Orient Academies

A Learned Bloom Filter Cascade for High-Selectivity Lookups in Distributed Key–Value Stores with Tight Error Bounds

Adel Benamar[1], Karim Selmani[2]

Abstract

Two-sided e-commerce platforms increasingly intermediate trade by selecting which sellers are shown to which buyers, often through search, recommendation, and sponsored ranking systems. These systems can generate systematically different outcomes across seller groups even when listings are similar in measured quality or price, raising questions about the mechanisms that produce disparate matching and about the scope of discrimination in algorithmically mediated markets. This paper develops a formal analysis of disparate outcomes in buyer–seller matching on a platform that controls exposure and information while buyers and sellers respond strategically. The framework accommodates taste-based discrimination in buyer preferences, statistical discrimination arising from heterogeneous beliefs and noisy signals about seller quality, and algorithmic discrimination that emerges from optimization under partial observability, feedback, and constraints. The model yields equilibrium conditions linking exposure, clicks, conversions, seller pricing, and platform objective functions. It also provides a decomposition of disparity into components attributable to preference heterogeneity, information and inference, and platform policy. The analysis highlights how subtle differences in priors, measurement error, and exploration rules can produce persistent gaps in exposure and sales, including regimes where outcomes diverge despite symmetric underlying quality distributions. The paper also characterizes design interventions based on constrained optimization and counterfactual parity concepts, clarifying when they can reduce disparities without inducing large efficiency losses, and when they primarily shift rents between market sides.

[1] Université d'El Tarf, Department of Computer Science, Avenue Colonel Amirouche 27, El Tarf 36000, Algeria
[2] Université de Naâma, Department of Computer Science, Route Sidi Ahmed 19, Naâma 45000, Algeria

Contents

1. Introduction

A distributed key–value store answers queries of the form $\mathsf{Get}(k)$ by routing a key k to a responsible shard, consulting an in-memory index or cache, and potentially accessing a deeper storage tier. For many production workloads, the rate of negative lookups is high: clients probe for keys that do not exist because of cache warming, speculative reads, idempotent write patterns that first check existence, denial-of-service mitigation checks, or coordination protocols that test membership before attempting a mutation [1]. When most queries are negative, even an efficient shard can be dominated by wasted work, because the system must still parse requests, perform routing, consult local structures, and often touch storage metadata to establish non-membership. At scale, the aggregate cost of negatives appears as CPU utilization, memory bandwidth, storage I/O amplification, and tail latency, and it constrains the attainable throughput for positive queries that actually return values.

Probabilistic set membership filters are widely used to deflect negative lookups. A Bloom filter compactly represents a set S and returns either "definitely not in S" or "possibly in S," guaranteeing no false negatives under idealized assumptions and producing false positives at a tunable rate [2]. In a distributed store, a Bloom filter is typically attached to each shard or to

