Research

Query-Aware Caching for Multi-Tenant Vector Search with SLA-Driven Eviction and Fairness Guarantees

Quang Nguyen[1] **and Khai Vo**[2]

[1] *Department of Computer Science and Engineering, Mekong Institute of Technology, Đường Hoa Phượng 12, Cần Thơ, Vietnam*
[2] *Department of Computer Science and Engineering, Red River University of Computing, Đường Lê Quý Đôn 88, Hà Nội, Vietnam*

Abstract

Multi-tenant vector search systems increasingly serve heterogeneous applications such as retrieval-augmented generation, recommendation, and deduplication, where each tenant issues embedding-based similarity queries over shared infrastructure. These workloads exhibit burstiness, temporal locality in query intent, and heavy-tailed latency sensitivity, while operators must satisfy per-tenant service-level agreements under tight memory and compute budgets. Caching is a natural lever, yet conventional policies optimize global hit rate or average latency and often ignore how approximate nearest-neighbor execution paths depend on the query embedding, index structure, and tenant-specific objectives. This paper studies query-aware caching for multi-tenant vector search with eviction driven by explicit SLA risk and fairness guarantees. We formalize cacheable objects beyond raw results, including centroid routes, candidate lists, graph neighborhoods, and quantization side data that reduce compute along typical ANN pipelines. We introduce a utility model that predicts marginal reductions in tail latency and SLA violation probability as a function of query similarity, routing state, and resource contention. Eviction is posed as an online constrained optimization problem balancing memory, energy, and tail latency, while enforcing tenant fairness via proportional or max-min style constraints. We develop practical algorithms that combine sketch-based query clustering, low-rank tenant intent models, and primal-dual updates that track shadow prices for fairness and memory. We also discuss performance engineering and distributed execution considerations, and outline an evaluation methodology emphasizing tail metrics, reproducibility, and robustness under tenant churn and workload shifts.

1. Introduction

Vector search has become a central primitive in modern information systems, where inputs and items are mapped to points in a high-dimensional space and similarity search returns nearest neighbors under metrics such as inner product or Euclidean distance [1]. In multi-tenant deployments, a single service hosts many independent tenants whose data, query distributions, and service-level agreements differ. The infrastructure typically shares compute, memory, network, and storage, and the system must simultaneously deliver predictable tail latency and acceptable recall for approximate nearest neighbor search. These constraints are aggravated by the operational reality that memory is scarce and expensive, and that the memory footprint of high-quality ANN indices can rival or exceed the underlying vectors due to graph links, inverted lists, and quantization metadata. Consequently, caching is pervasive, but the objects cached and the eviction policy used have outsized impact on tail latency, interference, and fairness.

The caching problem in vector search differs from conventional key-value caching in several ways. First, the access path is not a direct key lookup but an execution plan: a query embedding induces routing to coarse partitions, exploration of a search graph, and refinement under quantization or re-ranking. Second, the same tenant query can take different paths as the index evolves and as the system adapts parameters like efSearch or nprobe to meet latency. Third, the most valuable cached state may not be final results but intermediate artifacts that reduce compute, such as precomputed centroid-to-list mappings, partial graph neighborhoods, quantizer tables, or per-tenant feature projections [2]. Fourth, multi-tenancy introduces not only differing popularity patterns but also explicit per-tenant SLAs and fairness requirements: a cache policy that maximizes global throughput can systematically starve smaller tenants or degrade their tail latency under contention.

A query-aware cache must therefore reason about the geometry of embeddings, the structure of the ANN index, and the probabilistic relationship between cached artifacts and the execution cost distribution. It must also embed SLA semantics in eviction decisions. SLAs in this context often specify tail-latency thresholds, such as a p99 response time under a limit, and may include throughput guarantees, error-rate bounds, or energy budgets. Meeting these SLAs is complicated by queueing effects and shared resources, where a small increase in average compute can produce large tail increases under high utilization. A cache that reduces compute in common execution paths can lower utilization and thus improve tail behavior. However, if the cache is monopolized by one tenant with high query volume, other tenants may see disproportionate tail regressions even if their absolute load is modest. Fairness constraints aim to prevent such outcomes by enforcing bounds on relative SLA violation rates, allocation shares, or utility [3].

This paper develops a technical framework for query-aware caching in multi-tenant vector search with SLA-driven eviction and fairness guarantees. The focus is not limited to a particular ANN algorithm, but the discussion is grounded in typical pipelines, including inverted-file routing with product quantization and graph-based search. The key idea is to cache artifacts whose reuse probability can be

Dataset	#Vectors	Dimensionality	#Tenants
News-1B	1,000,000,000	768	32
Ecom-500M	500,000,000	512	24
Logs-200M	200,000,000	256	16
Media-100M	100,000,000	384	12
Emb-50M-Synth	50,000,000	128	08

Table 1. Workload characteristics used in the evaluation.

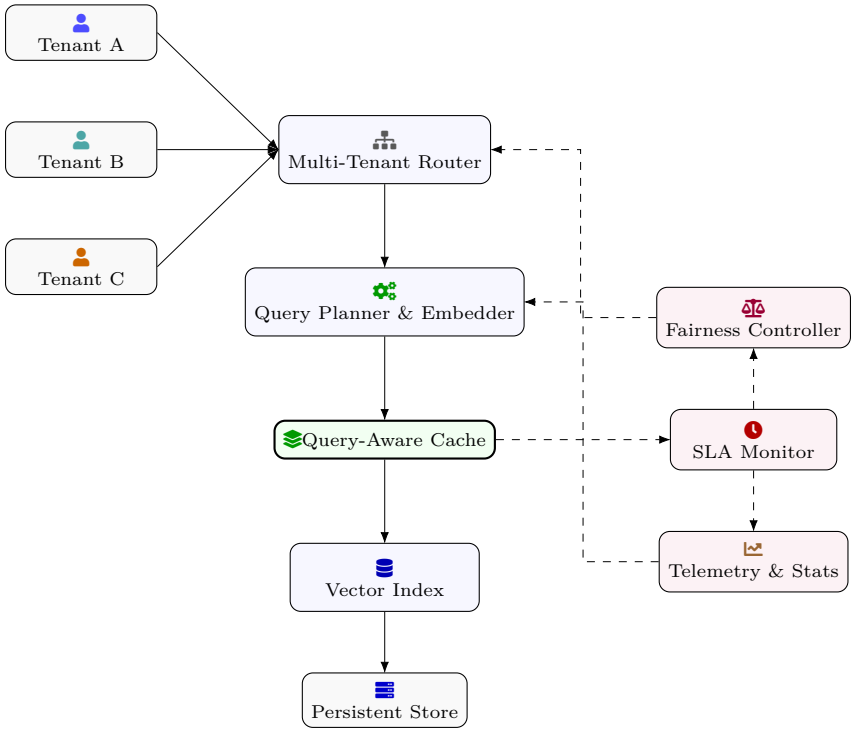

Figure 1. System architecture for query-aware caching in a multi-tenant vector search service. Tenant queries are routed through a shared planner and embedding layer before accessing a common cache and underlying vector index. SLA monitoring, fairness control, and telemetry form a feedback loop that tunes routing, cache behavior, and resource allocation across tenants.

predicted from query similarity, and whose marginal benefit can be estimated in terms of tail latency and SLA risk. The system maintains sketches of recent query embeddings per tenant, learns low-dimensional representations of tenant intent, and uses these to estimate reuse and benefit. Eviction is framed as an online constrained optimization problem where the objective is to minimize aggregate SLA violation risk while respecting memory and energy budgets and enforcing fairness constraints across tenants. A family of algorithms is derived using primal-dual updates and gradient-based methods, producing a policy that is implementable at high throughput and robust to workload drift.

Several technical themes recur throughout the paper. The first is representation: embeddings live in high dimension, but cache control must be fast [4]. We use random projections, principal components, and low-rank approximations to compress query history and to make similarity estimation cheap. The second is predictive modeling: utility estimates must incorporate approximate-search error and variable compute. We define utility in terms of predicted service time distributions and show how to propagate approximation error and sketch variance into conservative eviction decisions. The third is optimization under constraints: eviction and admission are posed as a constrained selection problem that is NP-hard in general, motivating approximations and online heuristics with principled structure. The fourth is systems engineering: distributed caches, NUMA locality, and memory tiering change the effective cost model, and the algorithms must be engineered to avoid adding overhead that negates their gains. The final theme is evaluation: tail behavior, fairness metrics, and reproducibility are treated as first-class outcomes, and we specify how to measure them under realistic multi-tenant load.

2. System Model and Problem Formulation

Consider a vector search service hosting a set of tenants indexed by $t \in \{1, \dots, T\}$. Tenant t has a dataset of item embeddings $\{y_{t,j}\}_{j=1}^{N_t}$ in $\mathbb{R}^d$ and a query stream producing embeddings $x \in \mathbb{R}^d$. The service answers top-k nearest neighbor queries under a similarity metric, commonly inner product $\langle x, y \rangle$ or Euclidean distance $\|x-y\|_2$. Let Q_t denote the random variable of queries from tenant t, with distribu-

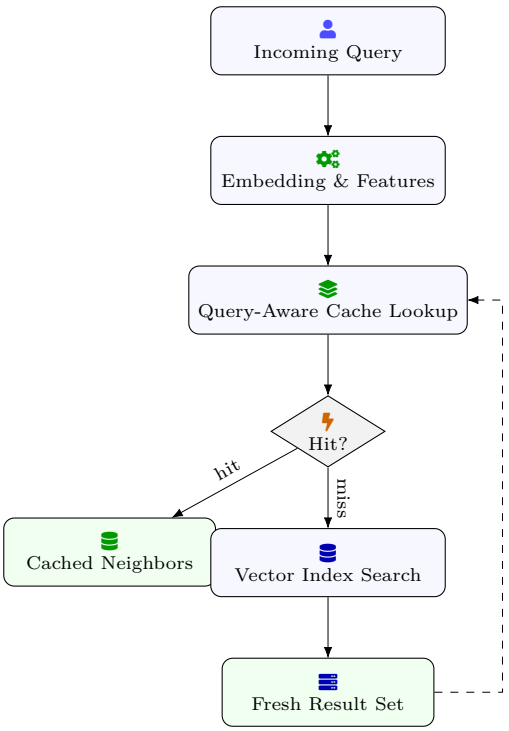

Figure 2. Per-query execution path through the cache-aware vector search pipeline. After feature extraction, the system probes a query-aware cache that captures reusable nearest neighbors. Cache hits bypass the vector index, while misses trigger a full search whose results may be selectively admitted into the cache based on query features and downstream policies.

