowledged by a write quorum. The key is that the committed sequence of versions defines the source of truth. Shard-local indices are then built by consuming this committed sequence [32]. If a shard-local index replica crashes, it can be rebuilt by replaying the log from the latest snapshot. If a replica becomes corrupted, it can be discarded and rebuilt without risking logical data loss.

Candidate generator state such as graph adjacency lists is not necessarily trivially reconstructible from vectors alone, because graph construction depends on insertion order and heuristic search [33]. The design therefore treats graph edges as a materialization that can be recovered approximately. Recovery can use one of two approaches depending on the required service level. In a fast recovery mode, the replica restores a minimal navigable structure using a subset of edges stored in periodic edge snapshots, then incrementally repairs the graph by re-inserting items from the log tail [34]. In a slower, higher-fidelity mode, the replica rebuilds the graph from vectors using a deterministic construction procedure parameterized by a seeded random generator, producing stable results across rebuilds. The deterministic option reduces divergence across replicas but can be more expensive. The architecture does not assume that graph-based indices are the only option; inverted-list indices are easier to rebuild because they depend primarily on centroid assignments and codebooks, which can be snapshot and reused [35].

A fault-tolerant query path must handle partial unavailability and stragglers. The system uses a quorum-based candidate acquisition policy. For each logical shard s, there is a set of replicas $\mathcal{R}_s$. The coordinator selects a subset $\mathcal{R}_s(q)$ to query and waits for a read quorum or until a deadline. Candidate acquisition for shard s is therefore a function of both failures and deadlines [36]. Let $A_{s,r}(q)$ be the candidate set returned by replica r for shard s. The coordinator forms

$$C_s(q) = \bigcup_{r \in \mathcal{R}_s(q)} A_{s,r}(q)$$

subject to r responds before deadline.

If no replica responds before the deadline, the shard is treated as missing [37]. Missing shards can be compensated by expanding the routing set $\mathcal{S}(q)$ to include additional shards, increasing candidate diversity. This introduces a controlled latency-recall trade-off: the router can increase the number of shards queried when it detects missing responses, thereby attempting to preserve recall while accepting higher compute and network cost. In heterogeneous environments, this expansion can favor shards whose replicas are on faster tiers, reducing the marginal latency increase.

To analyze recall loss under missing shards, consider the event that a true top-k neighbor resides in a shard whose candidate set is missing or too small [38]. Let $p_s(q)$ denote the probability that a true neighbor of q resides in shard s, conditional on routing. Let u_s denote the probability that shard s is unavailable within the deadline due to failures or stragglers. A simplified

Table 5. Failure scenarios for evaluating availability and resilience.

Scenario	Failure type	Affected component	Injection rate
S1	Crash-stop	Compute node	1 / 30 min
S2	Disk timeout	Storage tier T2	5% of I/Os
S3	Link degradation	ToR switch	1 link / rack
S4	Corrupted block	Data shard	0.01% of writes
S5	Power loss	Rack-level	1 event / run

Table 6. End-to-end performance under normal operation for different index and storage combinations.

Index	Storage tier	Throughput (kQPS)	99th-latency (ms)
Local-LSH	T1	120	9.8
Sharded-IVF	T2	210	7.2
Replicated-PQ	T1	160	6.5
Replicated-PQ	T2	135	8.1
FTDIndex	T1	245	5.4
FTDIndex	T2	230	5.9

upper bound on the probability that a specific true neighbor is missed due to shard unavailability can be expressed as [39]

$$P_{\mathrm{miss}}(q) \;\le\; \sum_{s \in \mathcal{S}(q)} p_s(q)\, u_s \;+\; \sum_{s \in \mathcal{S}(q)} p_s(q)\,(1-u_s)\,\epsilon_s(q),$$

where $\epsilon_s(q)$ captures the probability that the shard returns a candidate set that does not include the neighbor even when available, due to approximation. This expression separates availability-driven loss from approximation-driven loss and highlights that fault tolerance can be achieved either by reducing u_s through replica diversity and fast failover, or by reducing sensitivity to missing shards through routing expansion and redundancy.

Replica diversity is emphasized because correlated failures within a fault domain can defeat naive replication [40]. If replicas of a shard reside in the same fault domain, a domain outage yields $u_s \approx 1$ during the outage window. The system therefore places replicas across fault domains when possible and uses heterogeneous replicas to reduce correlated performance degradation. For example, one replica may store candidate generator state in memory and another on NVMe [41]. A memory pressure event might slow the in-memory replica but leave the NVMe replica unaffected, and vice versa for NVMe contention. This form of heterogeneity can function as a resilience mechanism rather than merely a performance optimization.

Recovery protocols coordinate the transition of replicas from failed to healthy states [42]. When a replica restarts, it loads the latest snapshot of the vector store and replays the log tail. It then rebuilds or repairs its candidate generator and re-ranking materializations. During recovery, the replica can participate in reads in a degraded mode, such as serving candidates from an inverted-list index while its graph index is still repairing [43]. To prevent serving logically deleted items, the replica maintains a compact deletion filter derived from the committed log. Deletion filters can be represented as versioned bitmaps or Bloom-like structures; the key requirement is that they can be updated from the log and can be applied quickly during candidate emission.

5. Consistency, Correctness, and Performance Modeling

Because the retrieval problem is approximate and because shard-local indices are asynchronously maintained, correctness must be defined in a way that is meaningful for similarity search [44]. The paper adopts a notion of semantic correctness tied to the versioned vector store. An item is considered live at version v if the latest committed record for its identifier at or before v is not a deletion marker. A query executed at time t is associated with a read version $v(t)$, such as the latest committed version observed by the coordinator [45]. The system is semantically correct if it does not return items that are not live at $v(t)$ and if it returns results derived from vectors that correspond to committed versions at or before $v(t)$, subject to a bounded staleness policy that may allow returning slightly older versions when necessary.

Candidate generators may emit identifiers that correspond to older versions or that have been deleted, particularly when indices lag behind the log. The co-

Table 7. Impact of injected failures on throughput and recovery latency.