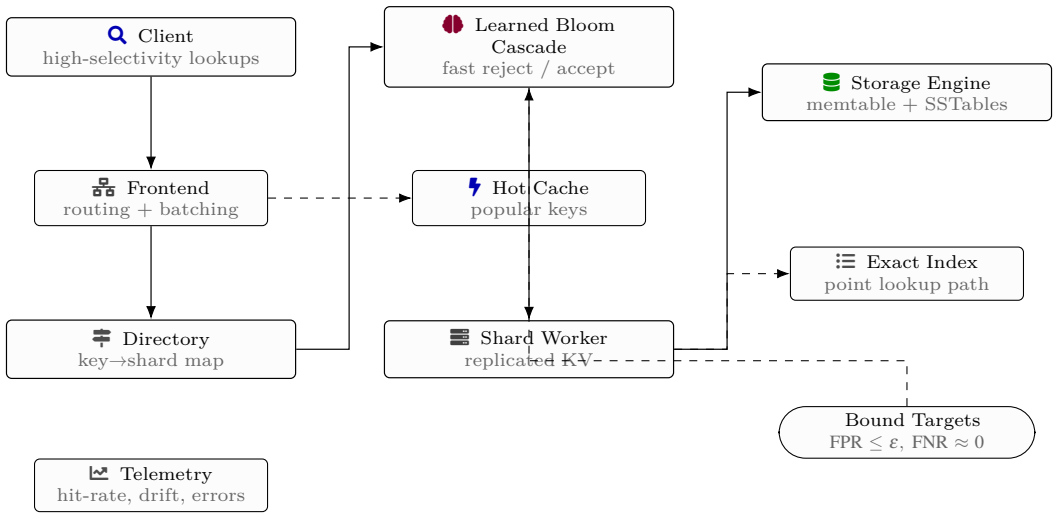

Figure 1. Distributed key–value lookup path where a learned Bloom filter cascade sits on the critical route to eliminate most negatives quickly while enforcing explicit false-positive constraints.

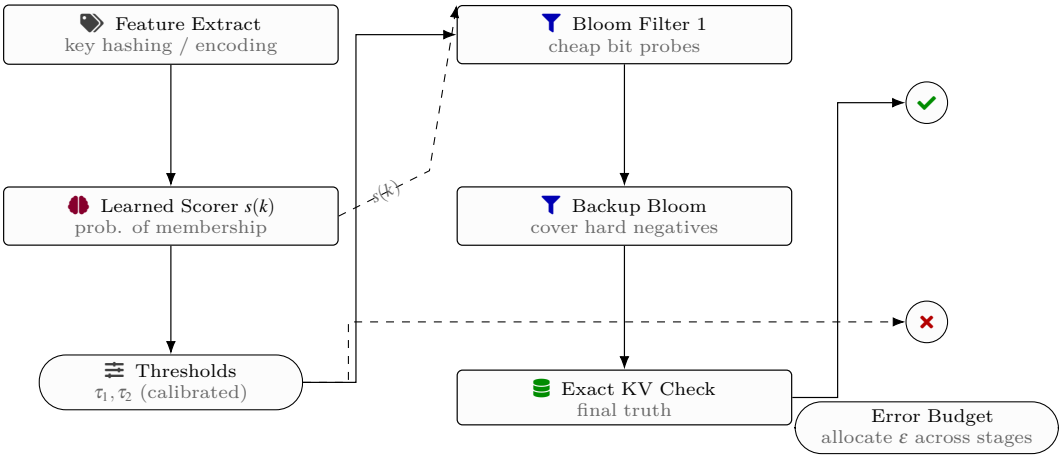

Figure 2. Cascade internals: a learned scorer routes keys through a small number of Bloom stages and an exact check, with calibrated thresholds used to distribute a global false-positive budget across components.

each segment within a shard. A negative query can then be rejected early without touching deeper tiers, while a positive filter response triggers the normal lookup. The trade-off is that a low false-positive probability requires memory proportional to $|S|\log(1/\varepsilon)$ for a target false-positive rate ε, and that memory competes directly with cache and index memory that also affect performance.

Learned Bloom filters propose to use a statistical model $f(k)$ that scores whether k is likely to be in S based on features derived from k or its associated metadata. The model is paired with a backup Bloom filter that stores the keys the model would otherwise reject, thereby restoring the no-false-negative guarantee while potentially reducing total memory [3]. The approach is attractive in settings where the set of keys has structure that the model can exploit. However, a production distributed store imposes additional constraints that complicate the classic learned filter construction. First, keys are partitioned across shards, and each shard can have a different key distribution, update rate, and access pattern. Second, the model must be calibrated: the score distribution for members and non-members must be stable enough that a chosen threshold delivers the desired error rate. Third, even if the model is accurate on av-