SLA tier	Target P99 latency (ms)	Min hit rate (%)	Priority weight
Platinum	30	95	1.00
Gold	60	90	0.75
Silver	100	80	0.50
Bronze	150	70	0.25
Best-effort	250	50	0.10

Table 2. Tenant SLA tiers and associated service objectives.

tion capturing both embedding geometry and arrival time [5]. The service runs an ANN index per tenant, per shard, or in a shared structure with tenant filters. Regardless of physical organization, a query triggers an execution plan that includes routing, candidate generation, and optional refinement. The end-to-end latency is a random variable influenced by cache state, CPU/GPU availability, memory bandwidth, and queueing.

We model the system as a set of worker nodes with capacity in compute and memory. Let there be M nodes, each with memory budget B_m and compute capacity expressed as k_m parallel servers. Under heavy load, queueing dominates tail latency. A useful approximation is to treat each node as an $M/G/k$ queue with arrival rate λ_m and service time distribution S_m induced by query execution. While exact tail analysis is complex, the qualitative dependence is clear: reducing the mean and variance of service time lowers the probability of long waits [6]. A cache that decreases per-query compute reduces both mean and variance by avoiding costly index traversals and memory fetches, and thus improves tail.

Cacheable objects in vector search are not limited to final results. Let $\mathcal{O}$ denote the universe of cacheable objects. An object $o \in \mathcal{O}$ has a size $s(o)$ in bytes and a placement location (node-local, shard-local, or global). The object also has a reuse function: given a query embedding x and tenant t, the probability that o will be accessed during the query execution is $p(o \mid x, t)$, and the resulting cost reduction is $\Delta c(o \mid x, t)$, measured in expected service time reduction or in reduced memory stalls. Examples include coarse centroids for inverted-file indices, posting lists for frequently probed clusters, quantizer codebooks, precomputed residual norms, HNSW neighborhood subgraphs around common entry points, or learned routing logits for hierarchical partitioners.

To capture query-awareness, we associate each object o with an embedding signature $z(o) \in \mathbb{R}^{d'}$ that represents the region of query space where o is useful. For instance, if o is a posting list for centroid c, then $z(o)$ may be the centroid vector itself. If o is a cached neighborhood around a graph node v, then $z(o)$ can be the embedding of that node or a local average. A query embedding x is mapped to $u(x) \in \mathbb{R}^{d'}$ through a projection or feature transform, and the system estimates similarity $\sigma(u(x), z(o))$ to predict reuse probability. A typical choice is cosine similarity or negative squared distance, but the model is extensible. The

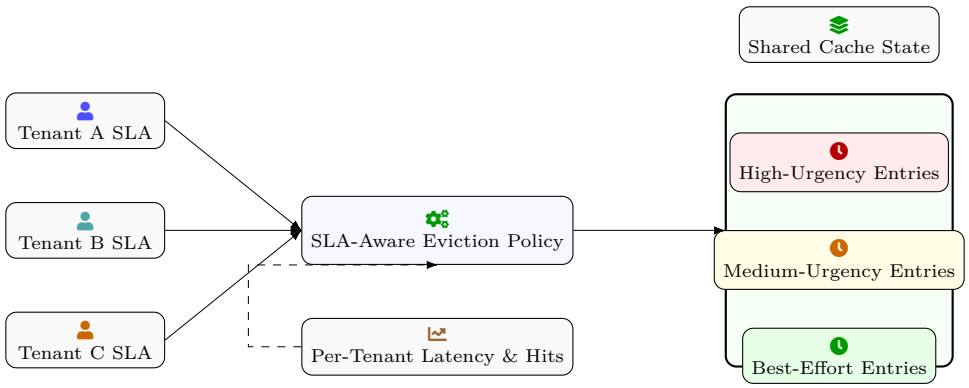

Figure 3. SLA-driven eviction mechanism for a shared cache. Per-tenant SLA signals and observed performance metrics are combined in a policy engine that assigns urgency levels to cached entries. The resulting priority bands bias eviction toward low-urgency items while preferentially protecting queries and tenants that are at risk of violating latency targets.

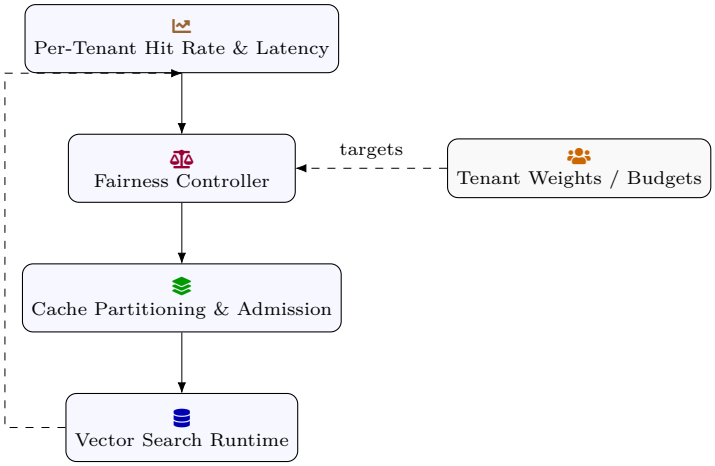

Figure 4. Fairness control loop enforcing per-tenant guarantees on a shared cache. Observed hit rates and latencies feed a controller that reconciles these metrics with configured tenant weights. The controller adjusts cache partitioning and admission decisions, closing the loop via runtime measurements to keep tenants close to their target service levels over time.

key requirement is fast evaluation and stable calibration under drift [7].

Each tenant has an SLA expressed as constraints on tail latency and possibly accuracy. Let L_t be the random variable of latency for tenant t queries. An SLA can be expressed as $\Pr(L_t > \tau_t) \leq \epsilon_t$ for a threshold τ_t and violation probability ϵ_t, or as a quantile constraint $\text{Quantile}_{\alpha_t}(L_t) \leq \tau_t$ for α_t such as 0.99. Accuracy constraints can be expressed in terms of recall-at-k, with approximate search returning a set $\widehat{R}$ compared to exact R, and requiring $\mathbb{E}[\text{recall@k}] \geq \rho_t$ or $\Pr(\text{recall@k} < \rho_t) \leq \delta_t$. These constraints interact: lowering compute can require more approximation, affecting recall. Query-aware caching primarily targets latency by reducing work while keeping ANN parameters stable, but it can also enable more expensive refinement within the same budget, improving recall.

The cache management problem spans admission and eviction. Admission decides whether to store an object produced or referenced by a query. Eviction chooses which objects to remove to satisfy memory budgets. In multi-tenant settings, admission and eviction must incorporate fairness [8]. We formalize fairness using utility allocations. Let U_t denote the long-run utility tenant t receives from caching, such as reduction in its SLA violation probability or reduction in its p99 latency. A fairness requirement can enforce proportional fairness by maximizing $\sum_t w_t \log(U_t + \eta)$ for weights w_t and small $\eta > 0$, or enforce max-min fairness by maximizing $\min_t U_t$ subject to constraints. Another operationally relevant constraint is that no tenant's SLA violation rate can exceed a multiple of another's beyond configured ratios, such as $\Pr(L_t > \tau_t) \leq \gamma_t$ with tenant-specific caps. The cache policy must produce allocations that satisfy these while adapting online to workload changes.

We now pose a concrete optimization. Let the cache state be a subset $\mathcal{C} \subseteq \mathcal{O}$, with total size constraint $\sum_{o \in \mathcal{C}} s(o) \leq B$, where B is the effective memory budget for the cache at a given scope. For a given tenant t, define the expected SLA risk under cache state $\mathcal{C}$ as

$$R_t(\mathcal{C}) = \Pr(L_t(\mathcal{C}) > \tau_t),$$

or, when using quantiles, define $R_t(\mathcal{C}) = \max(0, \text{Quantile}_{\alpha_t}(L_t(\mathcal{C})) - \tau_t)$. Define a global objective that balances SLA risk, energy, and fairness:

$$\min_{\mathcal{C} \subseteq \mathcal{O}} \sum_{t=1}^{T} w_t R_t(\mathcal{C}) + \beta E(\mathcal{C}) \quad \text{subject to} \quad \sum_{o \in \mathcal{C}} s(o) \leq B, \quad \Phi(\{U_t(\mathcal{C})\}_t$$

where $E(\mathcal{C})$ is an energy proxy, such as expected memory

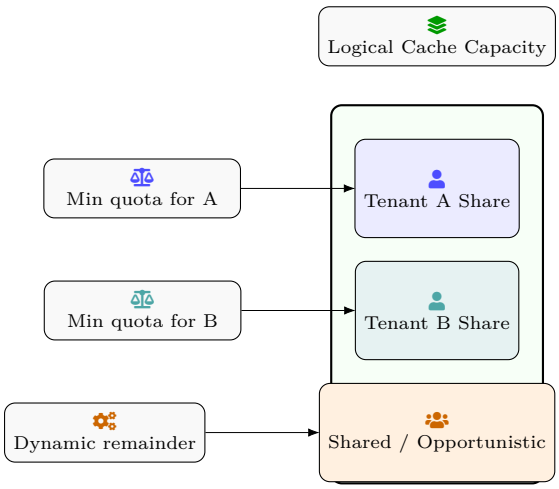

Figure 5. Logical cache partitioning for a multi-tenant workload. Each tenant receives a protected slice of the cache to guarantee a minimum hit rate, while a shared region absorbs bursty demand and hosts opportunistic entries. The controller can dynamically adjust slice sizes based on observed load and fairness objectives without violating per-tenant minimum allocations.