Index	Failure scenario	QPS degradation (%)	Recovery time (s)
Sharded-IVF	S1	42	18.3
Sharded-IVF	S4	15	6.7
Replicated-PQ	S1	21	4.1
FTDIndex	S1	9	2.8
FTDIndex	S5	13	7.5

Table 8. Load balance and network utilization for different placement strategies.

Configuration	Load imbalance (%)	Hotspot nodes	Network overhead (%)
Hash-based sharding	23	7	18
Range-based sharding	31	11	15
Topology-aware placement	12	3	13
FTDIndex placement	7	1	11

ordinator therefore performs a validation step by consulting a compact version map [46]. The version map maps identifiers to their latest committed version and live/dead status. The version map is derived from the vector log and can be stored in a replicated key-value service or embedded in shard-local replicas with periodic synchronization. The validation step filters out candidates that violate liveness and can optionally rescore candidates using the latest vector version if the re-ranking layer has access to it [47]. This design allows the system to decouple index maintenance staleness from semantic liveness, ensuring that deletions take effect promptly even if indices are stale.

A performance model is needed to reason about heterogeneity and fault-driven degradation. Let the end-to-end query latency be the sum of routing time, shard query times, network merge time, and re-ranking time [48]. The shard query time depends on the candidate generator type and the tier where its state resides. For shard s on replica r, let $T^{\mathrm{cand}}_{s,r}(q)$ denote candidate generation time and $T^{\mathrm{rank}}_{s,r}(q)$ denote time to fetch and score high-fidelity vectors for the subset of candidates. The coordinator selects replicas to minimize a tail-sensitive objective, which can be expressed as minimizing an upper quantile of the latency distribution. A simplified expected latency under a deadline D for each shard can be approximated by [49]

$$\begin{aligned}\mathbb{E}[T(q)] \;=\; & T^{\mathrm{route}}(q) \\ & + \mathbb{E}\Big[\max_{s\in\mathcal{S}(q)} \min_{r\in\mathcal{R}_s(q)} T^{\mathrm{cand}}_{s,r}(q)\Big] \\ & + \mathbb{E}\Big[T^{\mathrm{merge}}(q) + T^{\mathrm{rerank}}(q)\Big].\end{aligned}$$

The inner min reflects choosing the fastest responding replica among those queried, while the outer max reflects waiting for the slowest shard response among those needed to assemble the candidate pool. In practice, the coordinator may not wait for all shards and may proceed once enough candidates have been accumulated, which replaces the max with an order statistic [50]. The model highlights that heterogeneity can be exploited by querying multiple replicas per shard and using the first response, which reduces tail latency at the cost of extra compute and network.

The relationship between candidate set size and recall can be modeled using a probabilistic coverage argument. Let the true top-k set be $G_k(q)$ [51]. The system returns $R_k(q)$ based on candidates $C(q)$. Define recall as $|R_k(q) \cap G_k(q)|/k$. For a given shard s, let c_s be the number of candidates requested and let $\rho_s(c_s, \theta_s)$ be the probability that a true neighbor in shard s is included in $C_s(q)$, where θ_s captures index parameters and representation fidelity [52]. Under independence assumptions that may not hold exactly but can guide design, the probability that a true neighbor is included in the global candidate pool is

$$P_{\mathrm{inc}}(q) \;=\; 1 \;-\; \prod_{s\in\mathcal{S}(q)} \Big(1 - p_s(q)\,(1-u_s)\,\rho_s(c_s,\theta_s)\Big).$$

This expression suggests that increasing replica diversity reduces u_s, increasing candidate budget increases ρ_s, and improving routing increases $p_s(q)$ concentration on shards likely to contain near neighbors [53]. Under faults, the system can temporarily increase $|\mathcal{S}(q)|$ or c_s to compensate for increased u_s, but this must be balanced against latency constraints. The model also suggests that heterogeneous replicas can increase ρ_s without increasing c_s by using a higher-fidelity index on at least one replica, while a lower-fidelity replica provides a fallback when the higher-fidelity one is unavailable.

Consistency for updates is managed by ensuring that the committed log order per shard is well-defined.

Table 9. Index maintenance overhead before and after applying the proposed design.

Operation	Baseline cost (ms)	FTDIndex cost (ms)	Reduction (%)
Insert	4.1	2.3	44
Delete	3.7	2.0	46
Update	5.2	2.9	44
Rebalance	120	63	48
Checkpoint	340	176	48

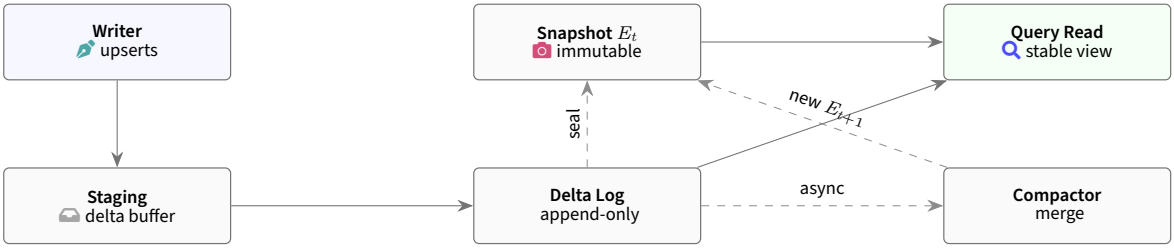


Figure 7. Epoch-based consistency for similarity search: writers append deltas while queries read a stable snapshot epoch; background compaction merges deltas into a new immutable epoch without blocking reads.

Let v index committed versions [54]. The index materialization at replica r for shard s is consistent up to a materialization version $m_{s,r} \leq v$. The system aims to maintain a bound Δ such that $v - m_{s,r} \leq \Delta$ for healthy replicas. This bound depends on ingestion rate, snapshot frequency, and replay throughput. Under failures, $m_{s,r}$ can fall behind. The coordinator can incorporate this information by preferring replicas with higher $m_{s,r}$ for queries that require freshness, or by selecting replicas with lower $m_{s,r}$ when freshness is less critical but latency is more important. This explicit tracking of materialization versions allows the system to make informed trade-offs rather than implicitly mixing stale and fresh state.