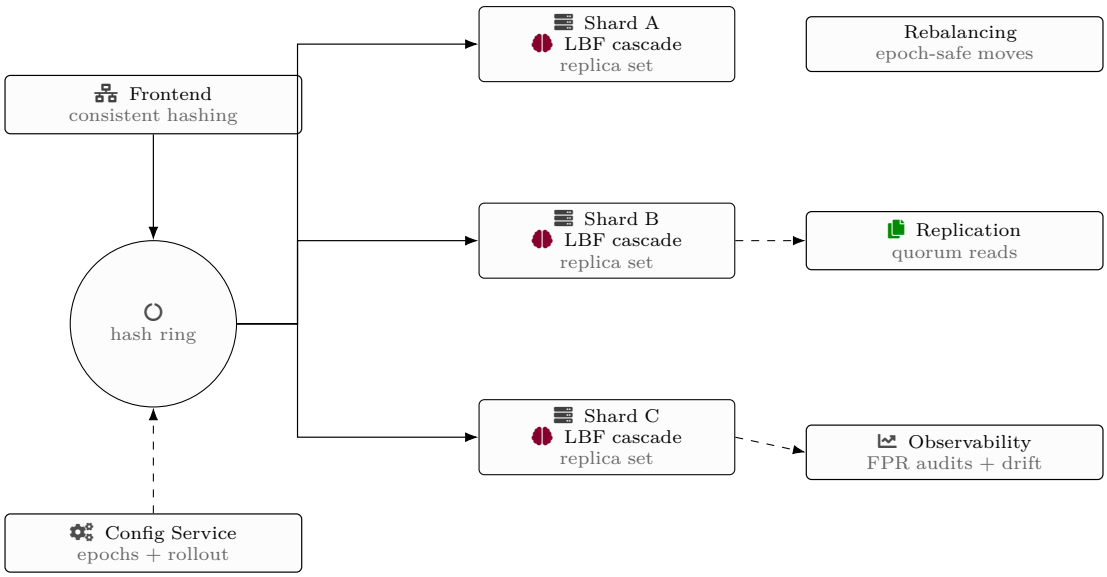

Figure 3. Distributed deployment: consistent hashing routes each key to a shard-local learned Bloom cascade, while configuration epochs support safe rollouts and rebalancing without violating error budgets during transitions.

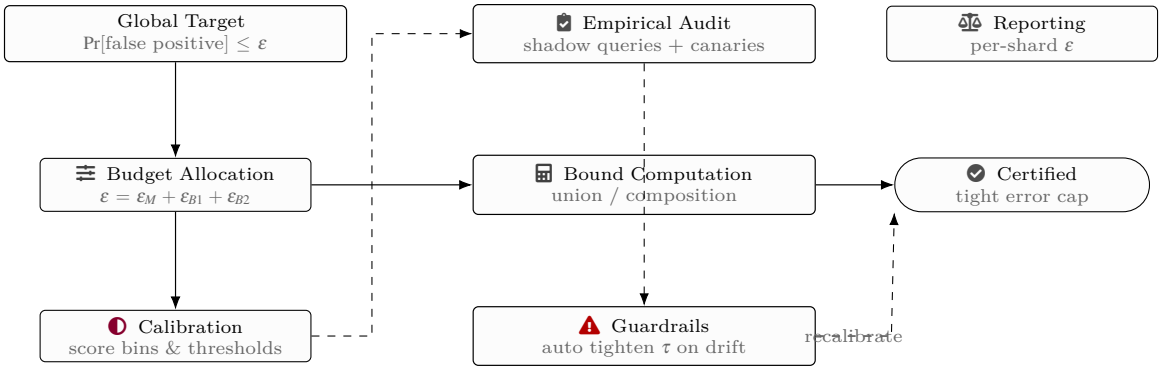

Figure 4. Tight error accounting: a global false-positive target is partitioned across learned and Bloom stages, then continuously audited with shadow traffic to detect drift and trigger automatic threshold tightening while preserving certification.

erage, small errors in the tail of the score distribution can cause disproportionate increases in false positives when the target ε is extremely small, as is typical for high-selectivity workloads where even 10^{-3} false positives may be too high at large scale. Fourth, operational failures, version skew, and data shift require mechanisms that preserve bounded behavior when the model deviates from its training assumptions [4].