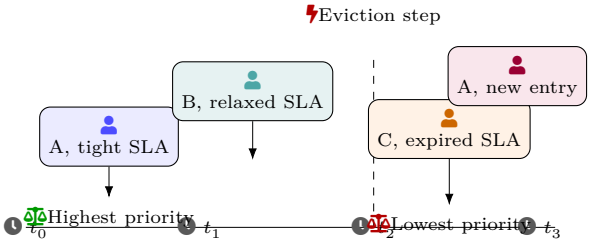

Figure 6. Timeline view of SLA-driven eviction decisions. Cached entries from different tenants carry distinct urgency levels based on remaining slack to their latency targets. At each eviction step, the policy considers both temporal information and tenant identity, favoring retention of items that remain critical for meeting per-tenant SLAs while demoting entries that have become less time-sensitive.

traffic or CPU cycles, and Φ encodes fairness constraints, for example $\Phi = \max_t(U^\star - U_t)$ for max-min against a target $U^\star$, or inequality constraints enforcing ratios. This formulation is combinatorial and depends on unknown distributions. It must be solved online, requiring estimators for R_t and U_t and fast eviction decisions.

The combinatorial core resembles knapsack with multiple objectives and fairness constraints [9]. Even if utility were additive, selecting objects to maximize weighted utility per byte is knapsack, which is NP-hard. In practice, utility is non-additive due to overlap: caching two correlated objects can provide diminishing returns because the same queries benefit from either. This can be modeled as submodularity. Let $F(\mathcal{C})$ denote total expected utility, then F is submodular when adding an object yields less marginal gain in larger sets. Submodular maximization under a knapsack constraint admits approximation via greedy selection under certain conditions, but the multi-tenant fairness constraints complicate it. For eviction, the online nature and the need for low overhead motivate policies derived from marginal utility estimates and primal-dual shadow prices.

To connect cache state to latency, we decompose per-query service time. For query q from tenant t with embedding x, let the baseline execution cost be

$$C(q) = C_{\text{route}}(x) + C_{\text{cand}}(x) + C_{\text{refine}}(x),$$

where routing cost includes centroid search or entry point selection, candidate generation includes graph expansions or list scans, and refinement includes exact distance computations and re-ranking [10]. Cache objects reduce one or more terms. If object o affects routing, then its marginal benefit is a reduction in C_{route}, and similarly for other terms. The latency includes queueing delay W and service time S, with $L = W + S$. Under queueing approximations, the tail of L is sensitive to utilization $\rho = \lambda\mathbb{E}[S]/k$, and cache-induced reductions in $\mathbb{E}[S]$ reduce ρ, which can lower $\Pr(W > w)$ sharply as ρ approaches 1. Consequently, utility models should emphasize tail impact, not only mean savings.

We represent cache utility per object as a tenant-weighted expected tail risk reduction:

$$g(o \mid \mathcal{C}) = \sum_{t=1}^{T} w_t \left(R_t(\mathcal{C}) - R_t(\mathcal{C} \cup \{o\})\right),$$

and define an eviction score incorporating size and fairness shadow prices:

$$\text{score}(o) = \frac{g(o \mid \mathcal{C} \setminus \{o\})}{s(o)} - \sum_t \mu_t \cdot h_t(o),$$

where μ_t are dual variables for fairness constraints and $h_t(o)$ measures how much object o contributes to tenant t utility. Objects with low score are evicted first. The challenge is that g is unknown and expensive to compute exactly; the

Tenant group	#Tenants	Avg QPS per tenant	Cache share (%)
Platinum	4	1,500	30
Gold	8	1,000	28
Silver	12	600	22
Bronze	16	300	15
Best-effort	20	150	5

Table 3. Multi-tenant mix and cache space allocation at steady state.

Policy	Query-aware	SLA-driven eviction	Time complexity
LRU	No	No	$O(1)$
LFU	No	No	$O(1)$
SLA-LRU	No	Yes	$O(1)$
QA-LRU	Yes	No	$O(\log k)$
QA-SLA-Cache	Yes	Yes	$O(\log k)$

Table 4. Comparison of evaluated cache management policies.

remainder of the paper develops estimators and efficient algorithms to approximate it while keeping overhead low [11].

3. Query-Aware Caching via Embedding Geometry and Low-Rank Intent Models

Query-aware caching relies on predicting which objects will be reused and how much they will reduce cost. In vector search, reuse is tied to similarity in the embedding space: queries near each other often route to similar partitions and explore overlapping neighborhoods. This is not guaranteed, especially when embeddings are anisotropic or when routing partitions are coarse, but it is sufficiently common that geometric prediction is useful. The system therefore maintains a compact representation of recent queries per tenant and a mapping from cached objects to the query regions where they are beneficial.

Let $x \in \mathbb{R}^d$ be the query embedding. Direct nearest-neighbor computation between x and historical queries is expensive. A common approach is to project embeddings to a lower-dimensional space. Let $P \in \mathbb{R}^{d' \times d}$ be a projection matrix with $d' \ll d$, and define $u = Px$. P can be a random Gaussian projection, a sparse sign projection, or a learned PCA basis. With random projections, the Johnson–Lindenstrauss effect motivates approximate distance preservation, while PCA captures dominant variance directions for the workload [12]. For multi-tenant workloads, a shared projection can be maintained for the whole service, or tenant-specific projections can be derived when tenants exhibit distinct embedding distributions. Tenant-specific PCA is expensive if done naively, but incremental PCA or randomized SVD can update principal directions from streaming queries.

We also maintain per-tenant intent summaries. Let $X_t \in \mathbb{R}^{n_t \times d}$ be a matrix of recent query embeddings for tenant t. A low-rank approximation $X_t \approx U_t \Sigma_t V_t^\top$ with rank r captures typical directions in query space. The rows of $V_t \in \mathbb{R}^{d \times r}$ span a tenant intent subspace. Mapping a query to coefficients $a = V_t^\top x$ yields a compact representation. This helps in two ways. First, it accelerates similarity comparisons, since r is small. Second, it enables prediction of which index partitions or graph regions are frequently visited, because these can be correlated with intent coefficients [13]. In practice, r might be on the order of tens, providing a substantial reduction in compute. The approximation error $\|x - V_t V_t^\top x\|_2$ can also be used to detect out-of-distribution queries and avoid overconfident caching decisions.

Cacheable objects are associated with signatures $z(o)$ and a mapping from signatures to predicted access. For an inverted-file index with K coarse centroids $\{c_i\}_{i=1}^K$, a query routes to a set of centroids based on distances $\|x - c_i\|_2$. Caching routing artifacts can include the top-n centroid IDs for a query cluster, or a precomputed shortlist for a region. For graph-based indices such as HNSW, queries start from entry points and traverse edges based on distance comparisons; caching can include hot entry points and neighborhoods. For product quantization, caching can include codebooks, lookup tables for asymmetric distance computation, and per-list centroid residual norms.

A central design choice is the level of granularity for query-aware caching. Caching per exact query embedding is rarely effective because embeddings are high entropy and repeated identical queries may be infrequent [**?**]. Instead, we cluster queries into regions. Let $\mathcal{K}$ denote a set of query clusters represented by prototypes $\{\pi_j\}$. A query x is assigned to a cluster via $\arg\min_j \|u(x) - \pi_j\|_2$. Clustering can be maintained online with limited overhead using streaming k-means updates in the projected space. Each cluster maintains statistics: frequency per tenant, typical routing decisions, and typical candidate set overlap. Cache admission can then store objects keyed by cluster ID rather than by exact query.

To estimate reuse probability, we define a similarity kernel between query embedding features u and object signature z:

$$p(o \mid x, t) \approx \kappa\left(u(x), z(o)\right) \cdot \theta_{t,o},$$

where κ is a bounded kernel such as $\exp(-\|u - z\|_2^2 / 2\sigma^2)$ or

Method	P99 latency (ms)	Hit rate (%)	Tail throughput (kQPS)
LRU	142	71	180
LFU	135	74	186
SLA-LRU	112	78	195
QA-LRU	96	84	207
QA-SLA-Cache	72	90	226

Table 5. End-to-end performance across cache management methods.

Tenant tier	Target P99 (ms)	Violation baseline (%)	Violation QA-SLA (%)
Platinum	30	12.4	1.3
Gold	60	15.1	2.7
Silver	100	18.9	4.5
Bronze	150	23.7	7.2
Best-effort	250	31.5	10.8

Table 6. SLA violation rates per tier under baseline and proposed policy.

a cosine-based mapping, and $\theta_{t,o}$ is a tenant-object affinity capturing tenant-specific patterns. $\theta_{t,o}$ can be estimated from counts: how often tenant t accessed o recently. To avoid heavy storage, we maintain sketches. A Count-Min sketch can approximate access frequencies for objects, while a reservoir sample of query clusters can approximate the distribution of $u(x)$ [14]. For geometric kernels, approximate nearest neighbor in the projected space can find the nearest prototypes or object signatures quickly.

Utility estimation requires predicting cost reduction, not only access probability. Let $c_0(q)$ be the baseline service time for query q without object o, and $c_1(q, o)$ the service time with o. The marginal benefit is $\Delta c(q, o) = c_0(q) - c_1(q, o)$. We model Δc as depending on the stage impacted and on system state. For example, caching a posting list avoids a disk fetch and reduces memory stalls, producing a relatively stable saving in service time, whereas caching a candidate set may reduce CPU distance computations but the saving depends on re-ranking thresholds and on whether the system adapts efSearch. To capture variability, we model Δc as a random variable conditioned on query cluster and tenant, and maintain empirical estimates of its mean and variance.

Because the SLA objective emphasizes tail behavior, we translate service time savings into tail risk reduction. A tractable approximation is to model per-tenant latency as the sum of queueing delay and service time, and to approximate the queueing tail using utilization [15]. For node m, utilization is $\rho_m = \lambda_m \mathbb{E}[S_m]/k_m$. Under high utilization, the p99 waiting time grows roughly like a factor that increases rapidly as ρ approaches 1. While the exact function depends on service time distribution, a conservative surrogate is to use a convex penalty in ρ, such as $\psi(\rho) = \frac{\rho}{1-\rho}$ for $\rho < 1$, which captures the blow-up near saturation. Then reducing mean service time by $\Delta\mathbb{E}[S]$ yields a reduction in $\psi(\rho)$ that is larger when ρ is high. This suggests a utility model:

$$\hat{g}(o) = \sum_t w_t \, \mathbb{E}_{q \sim Q_t} \left[p(o \mid q, t) \cdot \Delta c(q, o) \right] \cdot \Gamma_t,$$

where Γ_t is a tail sensitivity factor reflecting how close the tenant's workloads are to violating the SLA. Γ_t can be derived from observed violation rate gradients: if a small increase in mean latency increases violations significantly, then Γ_t is high. This factor implicitly accounts for queueing and contention without requiring a full queueing model.