6. Heterogeneity-Aware Placement and Scheduling

Heterogeneity-aware placement concerns deciding which index components reside on which tiers and how replicas differ [55]. The system treats placement as an optimization problem with constraints on capacity, durability, and performance. For each shard, the versioned vector store must be durable, suggesting replication across fault domains or storage on a backend with sufficient durability. Candidate generator state can be placed on faster tiers for low latency or on slower tiers for lower cost, given that it is reconstructible [56]. Reranking state can be tiered: full vectors can be in memory for a hot subset of items, while compressed codes and residuals can be on NVMe, and cold vectors can be fetched from object storage if required by correctness or if the application tolerates latency spikes.

Placement decisions also interact with failure behavior. If a candidate generator is stored only in memory, a node restart causes its loss and triggers rebuild, which can temporarily increase u_s or reduce ρ_s as the replica serves in degraded mode [57]. If the candidate generator is persisted on NVMe, recovery can be faster, but NVMe contention can increase tail latency. The placement policy therefore includes rebuild cost and expected failure rate. Let B_t denote bandwidth of tier t for sequential reads, and let L_t denote latency [58]. Let S_s^{idx} denote the size of shard-local index materialization and S_s^{vec} the size of vectors. A coarse estimate of rebuild time from a snapshot plus log tail is

$$T_{s,r}^{\text{rebuild}} = \frac{S_s^{\text{vec}}}{B_{t(r)}^{\text{read}}} + \frac{S_s^{\text{idx}}}{B_{t(r)}^{\text{write}}} + T_{s,r}^{\text{construct}},$$

where $t(r)$ denotes the dominant tier for the rebuild and $T_{s,r}^{\text{construct}}$ captures compute-dependent construction time. On GPU-enabled replicas, $T_{s,r}^{\text{construct}}$ may be lower for vector quantization and batched scoring, while graph construction may still be bottlenecked by memory operations. On CPU-only replicas, construction may be slower but more stable under GPU scheduling contention.

Scheduling determines how queries are mapped to replicas given placement and current load [59]. A heterogeneity-aware scheduler uses per-replica performance models and online measurements. For each replica, it maintains estimates of candidate generation throughput, tail latency under load, and failure probability. The scheduler can issue hedged requests to multiple replicas when tail latency exceeds a threshold, canceling slower re-

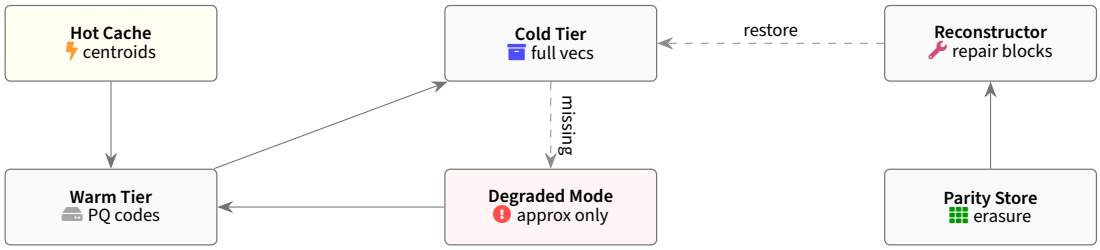


Figure 8. Tier-aware fault tolerance: hot caches and warm PQ codes keep search responsive; if cold full vectors are unavailable, the system serves approximate results until erasure-coded reconstruction restores the missing blocks.

quests when a faster response arrives [60]. Hedging can reduce tail latency but increases load, which can destabilize the system if applied indiscriminately. The scheduler therefore restricts hedging based on budget and on the marginal benefit predicted by the latency distribution.

A notable aspect of heterogeneous deployments is that different replicas can be specialized for different query types [61]. Queries with strict latency constraints may be routed to GPU-backed replicas that keep re-ranking vectors in memory, while queries that tolerate latency but require higher recall can be routed to replicas that query more shards or use higher candidate budgets. Metadata filters further complicate scheduling, because filtering can reduce effective candidate pool size and can cause more frequent re-ranking fetches when filter selectivity is low. The scheduler can incorporate predicate-aware statistics to decide whether to execute filtering early using per-shard posting lists over metadata, or late using validation after candidate generation [62]. Early filtering can reduce compute but may increase index complexity, while late filtering is simpler but may waste candidate budget.

The placement policy can be formalized as minimizing an objective that combines expected tail latency, expected recall loss under faults, and expected rebuild cost, subject to capacity constraints. Let $y_{s,r,t}$ indicate whether component r of shard s is placed on tier t. Let $\Lambda(q)$ denote the query distribution [63]. A schematic objective can be written as

$$\begin{aligned}\min_{\{y\}} \; & \mathbb{E}_{q\sim\Lambda}[\mathrm{TailLat}(q \mid \{y\})] \\ & + \lambda_1 \, \mathbb{E}_{q\sim\Lambda}[\mathrm{RecallLoss}(q \mid \{y\})] \\ & + \lambda_2 \, \mathbb{E}[\mathrm{RebuildCost}(\{y\})]\end{aligned}$$

$$\text{subject to} \sum_{s,r} y_{s,r,t} \, \mathrm{Size}_{s,r} \;\le\; \mathrm{Cap}_t \quad \forall t.$$

While solving this exactly is often impractical, the formulation clarifies the trade-offs [64]. In practice, the system can use heuristics such as placing at least one high-fidelity replica per shard on a fast tier, ensuring cross-domain diversity, and using lower-fidelity replicas on cheaper tiers as fallbacks. It can also adapt placements over time by migrating shards based on observed hotness and by adjusting snapshot frequency to control rebuild time.

7. Evaluation Methodology and Discussion

A rigorous evaluation of a fault-tolerant distributed similarity index must consider both steady-state performance and behavior under fault scenarios [65]. Steady-state metrics include recall at k, mean latency, and tail latency such as the 95% and 99% percentiles, measured under realistic concurrency. Because deployments are heterogeneous, evaluation should stratify results by replica type and tier, reporting how often each type is selected by the scheduler and how hedging affects load. Update workload metrics include ingest throughput, index materialization lag Δ, and the fraction of queries that observe stale materializations [66]. Correctness metrics include the rate at which deleted items appear in candidates and the effectiveness of validation in suppressing them.