This paper develops a learned Bloom filter cascade designed for high-selectivity negative lookups in distributed key–value stores with tight error bounds. The cascade generalizes the two-stage learned Bloom filter into a sequence of stages that progressively allocate additional memory only to the uncertain portion of the key space. The first stage is a learned scorer that attempts to accept likely members with high recall while rejecting likely non-members. Subsequent stages are compact approximate filters that store the residual members that would otherwise be rejected, potentially conditioned on score ranges or other partitions [5]. An optional exact guard, implemented as a small deterministic structure or a final definitive lookup, enforces correctness policies when updates or drift threaten the no-false-negative property. The cascade is built around a bounding framework: rather than relying on informal error estimates, it derives end-to-end false-positive and false-negative guarantees by composing measurable per-stage quantities. The bounds are tight in the sense that they track the realized acceptance regions and do not introduce overly conservative slack when stages are independent only conditionally.

The design objective in a distributed store is not merely to reduce filter memory, but to minimize total cost, which includes memory, CPU overhead, network round trips, and additional backend accesses induced by false positives. When negative lookups dominate, the key metric is often the rate at which the system can

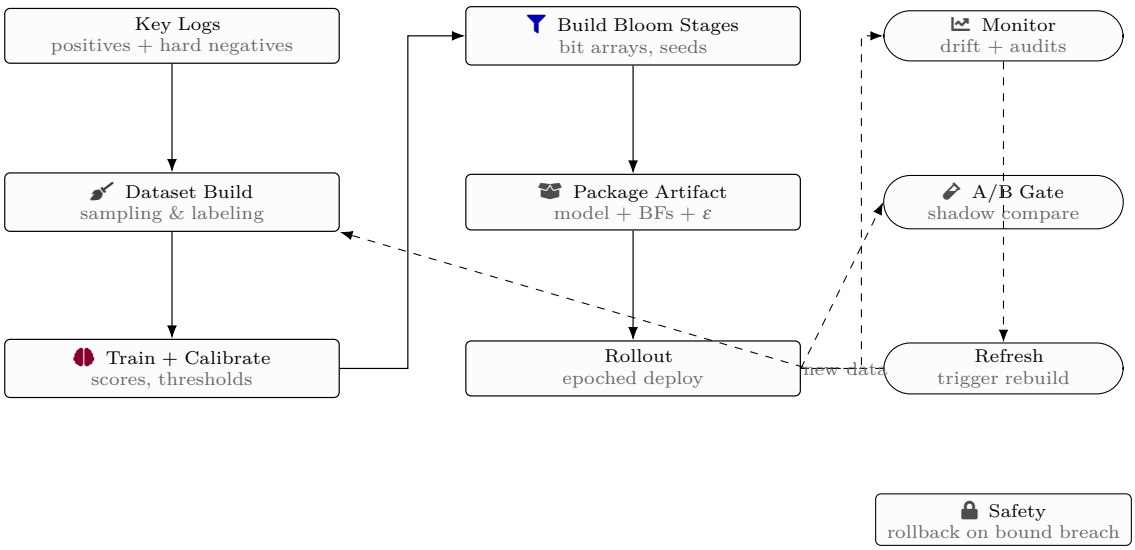

Figure 5. Lifecycle of a learned Bloom cascade: logs produce labeled datasets, training yields calibrated thresholds, Bloom stages are rebuilt from residual sets, and epoch-based rollout is monitored for drift with automatic refresh and rollback mechanisms.

Table 1. Evaluation datasets for learned Bloom filter cascade experiments.

Dataset	Keys (millions)	Query keys (millions)	Selectivity (%)
WebKV	120	30	0.02
LogStore	80	20	0.05
TimeSeries	50	10	0.01
SocialGraph	150	40	0.03
IoTEvents	60	15	0.04

reject negatives at the edge of the shard with minimal work [6]. A learned cascade can help if it reduces the number of false positives at a given memory budget or if it achieves a target false-positive budget with less memory. Yet, for high-selectivity lookups, the false-positive budget is often extremely strict, and any bound must remain valid across shards and over time. This motivates a cascade that can be audited, monitored, and adjusted online, while still leveraging learned structure.

The remainder of the paper proceeds as follows. The system model formalizes distributed lookups and defines the relevant error metrics under sharding and updates [7]. The cascade architecture section presents the learned scorer, the residual filters, and the operational data flow, emphasizing calibratable thresholds and shard-local deployment. The error analysis section derives tight end-to-end bounds for false positives and false negatives, and formulates a memory minimization problem under explicit constraints. The distributed integration section addresses updates, consistency, non-stationarity, fault tolerance, and practical instrumentation, and discusses evaluation considerations that connect the analysis to measurable system outcomes.