Estimating $p(o \mid q, t)$ and $\Delta c(q, o)$ online can introduce noise. We explicitly propagate estimation uncertainty to avoid brittle decisions. Let $\hat{p}$ and $\widehat{\Delta c}$ be estimators with confidence intervals. For example, if access counts are tracked with exponential decay, the effective sample size determines variance [16]. For geometric kernels, projection distortion introduces error. We represent this by bounding the difference between true and estimated similarity. If $u = Px$ with a random projection and d' is sufficiently large, then distances are preserved up to multiplicative factors with some probability; practically, we measure empirical distortion and treat it as an uncertainty term. The utility estimator can then be conservative:

$$\hat{g}_{\text{low}}(o) = \sum_t w_t \, \mathbb{E} \left[(\hat{p} - \delta_p)_+ \cdot (\widehat{\Delta c} - \delta_c)_+ \right] \cdot \Gamma_t,$$

where δ_p, δ_c are uncertainty margins and $(\cdot)_+$ truncates at zero. Eviction based on $\hat{g}_{\text{low}}$ reduces the risk of removing an object whose true utility is high but poorly estimated.

Query-aware caching also interacts with accuracy. Many ANN indices trade recall for speed by restricting candidate exploration. Caching can indirectly improve recall if it saves compute that the system reinvests into deeper exploration or refinement [17]. To model this, define recall recall@k(q) as a function of candidate set size n_c and refinement threshold. If caching reduces compute by Δc, the system may increase n_c within the same latency budget. If recall as a function of n_c is concave, then marginal improvements diminish. We can represent a joint utility:

$$\tilde{g}(o) = \sum_t w_t \left(\alpha \cdot \Delta\text{TailLatency}_t(o) + (1 - \alpha) \cdot \Delta\text{Recall}_t(o) \right),$$

where $\alpha \in [0, 1]$ controls the trade-off. In many SLA settings, latency dominates, but for tenants with strict quality constraints, recall must be incorporated. This becomes a multi-objective optimization, and Pareto frontiers can be

Removed component	ΔHit rate (pts)	ΔP99 (ms)	Δfairness index
Query popularity model	-6.8	+14	-0.05
Tenant-level admission	-3.1	+9	-0.08
SLA-weighted eviction score	-4.5	+18	-0.12
Cross-tenant rebalancing	-1.9	+6	-0.04
All components	-9.7	+27	-0.16

Table 7. Ablation study of key components in the proposed design.

Component	CPU overhead (%)	Memory overhead (%)	Update latency (μs)
Popularity estimator	3.2	0.8	4.1
SLA-aware score	2.5	0.4	3.6
Fairness controller	1.7	0.6	5.3
Metadata index	4.1	1.2	6.8
Total (QA-SLA-Cache)	8.9	3.0	10.4

Table 8. Control-plane overhead of query-aware, SLA-driven caching mechanisms.

considered: some cache allocations yield lower latency but lower recall, others the reverse. A weighted-sum approach with tenant-specific α_t is practical, and dual variables can enforce minimum recall constraints.

To efficiently map query clusters to objects, we use hashed indices [18]. Locality-sensitive hashing provides approximate neighborhood search in projected space. A set of random hyperplanes defines hash bits $b_i = \text{sign}(r_i^\top u)$, forming a binary code. Nearby queries share codes with higher probability, allowing constant-time lookup in buckets. Each bucket maintains a small set of object candidates with high historical utility for that region. Admission can then consider caching objects associated with the current bucket and its Hamming neighbors. This reduces overhead, since the policy only updates a bounded candidate set per query.

The cache is not only a set of data but also an implicit model of workload. To avoid instability under sudden shifts, we introduce a notion of cache inertia. Each object o has a retention prior, favoring keeping objects that have historically been useful, unless evidence indicates a shift [19]. A Bayesian-like update can be used: treat utility as a latent parameter with a prior and update with observations of cost savings. If Δc observations are noisy, a normal-inverse-gamma model yields posterior mean and variance for expected savings. The eviction policy can then incorporate posterior uncertainty, penalizing eviction when uncertainty is high and the potential regret is large. While exact Bayesian inference may be too heavy at scale, approximate updates using exponential moving averages and variance estimates capture similar behavior.

4. SLA-Driven Eviction with Fairness Guarantees: Optimization, NP-Hardness, and Online Algorithms

Eviction in a multi-tenant query-aware cache must reconcile competing goals: minimizing SLA violations, ensuring fairness, and respecting resource budgets. The combination of combinatorial selection and multi-tenant constraints leads to computational hardness, motivating approximations and online methods. This section formalizes the eviction problem, establishes NP-hardness via reduction, and develops practical algorithms using primal-dual updates and differentiable relaxations that enable backprop-friendly tuning.

We begin with a simplified offline formulation [20]. Assume a finite set of objects $\mathcal{O}$ and a time horizon where tenant query distributions are stationary and known. Suppose each object o yields an additive per-tenant utility $u_t(o) \geq 0$ when cached, interpreted as reduction in tenant t SLA risk or an equivalent surrogate. Consider a fairness requirement that tenant utilities must meet minimum shares $U_t \geq \underline{U}_t$. The offline selection is

$$\max_{\mathcal{C}\subseteq\mathcal{O}} \sum_{t=1}^{T} w_t \sum_{o\in\mathcal{C}} u_t(o) \quad \text{s.t.} \quad \sum_{o\in\mathcal{C}} s(o) \leq B, \ \sum_{o\in\mathcal{C}} u_t(o) \geq \underline{U}_t \ \ \forall t.$$

Even with a single tenant, this becomes knapsack if $u_t(o)$ is the value and $s(o)$ is the weight. With multiple tenants and minimum constraints, it becomes a multi-dimensional knapsack variant. This problem is NP-hard. A direct reduction from knapsack follows by setting $T = 1$ and $\underline{U}_1 = 0$. A reduction from partition can also be constructed for a feasibility version: given numbers $a_1, \ldots, a_n$, set objects with sizes $s(o_i) = a_i$ and utilities $u_1(o_i) = a_i$, set $B = \frac{1}{2}\sum_i a_i$ and require $\sum_{o\in\mathcal{C}} u_1(o) = B$, which is equivalent to finding a subset with sum B. Thus exact optimization is not tractable at scale.

In practice, utilities are not additive due to overlap. If two objects benefit the same query region, caching both yields less than the sum of their individual benefits [21]. This is captured by submodularity: a set function $F(\mathcal{C})$ is submodular if $F(A\cup\{o\})-F(A) \geq F(B\cup\{o\})-F(B)$ whenever $A \subseteq B$. For query-aware caching, F often exhibits approximate submodularity because marginal benefits diminish with more cached coverage. Submodular maximization under a knapsack constraint admits greedy approximations when F is monotone, but fairness constraints break simple greedy behavior because the objective couples tenants.

We therefore focus on online policies with shadow prices. Let time be discrete queries indexed by n. The cache must maintain state under streaming arrivals. Each query from tenant t produces a set of candidate objects $\mathcal{A}_n \subset \mathcal{O}$ that could be admitted or whose statistics are updated. The

cache has capacity B. When admitting new objects, eviction may be needed to free space. The key is to define an eviction priority that incorporates SLA urgency and fairness.

We track per-tenant SLA deficit [22]. Let $\widehat{R}_t(n)$ be an estimator of tenant t violation probability over a sliding window, and let the target be ϵ_t. Define deficit $D_t(n) = \widehat{R}_t(n) - \epsilon_t$, clipped below at zero. A positive deficit indicates the tenant is currently exceeding its SLA. We also track per-tenant cache utility achieved, $\widehat{U}_t(n)$, such as cumulative latency savings or reduction in tail risk. Fairness can be imposed by targeting proportional allocations, for example requiring $\widehat{U}_t(n)$ to be at least a fraction of the weighted total. Define fairness deficit $F_t(n) = \max(0, \underline{U}_t(n) - \widehat{U}_t(n))$, where $\underline{U}_t(n)$ is a dynamic target derived from weights and load.

We introduce dual variables $\lambda(n)$ for memory and $\mu_t(n)$ for fairness deficits. The eviction/admission decision at time n uses a score for each object o in the cache:

$$\text{keep}(o) = \frac{\sum_t \left(w_t + \eta D_t(n)\right) \hat{u}_t(o) + \mu_t(n)\hat{u}_t(o)}{s(o)} - \lambda(n),$$

where $\hat{u}_t(o)$ is estimated marginal benefit to tenant t, and η controls how aggressively SLA deficits amplify weights. Intuitively, if tenant t is violating its SLA, objects that help t become more valuable to keep. If fairness deficit is high, μ_t increases, further boosting objects beneficial to that tenant. Objects with low keep score are evicted first.

Dual variables are updated online. A primal-dual scheme updates λ and μ_t based on constraint violations [23]. If total cache size exceeds B, increase λ; if tenant utility falls below target, increase μ_t. A simple update is

$$\lambda(n+1) = \left[\lambda(n) + \gamma_\lambda \left(\sum_{o \in \mathcal{C}(n)} s(o) - B\right)\right]_+,$$

$$\mu_t(n+1) = \left[\mu_t(n) + \gamma_\mu \left(\underline{U}_t(n) - \widehat{U}_t(n)\right)\right]_+,$$

with step sizes $\gamma_\lambda, \gamma_\mu$ and projection onto nonnegative reals. These updates mirror gradient ascent on the dual of a constrained optimization. In steady state, if targets are feasible, the dual variables stabilize and enforce constraints. If targets are infeasible, dual variables grow, pushing the policy to allocate as much as possible to the constrained tenants; operators can detect infeasibility by monitoring sustained growth and adjust targets.