Fault scenarios should include single-node crashes, correlated fault-domain outages, and tier-specific unavailability such as a degraded object storage backend or an NVMe device failure. For each scenario, evaluation should measure time to detect failure, time to fail over, recall degradation during the incident, and time to recover replicas to healthy state [67]. Recovery should be evaluated both for cold restarts, where memory-resident state is lost, and for warm restarts, where local persistent state remains. The system's reconstructibility claims should be validated by measuring rebuild time from snapshots and log tails across tiers and by assessing how degraded modes affect recall and latency.

The evaluation should also explore routing redundancy and routing expansion policies [68]. When shards are missing, routing expansion increases the number of shards queried, which can restore recall but can increase latency and compute. The evaluation should therefore characterize the trade-off curve between additional shard fanout and recall under varying failure rates. In heterogeneous settings, expansion should be compared under policies that prioritize fast-tier shards versus policies that prioritize shards with historically

higher neighbor probability for the query distribution [69]. Because query distributions can drift, the evaluation should include scenarios where the router's statistics are stale, examining how quickly the system adapts and whether misrouting amplifies fault-induced recall loss.

Another dimension concerns replica diversity. When replicas differ in index type and representation fidelity, the system can combine them to improve resilience [70]. Evaluation should compare symmetric replication, where all replicas are identical, against diverse replication, where at least one replica uses a higher-fidelity candidate generator and another uses a lower-fidelity but more durable representation. Measurements should include how often the system falls back to lower-fidelity replicas, the incremental cost of maintaining multiple representations, and the degree to which diverse replication reduces recall loss under correlated performance degradation. It is also important to measure the operational cost of diversity, such as increased complexity in rebuild pipelines and the need to validate that different indices remain consistent with the same versioned vector store [71].

Update-heavy regimes should be included because they stress materialization lag and recovery. A sustained update rate can increase the log tail and extend replay time after a crash. Evaluation should examine how snapshot frequency affects both steady-state overhead and recovery time [72]. More frequent snapshots reduce replay time but increase write amplification and background I/O. Less frequent snapshots reduce overhead but can increase recovery time and the window of degraded service. The analysis should also examine how asynchronous materialization interacts with scheduling [73]. If one replica lags significantly behind in materialization version, routing it for freshness-sensitive queries can reduce semantic correctness, while routing it for latency-sensitive queries can be acceptable if validation filters stale candidates.

Finally, evaluation should discuss how the system behaves under partial re-ranking materialization loss. If full vectors are missing for some candidates due to tier unavailability, the system may rely on compressed scoring [74]. Evaluation should characterize the resulting score distortion and its impact on top-k ordering. In settings where the application uses retrieval for downstream neural re-ranking, the effect of slight distortions in the first-stage retrieval may be mitigated by the second stage, whereas in purely retrieval-driven applications, top-k ordering may be more sensitive. The system should therefore expose degradation signals and allow policies that adjust candidate budgets or shard fanout when full re-ranking is unavailable [75].

8. Adaptive Control Under Faults and Heterogeneous Drift

Distributed similarity search over heterogeneous fleets is inherently non-stationary because the effective performance envelope varies with background load, device wear, cache residency, and transient network conditions, while the query distribution and embedding geometry can drift with model refreshes and content churn. A static configuration of shard fanout, per-shard candidate budget, replica selection, and re-ranking depth is therefore unlikely to remain near-optimal, and under faults it can produce unstable behavior such as latency spikes caused by hedging storms or recall collapses caused by overly aggressive timeouts. This section develops an adaptive control layer that treats the retrieval pipeline as a closed-loop system. The controller observes telemetry that includes per-replica latency distributions, timeout rates, candidate yield statistics, materialization lag, and the empirical overlap between candidate pools and final top-k outputs. It then adjusts decision variables that govern query execution, explicitly coupling these adjustments to failure signals and tier availability, while enforcing stability constraints that prevent oscillations.

Let a query q induce a control vector $a(q)$ consisting of shard fanout $F(q)$, per-shard candidate budgets $\{c_s(q)\}_{s\in\mathcal{S}(q)}$, a replica selection policy $\pi_s(q)$ that may include hedged requests, and a re-ranking depth $r(q)$ that determines how many candidates are scored with high-fidelity representations when available. The environment provides stochastic outcomes: a latency random variable $T(q,a)$, a recall random variable $\mathrm{Rec}(q,a)$, and a degradation descriptor $D(q,a)$ capturing which shards timed out, which tiers were unavailable, and which representations were used. The controller aims to maintain a recall target while minimizing a tail-sensitive latency objective. A convenient formulation uses a constrained stochastic optimization where the constraint is expressed as an expectation over the query distribution and the current system state. Denote the system state by z, which summarizes current load, estimated failure probabilities, and materialization versions, and denote the query distribution by Λ. The control objective can be written as

$$\min_{\mu} \ \mathbb{E}_{q\sim\Lambda}[\phi(T(q,a_\mu(q,z)))]$$

$$\text{subject to } \mathbb{E}_{q\sim\Lambda}[\mathrm{Rec}(q,a_\mu(q,z))] \ \geq \ \rho_0$$

$$\mathbb{E}_{q\sim\Lambda}[\psi(C(q,a_\mu(q,z)))] \ \leq \ B_0,$$

where μ parameterizes the control policy, ϕ is a tail-sensitive penalty such as a soft approximation to a high quantile, ψ penalizes total candidate volume or compute cost, and B_0 is a budget reflecting cluster capacity constraints. This formulation treats recall as a controllable service-level outcome rather than a fixed property of an index type, which is appropriate because recall

can be increased by querying more shards, requesting more candidates, or deepening re-ranking when the tier is available.