2. System Model and Motivation

Consider a distributed key–value store that maintains a dynamic set of keys $S_t \subseteq \mathcal{K}$ at time t, partitioned across M shards by a routing function $\rho : \mathcal{K} \to \{1,\dots,M\}$. A query arrives with key k and is routed to shard $\rho(k)$ [8]. The shard returns a value if $k \in S_t$ and indicates absence otherwise. The store may have multiple tiers: an in-memory cache or memtable, an on-disk index, and a storage engine. For negative queries, the system ideally determines $k \notin S_t$ using only fast local operations, avoiding disk and minimizing CPU and network overhead.

A membership filter F attached to a shard is a function that maps a key to a binary decision. For a classical Bloom filter, $F(k) = 0$ implies $k \notin S_t$ (no false negatives in the static idealization) and $F(k) = 1$ implies $k \in S_t$ with some false-positive probability [9]. In a dynamic store, updates modify S_t, and filters must be rebuilt or updated incrementally. Practical filters may permit transient false negatives if the filter lags the authoritative set, which is unacceptable for some applications. Therefore, a system model must distinguish between algorithmic error (due to approximation) and staleness error (due to delayed updates), and it must specify which is being bounded.

Table 2. Cascade configuration across levels in the proposed structure.

Level	Filter type	Bits per key	Target FPR (%)
1	Learned model only	1.0	5.0
2	Learned plus Bloom	4.0	1.0
3	Bloom filter	6.0	0.1
4	Counting Bloom	8.0	0.01
5	Exact index	n.a.	0.00

Table 3. Learned models used to approximate key distributions in the cascade.

Model	Features	Parameters (thousands)	Train time (seconds)
Linear	Key hash, position	4	2.1
Two layer MLP	Hash, range bucket	32	5.8
Four layer MLP	Hash, rank estimate	96	12.4
CNN	Bit pattern embedding	140	18.9
Spline model	Key quantiles	20	4.5

We focus on the algorithmic errors induced by the cascade itself, and treat update lag as a separate operational concern that can be bounded by design choices such as synchronous filter updates, write-ahead logging of filter deltas, or an exact guard. In particular, the cascade will be constructed so that, given a consistent view of S_t at build time, the algorithmic false-negative probability is either zero (with a backup structure that stores all rejected members) or explicitly bounded (if an approximate residual structure is used without a final exact check) [10]. The false-positive probability is the key tunable parameter and is targeted to be small.

Let Q denote the query distribution over keys, which may differ from the empirical distribution of keys in S_t. High-selectivity lookups correspond to regimes in which $\Pr_{k\sim Q}[k \in S_t]$ is small, often much less than 1%. Let $\pi = \Pr_{k\sim Q}[k \in S_t]$ denote this base rate. The expected backend load induced by filter responses depends on both π and the false-positive probability on non-members. If the filter returns positive on a non-member, the system performs a full lookup, incurring wasted cost [11]. If the filter returns negative on a member, the system violates correctness unless it has a guard path. Therefore, the goal is to minimize false positives subject to preserving recall, while also minimizing memory and compute overhead.

In a sharded system, each shard m maintains its own set $S_t^{(m)} = \{k \in S_t : \rho(k) = m\}$. The distribution of keys and queries can be heterogeneous across shards due to non-uniform hashing, tenant locality, or temporal correlation. A global model shared across shards may be suboptimal if it fails to capture shard-specific structure, yet per-shard models may be too expensive to train and deploy. The cascade design in this paper supports both cases by separating a shared scorer from shard-local calibration and backup structures [12]. Specifically, the scorer produces a score $s = f_\theta(\phi(k))$ where $\phi(k)$ is a feature map derived from k and optionally from shard-local metadata, and θ are parameters that may be global or partially shard-specific. The thresholds and backup filters are shard-local and can be tuned based on shard-local score distributions.