The challenge is defining $\hat{u}_t(o)$ in a way that reflects tail risk. We define $\hat{u}_t(o)$ as estimated reduction in a tail surrogate. Let S_t be service time for tenant t queries on the relevant node, and let ρ be utilization. Consider a convex tail penalty $\psi(\rho)$ as before [24]. Since ρ depends on mean service time, a first-order approximation yields

$$\Delta\psi \approx \psi'(\rho) \cdot \Delta\rho = \psi'(\rho) \cdot \frac{\lambda}{k} \cdot \Delta\mathbb{E}[S].$$

If caching object o reduces mean service time for tenant t by $\Delta\mathbb{E}[S_t \mid o]$, then its utility in terms of tail penalty reduction is proportional to $\psi'(\rho) \cdot \Delta\mathbb{E}[S_t \mid o]$. For $\psi(\rho) = \frac{\rho}{1-\rho}$, $\psi'(\rho) = \frac{1}{(1-\rho)^2}$, which increases sharply near saturation. This formalizes the intuition that the same service time reduction matters more when the system is busy. Therefore, $\hat{u}_t(o)$ can be computed as

$$\hat{u}_t(o) = \hat{p}_t(o) \cdot \widehat{\Delta\mathbb{E}[S_t \mid o]} \cdot \frac{1}{(1-\hat{\rho})^2},$$

where $\hat{p}_t(o)$ is estimated access probability for tenant t.

This is a mean-based surrogate; tail SLAs often involve p99 directly. A practical method is to model per-tenant latency distribution using quantile regression. Let $q_{\alpha,t}$ be the predicted α-quantile of latency as a function of features, including estimated service time, queue depth, and cache hits. If the model is differentiable, we can compute a sensitivity of the quantile to an object's presence. For an object o, define a binary variable $c_o \in \{0, 1\}$ indicating whether it is cached. A relaxed continuous version uses $c_o \in [0, 1]$. The predicted quantile is $\widehat{Q}_{\alpha,t}(c)$, and utility is approximately $-\frac{\partial \widehat{Q}_{\alpha,t}}{\partial c_o}$. This enables gradient-based tuning of weights and dual variables. To keep the decision discrete, a rounding step selects objects with highest c_o under memory constraints, akin to fractional knapsack rounding [25]. While exact gradients depend on the regression model, a simple differentiable proxy uses a softmax-weighted hit indicator: if object access depends on similarity, model hit probability as $\sigma(\theta \cdot \sigma(u, z))$ and differentiate through σ.

A differentiable relaxation also supports learning eviction parameters. Suppose we parameterize keep scores by a vector ϕ controlling the relative importance of SLA deficit, fairness, and size normalization. The system can optimize ϕ to minimize observed SLA violations. Let loss at time n be $\ell_n(\phi) = \sum_t w_t \max(0, L_t(n) - \tau_t)$ or a smooth approximation. The cache decision is non-differentiable, but a straight-through estimator or Gumbel-Softmax relaxation can approximate gradients. For example, define a relaxed cache inclusion probability

$$\pi_o = \frac{\exp(\text{keep}_\phi(o)/\tau)}{\sum_{o'} \exp(\text{keep}_\phi(o')/\tau)},$$

with temperature τ [26]. The expected cache size constraint can be enforced via a Lagrangian. Then gradients of expected loss with respect to ϕ can be computed, enabling adaptive tuning. This approach must be engineered carefully to avoid instability and to ensure it does not add excessive overhead.

Fairness guarantees require more than heuristic weighting; they require that the policy enforces bounds on allocation or utility. One approach is to define fairness in terms of regret relative to per-tenant optimal caching. Let OPT_t be the best utility tenant t could achieve if it had exclusive cache budget B_t, and define a fairness requirement that each tenant receives at least a fraction α_t of OPT_t under shared caching. Computing OPT_t exactly is NP-hard, but approximate upper bounds can be obtained via fractional relaxation. The system can maintain a per-tenant upper bound $\widehat{\text{OPT}}_t$ and enforce $\widehat{U}_t \geq \alpha_t \widehat{\text{OPT}}_t$. Dual variables μ_t then enforce these. While this does not yield absolute guarantees against the unknown true optimum, it provides a robust mechanism to avoid extreme starvation.

Another fairness notion is Jain's index over utilities: [27]

$$J = \frac{\left(\sum_t U_t\right)^2}{T \sum_t U_t^2}.$$

A minimum $J \geq J_{\min}$ constraint is nonconvex, but a practical surrogate is to penalize variance of U_t. Define a penalty $\Omega = \sum_t (U_t - \overline{U})^2$, where $\overline{U}$ is mean utility. Adding $\kappa\Omega$ to the objective discourages imbalance. Online, this can be implemented by increasing weights for tenants below mean and decreasing for those above. This resembles proportional fairness when weights are inversely related to achieved utility.

Eviction is executed under strict time budgets, so algorithmic complexity matters. If the cache holds C objects, sorting by score is $O(C \log C)$, which may be too expensive per query. Instead, we maintain approximate priority queues or bucketed scores. Scores can be quantized into bins, enabling $O(1)$ approximate eviction by selecting from the lowest bin. To update scores without scanning the whole cache, we exploit that only a small set of objects is touched per query: those associated with the query's hash buckets and routing decisions [28]. We maintain lazy score updates: each object stores its last computed score and a timestamp; when considered for eviction, its score is recomputed using current dual variables and statistics. This yields amortized efficiency.

To further reduce overhead, we restrict eviction candidates to a victim set. When admitting a new object of size s, we sample a small set of existing objects proportional to size or inverse score, recompute their scores, and evict the worst until enough space is freed. This resembles randomized caching policies and provides good performance under heavy churn. The trade-off is that it may miss the globally worst objects, but in large caches the distribution of low-score objects can be broad, making sampling effective.

Because cache objects may be shared across tenants, we must define attribution of utility. If an object is used by multiple tenants, it contributes to multiple utilities [29]. This can create positive externalities, improving overall efficiency, but can also create perverse incentives if one tenant's load causes eviction of shared beneficial objects that others depend on. The dual-weighted scoring naturally favors shared objects when they provide multi-tenant utility. However, fairness constraints might require that shared objects count toward the utilities of the tenants that benefit. We define per-tenant utility attribution as the fraction of accesses: if object o is accessed $a_t(o)$ times by tenant t in a window, then its attributed utility to tenant t is proportional to $a_t(o)$. This is implementable using sketches. Then fairness constraints operate on attributed utilities, preventing a policy from claiming fairness by keeping objects that only benefit dominant tenants.

Finally, we address the interaction between eviction and approximate search error. Some cached artifacts can change the effective candidate set and thus recall. If caching is used to shortcut stages, it may increase approximation error [30]. We include an error constraint. Let $e_t(\mathcal{C})$ denote expected recall shortfall for tenant t. We can enforce $e_t(\mathcal{C}) \leq \bar{e}_t$. In the dual framework, introduce dual variables ν_t for error constraints, and include ν_t-weighted penalties in scoring. Practically, if an object increases error, it will be penalized, and if error constraints are tight, ν_t increases, pushing the policy toward safer objects.

5. Data Structures, Storage Internals, and Distributed Execution Mechanics

Implementing query-aware caching requires careful integration with ANN index structures and distributed execution. The core algorithms can be correct in an abstract model yet fail to provide benefit if overheads dominate or if cache placement conflicts with locality. This section describes cacheable objects and their representation, discusses storage internals and memory hierarchy considerations, and outlines distributed execution patterns that preserve fairness and SLA semantics.

Vector search indices commonly combine coarse partitioning with fine search. In an inverted-file pipeline, vectors are assigned to coarse centroids [31]. A query finds nearest centroids and scans associated inverted lists. With product quantization, vectors are stored as compact codes; re-ranking may fetch full-precision vectors. Cacheable objects include centroid lookup results for query clusters, inverted lists for hot centroids, PQ codebooks and precomputed lookup tables, and frequently accessed full-precision vectors for refinement. In graph-based indices, vectors are nodes with edges; cacheable objects include adjacency lists and neighborhood expansions for hot regions, as well as entry points. In either case, caching can occur at multiple layers: CPU caches, process memory, page cache, and distributed in-memory stores.

A key decision is object granularity. Caching entire inverted lists can be expensive in memory if lists are large. Instead, one may cache only the head portion likely to be scanned first, or a compressed representation [32]. For PQ, an inverted list stores codes and IDs; scanning requires reading codes and computing approximate distances. Caching codes in a compressed contiguous layout reduces memory stalls. If lists are stored on SSD, caching avoids random reads. For graph indices, caching adjacency of hot nodes reduces pointer chasing and improves CPU branch predictability. However, storing adjacency lists may not be enough; layout matters. A cache that stores a neighborhood in a contiguous block aligned to cache lines can reduce memory latency. Thus cache objects should be defined not only by logical content but also by physical layout.

Compression is central [33]. Cache memory is limited, and objects differ in compressibility. For PQ, codes are already compressed, but auxiliary data may be large. We can further compress cached lists using entropy coding if symbol distributions are skewed. For example, if IDs in a list exhibit locality, delta encoding and variable-length codes reduce size. Let a sequence of IDs be $i_1 < i_2 < \cdots < i_m$. Store deltas $\delta_j = i_j - i_{j-1}$. If deltas follow a distribution with entropy H, the expected code length lower bound is H bits per delta. Practical codes approach this with overhead. The cache scoring should incorporate compressed size $s(o)$, not raw size, and should consider decompression cost in Δc [34]. Sometimes a smaller compressed object with decompression overhead can be worse for tail latency than a slightly larger uncompressed object if decompres-

sion is CPU-bound. Therefore, the utility model should include CPU cycles for decompression and memory bandwidth saved.

In-memory representation must also respect NUMA locality. On multi-socket servers, accessing remote memory incurs additional latency and bandwidth contention. If cache objects are accessed frequently by threads pinned to a socket, storing them in the local NUMA node improves performance. Query-aware caching can incorporate placement: an object can have multiple replicas across NUMA nodes or across machines. Replication improves locality but increases memory usage. This creates another optimization: choose replication factor $r(o)$ for object o to balance locality benefits with memory [35]. A simple model is that expected access cost decreases with $r(o)$ but with diminishing returns. Replication decisions can be included in the optimization via additional variables. In a relaxed formulation, let $c_{o,m} \in \{0,1\}$ denote whether object o is placed on node m, with constraint $\sum_o s(o)c_{o,m} \le B_m$. The utility of o is then a function of whether it is present on the node serving the query. This becomes a placement problem akin to facility location and is NP-hard, so heuristics are required. A practical approach is to place objects based on observed access locality: if a shard receives most queries for an object, place it there; if access is spread, replicate to a limited number of top nodes.