In practice, the system does not observe exact recall against ground truth at full scale, and it cannot measure $\mathrm{Rec}(q, a)$ directly for every query. Instead, it uses proxy signals that correlate with recall, such as candidate set stability, agreement between independent replicas, and consistency between coarse scores and final reranked scores. One operationally useful proxy is the marginal gain curve of candidate acquisition. For each shard response, the system can track how often additional candidates beyond a prefix contribute to the final top-k after merging. If the marginal contribution of the tail candidates is near zero for a period, budgets can be reduced with limited recall impact. Conversely, when faults cause missing shards or reduced fidelity, the marginal contribution of additional fanout or hedged replicas may rise, signaling a need to allocate more budget to restore recall.

A key challenge is the interaction between adaptive policies and correlated failures. If a tier becomes slow, a naive controller that increases hedging for all queries can amplify load on alternative replicas, pushing them into overload and causing a cascade of timeouts. The control layer therefore enforces rate limits and stability constraints. Let $h_s(t)$ denote the probability of issuing a hedged request for shard s at time t, and let $\ell_s(t)$ denote the observed tail latency estimate for shard s under the current policy. A stability-oriented update uses smoothed statistics and bounded step sizes:

$$\ell_s(t) \;=\; (1-\alpha)\,\ell_s(t-1) \;+\; \alpha\,\widehat{\ell}_s(t)$$

$$h_s(t) \;=\; \mathrm{clip}\big(h_s(t-1) \;+\; \eta\, g(\ell_s(t)-\tau),\; 0,\; h_{\max}\big)\,,$$

where $\widehat{\ell}_s(t)$ is an instantaneous tail estimate derived from recent samples, α controls smoothing, η is a conservative learning rate, τ is a latency threshold above which hedging becomes beneficial, g is a bounded response function, and clip enforces a maximum hedging probability $h_{\max}$ chosen to avoid saturating replicas. The same pattern applies to fanout and candidate budgets, ensuring that the controller reacts to persistent degradation rather than transient noise. Because heterogeneous replicas differ in cost, the controller further conditions updates on tier identity and current queue depth, so that it increases hedging preferentially toward replicas whose marginal cost remains acceptable.

To connect adaptive control to fault tolerance, the controller maintains online estimates of shard unavailability within deadlines. For each shard s, define an empirical unavailability rate $\widehat{u}_s$ over a sliding window, counting both explicit failures and deadline misses. The controller uses $\widehat{u}_s$ to decide whether to expand routing, increase replica diversity, or increase candidate budgets. When missing shards are detected for a query, the controller can apply an on-the-fly expansion rule that is safe under load. Let $M(q)$ be the set of shards that missed deadlines for query q. The controller selects an expansion set $\Delta\mathcal{S}(q)$ from a pool of backup shards suggested by the router's coarse quantizer neighborhood or by a learned retrieval-of-shards model. The expansion size is chosen to respect compute budgets and to be proportional to the estimated recall deficit. A conservative approximation uses a linearized miss probability model:

$$\widehat{\delta}(q) \;=\; \sum_{s\in M(q)} \widehat{p}_s(q)$$

$$|\Delta\mathcal{S}(q)| \;=\; \mathrm{clip}\Big(\Big\lceil \gamma\,\widehat{\delta}(q)\Big\rceil\,,\; 0,\; F_{\max} - F(q)\Big)\,,$$

where $\widehat{p}_s(q)$ is an estimated probability mass that a near neighbor lies in shard s under current routing, γ is a gain parameter, and $F_{\max}$ is a hard cap on fanout to protect the cluster. This rule increases fanout only when missed shards correspond to shards that the router believes are relevant for the query, which reduces waste compared to blind expansion.

Candidate budget control similarly benefits from modeling diminishing returns and tier-specific cost. For a shard s on replica r, let $c_{s,r}$ be the candidate budget and let $\kappa_{s,r}$ be a cost coefficient reflecting compute and I/O per candidate on that replica, including the effect of cache misses and remote fetch amplification. The controller can solve a per-query allocation problem that chooses budgets to maximize expected candidate coverage under a total cost cap. If $\widehat{\rho}_{s,r}(c)$ denotes an empirical inclusion probability proxy that increases concavely with c, a per-query approximation is

$$\max_{\{c_{s,r}\}} \sum_{s\in\mathcal{S}(q)} \max_{r\in\pi_s(q)} \widehat{\rho}_{s,r}(c_{s,r})$$

$$\text{subject to} \sum_{s\in\mathcal{S}(q)} \sum_{r\in\pi_s(q)} \kappa_{s,r}\, c_{s,r} \;\le\; K(q),$$

where $K(q)$ is a query-specific compute budget that may depend on user tier or latency class. Although exact solutions require more structure on $\widehat{\rho}_{s,r}$, the key operational insight is that heterogeneous replicas induce heterogeneous marginal gains. A GPU-backed replica with an in-memory graph may have low $\kappa_{s,r}$ and high marginal gain at small c, whereas a CPU-only replica that must fetch codes from NVMe may have higher $\kappa_{s,r}$ and may be best used as a fallback with smaller budgets unless faults force reliance on it. The controller can approximate this allocation by greedily increasing $c_{s,r}$ where the ratio of marginal gain to marginal cost is highest, using a small number of increments to keep overhead bounded.

Re-ranking depth control must account for tier availability and failure-induced shifts in candidate quality. When high-fidelity vectors are unavailable due to tier

outage, scoring must rely on compressed representations, which can increase score noise and destabilize top-k ordering. The controller therefore adapts $r(q)$ to compensate by increasing the number of candidates passed to reranking when using lower-fidelity scoring, subject to latency constraints. Let σ_t denote an estimated score noise level for tier t and representation type, and let r_t be the reranking depth when that representation is active. A simple monotone rule is to increase r_t with σ_t until the marginal latency cost becomes prohibitive. This connects to the concept of robustness margins: when scoring is noisier, a larger candidate set reduces the risk that true neighbors are displaced by noise before the final selection.

The control layer must also respect freshness and semantic constraints driven by the versioned vector store. When materialization lag increases, candidate generators may emit stale identifiers whose latest versions differ materially in score, or they may emit identifiers that have been deleted. Validation suppresses semantically invalid candidates, but excessive suppression reduces effective candidate pool size and can reduce recall. The controller therefore uses lag-aware scheduling, preferring replicas with smaller lag for workloads that are sensitive to freshness, and compensating for replicas with larger lag by requesting more candidates or by consulting a small incremental index over the log tail. Because the incremental index is typically in memory and shared, it can act as a stabilizer during recovery when multiple replicas are rebuilding. The controller can explicitly allocate a bounded portion of the per-query budget to the incremental index when it detects elevated lag or increased deletion-filter hits.