Distributed vector search typically uses sharding by item ID, by centroid partitions, or by tenant. Queries may be broadcast to shards and results merged. In such architectures, caching can be local per shard or global [36]. Local caches are simpler and exploit locality, but may produce unfairness if some shards serve more tenants than others. Global caches can share objects across shards, but access latency increases due to network hops and contention. A hybrid design uses local caches for latency-critical objects and a second-tier cache for less frequently accessed objects. Admission policies can decide which tier to place objects into, based on predicted reuse frequency and size. For example, a frequently probed centroid list can be placed locally, while a less frequent but large neighborhood can be placed in a second tier.

Consistency is another concern. Index structures evolve as items are added, deleted, or updated. Cached objects must remain valid. For inverted lists, updates append entries; cached lists can become stale [37]. For graph indices, insertions can change neighborhood structure. A strict consistency model requires invalidating or updating cached objects upon index changes, which can be expensive. Many systems accept bounded staleness, especially if it affects only performance, not correctness. If cached objects are intermediate artifacts, staleness may only reduce utility, not break correctness, as long as final candidate evaluation uses current vectors. However, if cached objects include precomputed candidate sets, staleness can affect accuracy. Therefore, objects should carry version metadata: an object is associated with an index epoch, and queries check epoch compatibility. If incompatible, the object is treated as a miss. To avoid frequent invalidations, choose objects that are stable, such as codebooks updated infrequently, or centroid partitions that change slowly [38].

Performance engineering requires minimizing per-query overhead. Query-aware caching introduces additional computations: projecting embeddings, hashing, updating sketches, and computing scores. If these costs exceed saved execution time, caching is counterproductive. The design should therefore bound overhead. Projection can be implemented as a matrix multiply $u = Px$; if P is sparse with $O(d')$ nonzeros per row, this is $O(d' \cdot \text{nnz})$ operations. Alternatively, if embeddings are already produced by a model, one can reuse intermediate low-dimensional features. Hashing via sign of dot products is also fast if projection vectors are sparse. Sketch updates are constant time. Score computations should be limited to a small candidate set [39].

Storage internals matter for tail latency. Many systems store vectors and codes in memory-mapped files backed by SSD, relying on OS page cache. Cache objects that avoid page faults provide large tail benefits. However, page cache behavior can be unpredictable under memory pressure. A managed cache inside the process can avoid page cache eviction anomalies by pinning objects. Yet pinning too much can cause the OS to reclaim other useful pages. Therefore, the cache budget should be coordinated with the OS, either by using hugepages and explicit allocation or by controlling memory cgroups. Tail latency is also influenced by GC pauses in managed runtimes; if the implementation uses a GC language, caching large objects can increase GC pressure [40]. A systems approach might allocate cache in off-heap memory and manage it manually.

On modern hardware, CPU caches and SIMD can accelerate distance computations. Query-aware caching can store prepacked data aligned for SIMD operations, such as contiguous PQ codes. For inner product, storing vectors in a transposed blocked layout can improve cache utilization. Caching such packed blocks yields greater speedup than caching raw lists because it reduces both memory stalls and instruction overhead. The utility estimator should therefore incorporate the speedup from vectorization, which can vary by object. Measuring this empirically and storing per-object microbenchmarks can calibrate utility.

GPU acceleration adds another layer [41]. GPUs excel at batched distance computations, but transferring data between CPU and GPU and managing GPU memory is complex. A query-aware cache can include GPU-resident objects such as codebooks and hot vector blocks. Eviction then must consider GPU memory constraints and transfer costs. A two-level eviction with separate dual variables for CPU RAM and GPU VRAM is appropriate. Let B_{cpu} and B_{gpu} be budgets. Objects have two sizes: CPU footprint and GPU footprint. Some objects can be stored in both; others only on CPU. The optimization becomes multi-constraint knapsack. A practical heuristic is to prioritize GPU residency for objects that accelerate compute-heavy stages and are reused frequently, and to keep infrequently used objects on CPU [42].

Distributed fairness enforcement also requires careful design. If eviction is decided independently per node, fairness constraints may be violated globally. For example, a tenant's queries may hit multiple nodes; local fairness might allocate too little on each. A solution is to maintain global fairness signals: each node periodically reports per-tenant utilities and deficits to a coordinator, which computes global dual variables μ_t and broadcasts them. Nodes

then use these common dual variables in scoring. This keeps fairness consistent across the cluster. The coordinator does not need detailed per-object data, only aggregate utilities and SLA violation estimates, keeping overhead manageable. The system must tolerate delays: dual variables updated periodically introduce lag [43]. To mitigate oscillations, use small step sizes and smoothing.

Queueing and admission control interact with caching. If a tenant exceeds its rate, the system can throttle or shed load, which changes the utility of caching. In some deployments, SLAs are enforced by rate limits and priority scheduling. Caching can be integrated by giving higher priority to queries from tenants with high SLA deficits, and by using cache allocations to reduce their service times. This resembles a joint scheduling and caching problem. A simplified approach is to treat scheduling as given and focus caching on reducing service times. However, joint optimization can provide better fairness. For instance, a tenant with low volume but strict SLA could receive both higher scheduling priority and more cache allocation, reducing tail [44]. The dual variables can conceptually unify these: the same fairness deficit that increases caching weight can also increase scheduling weight. Implementing this requires careful coordination to avoid feedback loops where increased priority increases load on shared resources and worsens others' tail.

6. Approximation Error, Robustness Under Drift, and Multi-Objective Trade-offs

Vector search is inherently approximate in most large-scale deployments. Query-aware caching must operate within and sometimes alter the approximation regime. It must also remain robust under workload drift, tenant churn, and evolving embedding models. This section analyzes approximation error interactions, derives bounds and error propagation for sketches and projections, and discusses multi-objective optimization trade-offs including energy and budget constraints.

Approximate nearest neighbor indices return results that may differ from exact nearest neighbors. The approximation arises from restricting the search space [45]. In inverted-file indices, only a subset of lists is scanned. In graph indices, search explores a limited number of nodes. In quantization-based indices, distances are approximated by code-based estimates. Recall depends on parameters such as the number of probed lists, efSearch, and quantizer resolution. Caching can affect these in two ways. First, caching can reduce the latency cost of increasing search parameters, enabling deeper exploration and potentially increasing recall. Second, caching can encourage shortcutting certain computations, which might reduce accuracy if done improperly, such as reusing candidate sets across similar queries without sufficient validation.

We analyze error when reusing artifacts across similar queries [46]. Suppose an artifact o encodes a candidate set C_o produced for a prototype query π, and for a new query x in the same cluster, the system reuses C_o as an initial candidate pool. The risk is that C_o misses true nearest neighbors for x. Let $f_x(y)$ denote true similarity score. The exact top-k set is R_x. If C_o does not contain R_x, recall drops. A sufficient condition for reuse is that similarity ordering does not change much between π and x. For inner product, the difference in score is $f_x(y) - f_\pi(y) = \langle x - \pi, y \rangle$. If $\|x - \pi\|_2$ is small and $\|y\|_2$ is bounded, then score differences are bounded by Cauchy-Schwarz: $|\langle x - \pi, y \rangle| \le \|x - \pi\|_2 \|y\|_2$ [47]. If there is a margin between the k-th and $(k+1)$-th neighbors for π, reuse may preserve the top-k set. Specifically, if for all $y \in R_\pi$ and $y' \notin R_\pi$ we have $f_\pi(y) - f_\pi(y') \ge 2\Delta$ and if $|\langle x - \pi, y \rangle| \le \Delta$ for all candidates, then $R_x = R_\pi$. In practice, margins are small and norms vary, so exact preservation is rare, but this analysis motivates conservative reuse: only reuse candidate sets when cluster radius is small and when observed margins are large. The system can estimate margin statistics per cluster and disable candidate reuse when margins are low.

For quantization, distance approximation introduces error. Consider product quantization where a vector y is approximated by $\hat{y}$, and distance estimates use $\|x - \hat{y}\|_2^2$ instead of $\|x - y\|_2^2$. The error is

$$\|x - \hat{y}\|_2^2 - \|x - y\|_2^2 = \|y - \hat{y}\|_2^2 + 2\langle x - y, y - \hat{y} \rangle.$$

If quantization error $\|y - \hat{y}\|_2$ is bounded by ϵ_q and vector norms are bounded, the distance error can be bounded. Cached PQ lookup tables reduce compute but not error; however, caching may enable using higher-resolution PQ or larger candidate sets, reducing recall loss. Utility models should therefore reflect not only latency savings but also how savings might be spent to improve recall [48]. This is multi-objective: latency and recall.

We incorporate multi-objective optimization explicitly. Let $J(\mathcal{C})$ be a vector of objectives: J_1 is weighted SLA violation risk, J_2 is expected energy consumption, J_3 is recall shortfall. A Pareto-optimal cache state is one where no objective can be improved without worsening another. In practice, operators choose weights and constraints. A common approach is weighted sum:

$$\min_{\mathcal{C}} \quad \theta_1 J_1(\mathcal{C}) + \theta_2 J_2(\mathcal{C}) + \theta_3 J_3(\mathcal{C}),$$

with $\theta_i \ge 0$. Another approach is constrained optimization: minimize SLA risk subject to energy and recall constraints. The primal-dual method naturally supports the constrained form via Lagrange multipliers [49]. For example, to constrain energy, define $E(\mathcal{C}) \le \bar{E}$, introduce dual variable ζ, and include $\zeta(E(\mathcal{C}) - \bar{E})$ in the Lagrangian. Online, ζ updates based on observed energy consumption. Energy estimation can use hardware counters for memory bandwidth and CPU cycles. If the cache increases memory traffic by keeping large objects hot, energy may increase; if it reduces compute, energy may decrease. The policy balances these using ζ.

Robustness under drift is essential. Workloads drift due to seasonality, tenant behavior changes, and embedding model updates. Query-aware caching relies on historical similarity and utility estimates, which can become stale. A naive policy may keep objects that were useful but no longer are, wasting memory [50]. Conversely, it may evict objects needed for new workloads. We address drift with adaptive time scales. Each statistic, such as access frequency and cost savings, is maintained with exponential decay. Different tenants can have different decay rates depending on

stability. A tenant with stable workload can use slow decay, preserving long-term knowledge; a tenant with volatile workload uses fast decay, adapting quickly. Selecting decay rates is itself an optimization. One can estimate drift by monitoring prediction error: if predicted access probability differs from observed, drift is high, and decay should increase.

We also handle tenant churn [51]. Tenants may arrive and depart, and their SLAs may change. Fairness targets must adjust. If a new tenant arrives with strict SLA, the policy must allocate cache quickly without destabilizing others. Dual variables provide a mechanism: initialize new tenant dual variables to a moderate value and ramp them based on observed deficits. To avoid sudden eviction storms, use admission control: limit the rate at which cache allocation can change per tenant. This can be represented by a constraint on utility change: $|U_t(n+1) - U_t(n)| \leq \Delta_t$. Enforcing this exactly is complex, but heuristically one can cap dual variable step sizes and limit object admissions per query for new tenants.

Sketches and projections introduce their own approximation error. Count-Min sketch overestimates counts due to collisions [52]. If access frequency estimates are biased upward, the policy might keep objects believed to be popular. To mitigate this, use conservative updates and multiple sketches per tenant or per object class. Alternatively, track top objects explicitly using heavy-hitter algorithms and use sketches for the long tail. Projection distortion affects geometric similarity. If random projection dimension d' is too small, clusters may overlap. A practical calibration is to sample pairs of query embeddings, compute true cosine similarity and projected similarity, and fit a correction function. The system can then map projected similarity to predicted reuse probability with calibration that accounts for distortion.

We analyze the impact of these approximations on eviction decisions [53]. Suppose the true utility per byte of object o is $v(o)$ and we estimate $\hat{v}(o) = v(o) + \epsilon(o)$ with error ϵ. If eviction removes objects with smallest $\hat{v}$, errors can cause removing a valuable object. The expected regret depends on error variance and on the gap between utilities of objects near the eviction threshold. If the utility distribution near the threshold is flat, small errors can cause large churn but low regret because many objects have similar utility. If the distribution has sharp gaps, errors can cause high regret by removing a uniquely valuable object. Therefore, uncertainty-aware scoring that uses lower confidence bounds is valuable in regimes with sharp utility gaps. The system can estimate utility gaps by examining the score distribution: if the lowest scores are tightly clustered, risk of catastrophic eviction is low; if there is a distinct separation, it is safer.

Gradient descent variants appear in updating model parameters such as calibration weights and dual variable step sizes. While dual updates are a form of gradient ascent on constraints, additional model parameters can be learned. For example, the kernel bandwidth σ in κ affects reuse predictions [54]. A too-small σ makes reuse overly local, missing beneficial reuse; too-large σ overgeneralizes. We can adapt σ by minimizing prediction loss between predicted and observed accesses. Let $a_n(o)$ be an indicator of whether object o was accessed on query n, and predicted probability $\hat{p}_n(o;\sigma)$. Minimize negative log-likelihood $\sum_n -a_n \log \hat{p}_n - (1-a_n)\log(1-\hat{p}_n)$. Stochastic gradient descent updates σ. Because σ must remain positive, parameterize $\sigma = \exp(\xi)$ and update ξ. Adaptive optimizers such as Adam can stabilize updates in nonstationary environments. However, using such learning online requires caution; the control loop must be slow relative to query processing to avoid oscillations.

We also consider explicit constraints on latency and budget. Suppose the operator specifies a total memory budget B and a per-tenant minimum utility share [55]. The Lagrangian is

$$\mathcal{L}(\mathcal{C}, \lambda, \mu) = \sum_t w_t R_t(\mathcal{C}) + \lambda \left(\sum_{o \in \mathcal{C}} s(o) - B \right) + \sum_t \mu_t \left(\underline{U}_t - U_t(\mathcal{C}) \right).$$

The primal-dual updates aim to find a saddle point. While $\mathcal{C}$ is discrete, we approximate with fractional variables $c_o \in [0,1]$ representing inclusion probability. Then U_t and size become linear, and with a convex surrogate for R_t, the problem becomes convex, enabling gradient methods. Rounding yields a discrete cache. This provides a principled way to design scoring functions: the coefficient of c_o in $\mathcal{L}$ yields a keep score. Even when the surrogate is imperfect, it aligns eviction with the operator's objectives.

A final trade-off is between fairness and efficiency. Enforcing strict max-min fairness can reduce total utility if some tenants have low cacheability, meaning their workloads do not benefit much from caching. Allocating cache to such tenants yields low returns but improves fairness [56]. Proportional fairness balances this by allocating more to tenants with higher marginal returns while still giving nonzero shares to others. Operators may choose fairness based on business priorities. The system can present a tunable fairness knob, such as a parameter controlling the curvature of the fairness objective. For instance, maximizing $\sum_t w_t \frac{U_t^{1-\alpha}}{1-\alpha}$ interpolates between utilitarian ($\alpha = 0$) and max-min-like behavior as α increases. Online, the dual variables and scoring incorporate this curvature.

7. Evaluation Methodology, Metrics, and Reproducibility Considerations

A caching policy for multi-tenant vector search must be evaluated on tail latency, SLA violation rates, and fairness, not merely hit rate. It must also be evaluated under realistic workload dynamics, including bursts, tenant churn, and drift. This section specifies an evaluation methodology emphasizing controlled experiments, trace-driven simulation, and reproducibility, and defines metrics that reflect the objectives of SLA-driven fairness-aware caching.

Workloads can be constructed from real traces when available, or from synthetic generators calibrated to realistic properties [57]. Query embeddings exhibit temporal locality: similar queries tend to occur in bursts due to user behavior or upstream batching. They also exhibit anisotropy and clustering. A synthetic generator can model this by sampling cluster centers and generating queries around them with time-correlated selection. For tenant t, define clusters with centers $\pi_{t,j}$ and mixture weights that evolve

over time. A Markov chain over clusters induces temporal locality. Query arrivals follow a nonhomogeneous Poisson process to capture bursts. Per-tenant volumes differ by orders of magnitude, creating multi-tenancy imbalance. SLAs differ: some tenants require p99 below a threshold, others have looser constraints.

The index configuration must be specified because cache objects depend on the ANN pipeline [58]. Evaluation should include at least one inverted-file with PQ configuration and one graph-based configuration. For each, define baseline parameters yielding acceptable recall and latency without caching. Then enable caching and measure improvements. It is important to separate the benefit of caching from the benefit of changing ANN parameters. If caching allows deeper search, measure the combined effect and also isolate the effect by holding ANN parameters fixed.

Metrics include per-tenant and aggregate. For latency, measure mean, p50, p90, p99, and p99.9. SLA violation rate is the fraction of queries exceeding τ_t. Tail metrics should be measured under steady load and during bursts, because caching often helps steady-state but may fail under burst-induced cache churn [59]. For fairness, measure the distribution of per-tenant utilities U_t, as well as Jain's index J. Also measure maximum relative violation: $\max_t \widehat{R}_t/\epsilon_t$, which indicates which tenants are most harmed. If fairness constraints are configured, report the fraction of time constraints are satisfied.

Utility measurement requires defining what counts as utility. A natural choice is reduction in SLA violation probability relative to a baseline policy. Another is reduction in p99 latency. Because tenants have different thresholds, normalized metrics are helpful, such as $\frac{\mathrm{p99}_t}{\tau_t}$ and $\frac{\widehat{R}_t}{\epsilon_t}$. For energy, measure CPU cycles, memory bandwidth, and optionally power if available. Cache overhead should be measured: time spent in projection, hashing, sketch updates, and eviction decisions. A policy that reduces service time by a small amount but adds overhead may not improve end-to-end latency [60].

Cache-centric metrics still matter. Measure hit rate by object type, such as routing artifacts, lists, neighborhoods, and vectors. Measure byte hit rate, which accounts for size. Measure churn: evictions per second and admissions per second. High churn can indicate instability and can increase overhead. Measure reuse distance distributions to understand whether cached objects are reused within their TTL. For query-aware caching, also measure cluster quality: average within-cluster distance and the fraction of accesses captured by top clusters. Drift can be quantified by comparing the distribution of clusters over time [61].

Evaluation should compare against baselines. Baselines include LRU and LFU on a suitable key space, such as object IDs, as well as size-aware variants like GreedyDual. A query-unaware SLA-weighted cache that prioritizes tenants by SLA deficits but ignores embedding similarity is also a relevant baseline. Another baseline is static partitioning: allocate fixed cache shares per tenant. The proposed policy should be evaluated in scenarios where static partitioning works well and where it fails, such as when tenant cacheability differs over time.

Because the proposed policy uses models and sketches, calibration matters. Evaluation should report sensitivity to projection dimension d', clustering parameters, sketch sizes, decay rates, and dual step sizes. Instead of enumerating a large grid, one can report representative settings and show robustness by varying each parameter within a reasonable range and observing impact on SLA and fairness [62]. It is also important to evaluate under different memory budgets, such as 10%, 30%, and 50% of the working set size, because caching behavior changes dramatically in these regimes.

Reproducibility requires deterministic configurations or controlled randomness. Random projections and hashing introduce randomness; store seeds and report them. For distributed systems, scheduling nondeterminism affects tail latency. Use repeated runs and report confidence intervals. When traces are proprietary, provide synthetic generators and describe how to calibrate them. Provide enough details about index configurations, vector dimensionality, dataset size, and hardware to allow replication. The evaluation should also record software versions and configuration flags, because micro-optimizations can alter results [63].

A trace-driven simulator is valuable for exploring algorithmic variants at scale. The simulator can model query arrivals, cache decisions, and service times using empirical distributions. It can incorporate queueing by simulating a multi-server queue with service times dependent on cache hits. This allows exploring tail behavior under different utilization. However, simulators risk missing low-level effects such as NUMA and page faults. Therefore, complement simulation with microbenchmarks of cache object access costs and with an end-to-end prototype. The prototype can be partial, focusing on critical paths. For example, implement query-aware caching for inverted-list caching and measure real latency improvements under controlled load, validating the simulator's service-time model [64].

Finally, fairness evaluation should consider edge cases. Some tenants may have workloads that do not benefit from caching. In such cases, strict fairness may allocate memory inefficiently. The evaluation should report how the policy behaves: whether it recognizes low cacheability and adjusts targets, or whether it wastes cache. Another edge case is a tenant with adversarial workload designed to thrash the cache, such as rapidly changing query regions. The policy's drift adaptation should prevent thrashing from harming others, possibly by isolating such tenants via fairness and by limiting their impact on shared cache.