An important operational consideration is multi-tenant fairness. Adaptive policies that increase fanout or hedging under faults can consume shared capacity, harming other tenants and potentially causing global instability. The controller therefore operates under tenant-level budgets, enforcing that each tenant's expected compute and I/O consumption stays within an allocation. This is naturally incorporated through the budget constraint B_0 and through per-tenant $K(q)$, but practical enforcement requires a runtime mechanism that meters work in units that reflect heterogeneous cost. A cost unit can be defined as an affine combination of vector operations, memory bandwidth, and remote reads, calibrated from online profiling. The controller then throttles expansions and hedging when a tenant approaches its budget, preferring to degrade recall gracefully rather than triggering widespread timeouts.

9. Conclusion

Fault-tolerant similarity search in heterogeneous environments benefits from treating indexing state as layered, with small strongly consistent routing metadata and large reconstructible shard-local materializations derived from a durable versioned vector store. By separating semantic item state from the operational details of candidate generator construction, the system can rebuild or replace index structures without risking logical data loss, which is useful under crashes, corruption, and evolving hardware configurations. Quorum-based candidate acquisition and routing expansion provide mechanisms for continuing query service under partial failures, while explicit validation against version maps bounds the risk of returning deleted or semantically invalid items even when indices lag behind updates [76].

The paper's models connect failure probabilities, replica diversity, and candidate generation fidelity to expected recall loss and latency behavior, emphasizing that resilience can be achieved not only by increasing replication but also by diversifying replica implementations and placements across tiers and fault domains. Heterogeneity-aware placement and scheduling further allow the system to exploit fast compute and storage where available while maintaining fallbacks on cheaper or more durable tiers, with rebuild cost and tail latency treated as first-class considerations. While the resulting architecture increases system complexity compared to homogeneous, single-index designs, it provides a structured way to reason about trade-offs among durability, latency, recall, and recovery, and it supports predictable degradation rather than abrupt failure when parts of the environment become unavailable [77].

References

[1] C. Y. Guo and H. B. Wu, "Capacity optimization of energy storage unit in distributed generation system," Advanced Materials Research, vol. 608-609, pp. 1116–1119, 12 2012.

[2] J. Y. Hui and J. Zhang, "Grams: a distributed multimedia service system," in SPIE Proceedings, vol. 2188, pp. 189–199, SPIE, 4 1994.

[3] R. R. Bate, "Distributed micfoprccessors in avionics systems," in 2nd Computers in Aerospace Conference, American Institute of Aeronautics and Astronautics, 8 1979.

[4] Y. Latreche, H. Bouchekara, F. Kerrour, K. Naidu, H. Mokhlis, and M. S. Javaid, "Comprehensive review on the optimal integration of distributed generation in distribution systems," Journal of Renewable and Sustainable Energy, vol. 10, pp. 055303–, 9 2018.

[5] H. Chang, Y. S. Lin, and A. Priver, "Formulation of dynamic switching in distributed ferromagnetic-film systems," Journal of Applied Physics, vol. 38, pp. 2294–2301, 4 1967.

[6] W. Yang, Y. Cai, and L. Wang, "Distributed space-time-frequency coding for cooperative ofdm systems," Science in China Series F: Information Sciences, vol. 52, pp. 2424–2432, 12 2009.

[7] Y. Chen, A. Pan, H. Lei, A. Ye, S. Han, Y. Tang, W. Lu, Y. Chai, F. Zhang, and X. Du, "Tdsql: Tencent distributed database system," Proceedings of the VLDB Endowment, vol. 17, pp. 3869–3882, 11 2024.

[8] A. W. Rodriguez and J. A. Reggia, "A distributed learning algorithm for particle systems," Integrated Computer-Aided Engineering, vol. 16, pp. 1–20, 1 2009.

[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 dimen-

sions," in International Conference on Scientific and Statistical Database Management, pp. 167–184, Springer, 2009.

[10] R. Luck and A. Ray, "An observer-based compensator for distributed delays in integrated control systems," in Guidance, Navigation and Control Conference, American Institute of Aeronautics and Astronautics, 8 1989.

[11] D. K. Tosh, S. Shetty, X. Liang, L. Njilla, C. A. Kamhoua, and K. Kwiat, Blockchain for Distributed Systems Security - Blockcloud Security Analysis. Wiley, 3 2019.

[12] N. Tampouratzis, I. Papaefstathiou, A. Nikitakis, A. Brokalakis, S. Andrianakis, A. Dollas, M. Marcon, and E. Plebani, "A novel, highly integrated simulator for parallel and distributed systems," ACM Transactions on Architecture and Code Optimization, vol. 17, pp. 1–28, 3 2020.

[13] L. A. Montestruque and M. D. Lemmon, "Globally coordinated distributed storm water management system," in Proceedings of the 1st ACM International Workshop on Cyber-Physical Systems for Smart Water Networks, pp. 10–6, ACM, 4 2015.

[14] R. J. Stroud, "Acm sigops european workshop - autonomy or interdependence in distributed systems," in Proceedings of the 3rd workshop on ACM SIGOPS European workshop: Autonomy or interdependence in distributed systems?, pp. 1–4, ACM, 9 1988.

[15] J. Luo and Q. Wu, "Scan statistics with local vote for target detection in distributed system," EURASIP Journal on Advances in Signal Processing, vol. 2017, pp. 32–, 5 2017.

[16] K. cai Cao, Y. Chai, and C. Liu, "Distributed consensus of multi-agent systems with increased convergence rate," International Journal of Nonlinear Sciences and Numerical Simulation, vol. 22, pp. 419–435, 11 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. Lee and A. S. Kiremidjian, "Uncertainty and correlation for loss assessment of spatially distributed systems," Earthquake Spectra, vol. 23, pp. 753–770, 11 2007.

[19] K. D. Frampton, "Distributed group-based vibration control with a networked embedded system," Smart Materials and Structures, vol. 14, pp. 307–314, 1 2005.

[20] M. Burgin and B. Gupta, "Second-level algorithms, superrecursivity, and recovery problem in distributed systems," Theory of Computing Systems, vol. 50, pp. 694–705, 11 2011.

[21] N. Li, H. Wang, Q. Luo, and W. Zheng, "Distributed formation control for multiple quadrotor uavs based on distributed estimator and singular perturbation system," International Journal of Control, Automation and Systems, vol. 22, pp. 1349–1359, 2 2024.

[22] N. Evangelou-Oost, C. Bannister, and I. J. Hayes, Contextuality in Distributed Systems, pp. 52–68. Germany: Springer International Publishing, 3 2023.

[23] A. Squicciarini, G. Petracca, and E. Bertino, "Codaspy - adaptive data protection in distributed systems," in Proceedings of the third ACM conference on Data and application security and privacy, pp. 365–376, ACM, 2 2013.

[24] J. Chen, H. Zhuang, and M. Yang, A Distributed Vehicle-Infrastructure Cooperation System Based on OpenHarmony, pp. 258–271. Germany: Springer Nature Singapore, 2 2024.

[25] Z. S. Bischof, J. S. Otto, and F. E. Bustamante, "Distributed systems and natural disasters: Bittorrent as a global witness," in Proceedings of the Special Workshop on Internet and Disasters, pp. 4–8, ACM, 12 2011.

[26] C. Ramachandran, S. Misra, and M. Obaidat, "On evaluating some agent-based intrusion detection schemes in mobile ad-hoc networks," Proceedings of the SPECTS 2007, pp. 594–601, 2007.

[27] C. Vogt, L. Wolf, R. G. Herrtwich, and H. Wittig, "Heirat—quality-of-service management for distributed multimedia systems," Multimedia Systems, vol. 6, pp. 152–166, 5 1998.

[28] Y. Zhao, W. Jiang, and L. Ni, "The application of solar energy in distributed energy systems," in 2nd International Conference on Laser, Optics and Optoelectronic Technology (LOPET 2022), pp. 62–62, SPIE, 10 2022.

[29] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," ACM SIGOPS Operating Systems Review, vol. 25, pp. 198–212, 10 1991.

[30] Z. Qiu, L. Zhang, Y. Tian, X. K. Feng, and S. Y. Zhang, "An optimized design of humanoid robot distributed control system," Applied Mechanics and Materials, vol. 541-542, pp. 1043–1048, 3 2014.

[31] A. Tharumarajah and A. J. Wells, "A behavior-based approach to scheduling in distributed manufacturing systems," Integrated Computer-Aided Engineering, vol. 4, pp. 235–249, 10 1997.

[32] G. Chen, Z. Zhang, C. Li, D. Qiao, and B. Sun, "Robust stabilization of stochastic markovian jump systems with distributed delays," Complexity, vol. 2021, pp. 1–8, 2 2021.

[33] H. Lee, R. Dathathri, and K. Pingali, "Kimbap: A node-property map system for distributed graph analytics," in Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp. 566–581, ACM, 4 2024.

[34] L. Yang, Y. Zhou, Y. Liu, and Y. Zhao, "Distributed prescribed-time containment tracking for heterogeneous linear systems," in 2023 38th Youth Academic Annual Conference of Chinese Association of Automation (YAC), pp. 463–468, IEEE, 8 2023.

[35] S. Distefano and A. Puliafito, Information Security in Large Scale Distributed Systems, pp. 485–500. Springer New York, 11 2011.

[36] K. M. Goeschka, S. Hallsteinsen, R. Oliveira, and A. Romanovsky, "Session details: Dependable and adaptive distributed systems track," in Proceedings of the 27th Annual ACM Symposium on Applied Computing, ACM, 3 2012.

[37] Z. Shi, Y. Li, Y. Lou, R. Yang, X. Yu, Y. Lu, L. Yang, S. Gu, B. Huang, Y. Luo, and S. Yan, "Design of a 600-kw distributed photovoltaic system," in Second International Conference on Energy, Power, and Electrical Technology (ICEPET 2023), pp. 212–212, SPIE, 9 2023.

[38] R. Liao, L. Han, F. Liu, X. Dong, Q. Li, and Z. Ren, "Distributed shortest-distance formation for multi-agent systems," in 2020 Chinese Automation Congress (CAC), pp. 4662–4667, IEEE, 11 2020.

[39] R. Sobie, F. Berghaus, K. Casteels, C. Driemel, M. Ebert, F. F. Galindo, C. Leavett-Brown, D. M. Macdonell, M. Paterson, R. Seuster, S. Tolkamp, and J. Weldon, "Cloudscheduler: a vm provisioning system for a distributed compute cloud," EPJ Web of Conferences, vol. 245, pp. 07031–, 11 2020.

[40] R. Heersmink, "Distributed cognition and distributed morality: Agency, artifacts and systems.," Science and engineering ethics, vol. 23, pp. 431–448, 7 2016.

[41] R. van Engelen, N. Koziris, M. Furnari, and K. Psarris, "Session details: Parallel and distributed systems (pds)," in Proceedings of the 2004 ACM symposium on Applied computing, ACM, 3 2004.

[42] 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.

[43] N. MacDonald, " -distributed delay in a model of a nonlinear hybrid optical system," Physical review. A, Atomic, molecular, and optical physics, vol. 41, pp. 4077–4078, 4 1990.

[44] A.-M. Gnacek, Y. An, and J. Ryan, "A data analysis expert system for large established distributed databases," in SPIE Proceedings, vol. 0786, pp. 132–137, SPIE, 5 1987.

[45] S. B. Sarsilmaz and T. Yucelen, "A distributed control approach for heterogeneous linear multiagent systems," International Journal of Control, vol. 94, pp. 1402–1414, 8 2019.

[46] I. C. Jong, "On stability of a circulatory system with weak distributed yielding effects," Journal of Applied Mechanics, vol. 37, pp. 870–873, 9 1970.