8. Conclusion

Query-aware caching for multi-tenant vector search must go beyond traditional hit-rate maximization by incorporating embedding geometry, ANN execution structure, explicit SLA semantics, and fairness constraints. The central difficulty is that cacheable objects are diverse and their benefits depend on query similarity, index routing, and system load, particularly under tail-sensitive SLAs where small service time reductions can produce large improvements near saturation [65]. A practical approach models cached artifacts as objects with embedding-linked signatures, estimates reuse and cost reduction using low-rank intent models and sketches, and drives eviction using online constrained optimization with primal-dual shadow prices

that encode memory, fairness, and optional accuracy or energy constraints. While the underlying selection problems are NP-hard, the derived scoring policies and approximations provide implementable mechanisms that adapt to drift and tenant churn without requiring expensive global optimization.

Systems considerations are inseparable from the algorithmic design. Object granularity, compression, layout, NUMA locality, and distributed placement determine whether cached artifacts translate into real tail improvements. Fairness must be enforced across nodes, typically via shared dual variables derived from aggregate per-tenant deficits. Approximation error from ANN algorithms and from caching-specific shortcuts can be managed by conservative reuse rules and by integrating accuracy constraints into the same dual framework. Multi-objective tradeoffs among latency, recall, and energy can be expressed through weighted sums or constraints and addressed with Lagrangian methods.

An evaluation methodology emphasizing per-tenant tail metrics, SLA violation rates, fairness indices, overhead accounting, and reproducibility is necessary to assess such systems. The techniques described here provide a technical basis for building caches that respect multi-tenant objectives and for engineering them into distributed vector search services where performance predictability and equitable resource allocation are explicit goals rather than emergent properties [66].

References

[1] D. Woos, Z. Tatlock, M. D. Ernst, and T. Anderson, "A graphical interactive debugger for distributed systems.," 6 2018.

[2] X. Fu, "Issta - towards scalable defense of information flow security for distributed systems," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 438–442, ACM, 7 2019.

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

[4] 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.

[5] 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.

[6] D. Sindičić and N. Momčilović, "Efficient container image distribution for multi-tenant kubernetes clusters," in *2025 MIPRO 48th ICT and Electronics Convention*, pp. 1973–1978, IEEE, 6 2025.

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

[8] V. Karagiannis, P. A. Frangoudis, S. Dustdar, and S. Schulte, "Context-aware routing in fog computing systems," *IEEE Transactions on Cloud Computing*, vol. 11, pp. 532–549, 1 2023.

[9] "2023 4th international conference on information science, parallel and distributed systems (ispds)," in *2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*, pp. 1–1, IEEE, 7 2023.

[10] L. Bull and H. Liu, "A generalised dropout mechanism for distributed systems.," *Artificial life*, vol. 29, pp. 146–152, 5 2023.

[11] 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.

[12] *AAMAS - Achieving Sybil-Proofness in Distributed-Work Systems*, 9 2021.

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

[14] *Blockchain for Distributed Systems Security*. Wiley, 4 2019.

[15] K. A. Jothy, K. Sivakumar, and M. J. Delsey, "Distributed system framework for mobile cloud computing," *Bonfring International Journal of Research in Communication Engineering*, vol. 8, pp. 05–09, 2 2018.

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

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

[18] 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.

[19] "Parameterized synthesis of concurrent and distributed system," 11 2023.

[20] I. Schagaev and S. Farrell, *Distributed Systems: Resilience, Desperation*, pp. 267–292. Springer International Publishing, 7 2019.

[21] "Ieee distributed systems online," 3 2023.

[22] M. Kawulok, S. Maćkowska, M. Maćkowski, and D. Spinczyk, "Reducing blind students' learning effort with an audio-tactile approach to understanding basic computer algorithms," in *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, pp. 472–478, ACM, 6 2025.

[23] K. Akinepalli, "Microservices with spring boot: Simplifying distributed systems," *International Journal For Multidisciplinary Research*, vol. 6, 10 2024.

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

[25] V. Anumolu, "Building secured microservices and distributed systems," *INTERNATIONAL JOURNAL OF INFORMATION TECHNOLOGY AND MANAGEMENT INFORMATION SYSTEMS*, vol. 16, pp. 717–739, 3 2025.

[26] L. Colombi, S. Dahdal, E. D. Caro, R. Fronteddu, A. Gilli, A. Morelli, F. Poltronieri, M. Tortonesi, N. Suri, and C. Stefanelli, "Efficient data dissemination via semantic filtering at the tactical edge," in *MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM)*, pp. 457–462, IEEE, 10 2024.

[27] I. Murturi and S. Dustdar, "Decent: A decentralized configurator for controlling elasticity in dynamic edge networks," *ACM Transactions on Internet Technology*, vol. 22, pp. 1–21, 8 2022.

[28] S. Akter, A. Hossain, and M. R. R. Akanda, "A noble security analysis of various distributed systems," *International Journal of Engineering, Science and Information Technology*, vol. 1, pp. 62–71, 4 2021.

[29] D. Karnehm and A. Neve, "Adaptive compressing electric vehicle battery pack measurements using polynomial coding," in *2024 IEEE International Conference on Smart Mobility (SM)*, pp. 141–146, IEEE, 9 2024.

[30] L. Zhang and S. Fulton, "Discussion of quantum consensus algorithms," 1 2022.

[31] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H.-J. Bungartz, L. C. Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, and O. Z. Koseomur, "precice v2: A sustainable and user-friendly coupling library.," *Open research Europe*, vol. 2, pp. 51–51, 9 2022.

[32] R. Darbali-Zamora, J. Johnson, and M. J. Reno, "Parametric analysis of photovoltaic inverters under balanced and unbalanced voltage phase angle jump conditions," in *2023 IEEE 50th Photovoltaic Specialists Conference (PVSC)*, pp. 1–6, IEEE, 6 2023.

[33] 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.

[34] M. Alsayasneh, V. Quéma, and R. Lachaize, "The case for dynamic placement of distributed systems components," 7 2018.

[35] S. Dahdal, F. Poltronieri, M. Tortonesi, C. Stefanelli, and N. Suri, "A data mesh approach for enabling data-centric applications at the tactical edge," in *2023 International Conference on Military Communications and Information Systems (ICMCIS)*, pp. 1–9, IEEE, 5 2023.

[36] "Journal of security in computer networks and distributed systems," 3 2024.

[37] A. Poshtkohi and M. B. Ghaznavi-Ghoushchi, *Implementing Parallel and Distributed Systems*. Auerbach Publications, 2 2023.

[38] C.-F. Cheng and C.-W. Huang, "The harmonized consensus protocol in distributed systems," *The Journal of Supercomputing*, vol. 75, pp. 7690–7722, 8 2019.

[39] M. Farhadi, D. Miorandi, and G. Pierre, "Blockchain enabled fog structure to provide data security in iot applications," 1 2019.

[40] *IM - Tofino + P4: A Strong Compound for AQM on High-Speed Networks?*, 5 2021.

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

[42] C. Hanane, A. Battou, and O. Baz, "Performance security in distributed system: Comparative study," *International Journal of Computer Applications*, vol. 179, pp. 29–33, 1 2018.

[43] A. Balalaie and J. A. Jones, "Socc - towards a library for deterministic failure testing of distributed systems," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 486–486, ACM, 11 2019.

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

[45] 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.

[46] I. B. Fink, I. Kunze, P. Hein, J. Pennekamp, B. Standaert, K. Wehrle, and J. Rüth, "Advancing network monitoring with packet-level records and selective flow aggregation," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*, pp. 1–6, IEEE, 5 2025.

[47] N. Naik, *SysCon - Performance Evaluation of Distributed Systems in Multiple Clouds using Docker Swarm*. IEEE, 4 2021.

[48] 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.

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

[50] 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.

[51] B. J. D. Mello, F. Viel, C. A. Zeferino, T. Ribeiro, F. Ribeiro, E. A. Bezerra, and J. R. Pinheiro, "Operationalization of yolov11 in ros 2 for computer vision applied to human-robot interaction," in *2025 17th Seminar on Power Electronics and Control (SEPOC)*, pp. 1–6, IEEE, 11 2025.

[52] I. Colonnelli and M. Aldinucci, "Workflow models for heterogeneous distributed systems," 5 2022.

[53] M. Qiu and S.-Y. Kung, "Special issue on 'smart computing and communication' in international journal of parallel, emergent and distributed systems," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, pp. 217–218, 5 2020.

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

[55] A. Miller, *Blockchain for Distributed Systems Security - Permissioned and Permissionless Blockchains*. Wiley, 3 2019.

[56] A. R. Pratama, F. J. Simanjuntak, A. Lazovik, and M. Aiello, *APPIS - Low-power Appliance Recognition using Recurrent Neural Networks*. 3 2018.

[57] H. Bazille, E. Fabre, and B. Genest, "Certification formelle des réseaux neuronaux profonds : un état de l'art en 2019," 11 2019.

[58] Z. Wang, H. Chen, Y. Wang, C. Tang, and H. Wang, "The concurrent learned indexes for multicore data storage," *ACM Transactions on Storage*, vol. 18, pp. 1–35, 1 2022.

[59] Y. Teng, F. Zhao, J. Liu, M. Zhang, J. Duan, and Z. Shi, *SeTS*

3

: A Secure Trajectory Similarity Search System, pp. 522–526. Germany: Springer International Publishing, 4 2022.

[60] M. Eddoujaji, H. Samadi, and M. Bohorma, *Data Processing on Distributed Systems Storage Challenges*, pp. 795–811. Germany: Springer Singapore, 10 2021.

[61] S. Gupta, R. Verma, and N. Dhanda, "Introduction to next-generation internet and distributed systems," 7 2024.

[62] B. K. Shukla, B. Parashar, V. K. Singla, and S. Verma, "Autonomy and adaptive architectures in distributed systems," 11 2025.

[63] 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.

[64] M. Krogh-Jespersen, A. Timany, M. E. Ohlenbusch, S. O. Gregersen, and L. Birkedal, *ESOP - Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems*, pp. 336–365. Germany: Springer International Publishing, 4 2020.

[65] Z. J. Hamad and S. R. M. Zeebaree, "Recourses utilization in a distributed system: A review," 10 2021.

[66] Z. Dong, Z. Wang, X. Zhang, X. Xu, C. Zhao, H. Chen, A. Panda, and J. Li, "Fine-grained reexecution for efficient batched commit of distributed transactions," *Proceedings of the VLDB Endowment*, vol. 16, pp. 1930–1943, 6 2023.