[47] P. Lyngbaek and D. McLeod, "Object management in distributed information systems," ACM Transactions on Information Systems, vol. 2, pp. 96–122, 5 1984.

[48] W. Lohmiller and J.-J. E. Slotine, "Contraction analysis of nonlinear distributed systems," International Journal of Control, vol. 78, pp. 678–688, 6 2005.

[49] F. L. Iudice, E. Dudkina, M. Coraggio, M. D. Bernardo, and E. Crisostomi, "A distributed strategy for parallel power system restoration," in 2023 IEEE Belgrade PowerTech, pp. 1–6, IEEE, 6 2023.

[50] S. Xu, R. Wang, B. Chen, and X. Qiu, "Distributed fusion estimation for nonlinear binary sensor systems," in 2024 43rd Chinese Control Conference (CCC), pp. 5255–5260, IEEE, 7 2024.

[51] W. Jiantao, Z. Changfu, H. Zheng, L. Chen, and C. Cheng, "Research on the dynamic integration control for distributed-traction electric vehicle with four-wheel-distributed steering system," in SAE Technical Paper Series, vol. 1, (United States), SAE International, 4 2018.

[52] D. Dutta, "Real time optimal control of distributed-parameter systems," 5 2023.

[53] Z. Chen, P. Liu, Y. Chen, W. Zhan, G. Interdonato, and S. Buzzi, "Distributed learning-based beamforming codebooks for unevenly distributed users in mmwave massive mimo system," in 2024 IEEE Wireless Communications and Networking Conference (WCNC), pp. 1–6, IEEE, 4 2024.

[54] B. Kouvaritakis, M. Cannon, B. Rohal'-Ilkiv, A. Karas, and C. Belavý, "Constrained predictive control of linear distributed parameter systems," International Journal of Control, vol. 77, pp. 941–948, 7 2004.

[55] H. F. Rashvand, K. Salah, J. M. A. Calero, and L. Harn, "Distributed security for multi-agent systems - review and applications," IET Information Security, vol. 4, pp. 188–201, 12 2010.

[56] S. Daniel, "Modified bully election algorithm for crash recovery in distributed systems," International Journal of Science and Research (IJSR), vol. 6, pp. 958–961, 12 2017.

[57] I. Georgiadis, J. Magee, and J. Kramer, "Woss - self-organising software architectures for distributed systems," in Proceedings of the first workshop on Self-healing systems, pp. 33–38, ACM, 11 2002.

[58] 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.

[59] P. Heredia, E. Garcia, and S. Mou, "Distributed state estimation for nonlinear systems with unknown parameters," in 2022 American Control Conference (ACC), pp. 96–101, IEEE, 6 2022.

[60] Y. Liu, J. Liu, Z. He, Z. Li, Q. Zhang, and Z. Ding, "A survey of multi-agent systems on distributed formation control," Unmanned Systems, vol. 12, pp. 913–926, 5 2023.

[61] J. Patel, "Effects of software aging and rejuvenation on performability of layered distributed systems," 5 2021.

[62] J. Kaiser, G. Hawkridge, G. Yilmaz, and D. McFarlane, "A low-cost integration approach for distributed manufacturing information systems," in 2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), pp. 763–769, IEEE, 12 2022.

[63] R. D. Nicola, D. Latella, M. Loreti, and M. Massink, Quantitative Assessments of Distributed Systems: Methodologies and Techniques - Quantitative Analysis of Distributed Systems in Stoklaim: A Tutorial. Wiley, 4 2015.

[64] D. Wang, W. Dou, Y. Gao, C. Wu, J. Wei, and T. Huang, "Model checking guided testing for distributed systems," in Proceedings of the Eighteenth European Conference on Computer Systems, pp. 127–143, ACM, 5 2023.

[65] K. Sun, X. Xiao, S. Wu, and L. Chen, "Fault diagnosis and state evaluation of distributed photovoltaic systems in microgrids," in 2023 11th International Conference on Power Electronics and ECCE Asia (ICPE 2023 - ECCE Asia), vol. 9, pp. 440–445, IEEE, 5 2023.

[66] E. Börger and A. Raschke, Modeling Distributed Systems, pp. 207–239. Springer Berlin Heidelberg, 4 2018.

[67] H. Li, M. Zhang, L. Zhang, L. Ma, and X. Zhang, "A reconfiguration strategy for high-power distributed spacecraft power system," in 2021 IEEE 1st International Power Electronics and Application Symposium (PEAS), pp. 1–6, IEEE, 11 2021.

[68] F. Guo, "Distributed control of energy systems," 1 2016.

[69] B. Ning, Q.-L. Han, and Z. Zuo, "Distributed optimization of multiagent systems with preserved network connectivity," IEEE transactions on cybernetics, vol. 49, pp. 3980–3990, 8 2018.

[70] Q. Liu, Z. Wang, and X. He, Event-Based Distributed Filtering of Continuous-Time Nonlinear Systems, pp. 77–98. Springer International Publishing, 10 2018.

[71] B. Luo, Z. Luo, and D. Guo, "Ant colony optimization strategy in distributed generation systems," in 2024 International Conference on Telecommunications and Power Electronics (TELEPE), vol. 43, pp. 762–767, IEEE, 5 2024.

[72] C. Li, Research on Distributed Synchronous CAD Collaborative Design System, pp. 931–939. Germany: Springer Netherlands, 12 2013.

[73] H. Chen, J. Yu, F. Liu, Z. Cai, and J. Xia, "Archipelago: A medical distributed storage system for interconnected health," IEEE Internet Computing, vol. 24, pp. 28–38, 3 2020.

[74] 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.

[75] K.-Y. Lam and S.-L. Hung, "Concurrency control for time-constrained transactions in distributed databases systems," The Computer Journal, vol. 38, pp. 704–716, 9 1995.

[76] H. L. Alexander, "State estimation for distributed systems with sensing delay," in SPIE Proceedings, vol. 1470, pp. 103–111, SPIE, 8 1991.

[77] J. W. Y. Lam, H. Gao, and C. Wang, "H∞ model reduction of linear systems with distributed delay," IEE Proceedings - Control Theory and Applications, vol. 152, pp. 662–674, 11 2005.