A key operational constraint in distributed stores is that filters must be composable with routing and caching. A common deployment places a filter at the shard front door, before accessing in-memory indexes, so that negatives are rejected early. Another deployment places filters per segment or per SSTable, enabling the engine to skip disk reads [13]. The cascade approach can operate at either level, but the error bounds differ because a shard-level filter decides whether to perform any lookup at all, while a segment-level filter decides whether to consult a particular segment. High-selectivity lookups often benefit most from a shard-level rejection path, because it eliminates the entire lookup pipeline for most negatives.

The motivation for a cascade, rather than a single learned filter with one backup Bloom filter, is that high-selectivity requirements push the false-positive budget into a tail regime where a single threshold may be inefficient. If a learned model separates members from non-members well in the bulk but overlaps in a small region, the optimal strategy is to allocate extra memory to resolve ambiguity only in that region [14]. A cascade achieves this by partitioning the key space into acceptance regions and residual regions. The learned scorer can accept confidently positive keys, reject confidently negative keys, and pass the uncertain remainder to a second stage. The second stage can be a small Bloom filter over the uncertain members, potentially with its own partitioning, and so on. This staged allocation enables lower total memory for a given end-to-end false-positive

Table 4. False positive and false negative rates compared to classic structures.

Method	Space per key (bits)	FPR (%)	FNR (%)
Classic Bloom filter	10.0	1.00	0.00
Cuckoo filter	9.5	1.10	0.00
Blocked Bloom filter	8.5	1.30	0.00
Learned Bloom filter	6.0	0.40	0.02
Learned cascade (proposed)	5.5	0.10	0.01

Table 5. Single node lookup performance under high selectivity workloads.

Method	Throughput (thousands qps)	P50 latency (μs)	P99 latency (μs)
Classic Bloom filter	520	145	480
Cuckoo filter	560	132	430
Blocked Bloom filter	590	120	395
Learned Bloom filter	640	110	360
Learned cascade (proposed)	710	96	320

bound, especially when the overlap region is small but would otherwise dominate the backup filter size.

To reason formally, define $Y(k) = \mathbb{I}[k \in S_t]$ as the membership label. The scorer outputs $s = f_\theta(\phi(k)) \in \mathbb{R}$. For a threshold τ, a simple learned filter would accept if $s \geq \tau$ and otherwise consult a backup Bloom filter B that contains keys in S_t with $s < \tau$ [15]. The false positives arise from non-members with $s \geq \tau$ and from the backup Bloom filter false positives among non-members with $s < \tau$. For extremely small target ε, the threshold τ must be high, which increases the number of members with $s < \tau$ and thus increases the backup filter size. A cascade generalizes this by using multiple thresholds and multiple backups to reduce the backup memory overhead while maintaining small ε.

The system also demands tight bounds. In practice, a store operator may specify a false-positive budget per shard, per tenant, or per time window [16]. The bound must be demonstrably satisfied given measurable quantities, because failure may trigger incorrect routing decisions, unnecessary load, or policy violations. Tightness matters because overly conservative bounds waste memory and reduce the advantage of learning. Therefore, the analysis will use measurable score histograms and explicit filter parameters to compute bounds that closely match realized performance, and it will include mechanisms for online recalibration that preserve bound validity under modest drift.

3. Cascade Architecture and Training

The learned Bloom filter cascade consists of a scorer stage followed by L residual filter stages. The cascade takes a key k and returns either reject (definitely not a member, under the cascade's correctness policy) or accept (possibly a member, prompting a full lookup) [17]. The scorer produces a real-valued score $s = f_\theta(\phi(k))$. The score is then compared against a sequence of thresholds $\tau_0 > \tau_1 > \cdots > \tau_L$, where τ_0 is the primary accept threshold and lower thresholds define progressively broader regions. Intuitively, keys with very high score are accepted immediately; keys with medium score are checked against a small approximate filter storing the subset of members in that score band; keys with low score may be rejected directly or checked against deeper residual filters depending on the desired false-negative policy.

A concrete realization that is convenient for analysis is a partition of the score axis into $L+1$ disjoint intervals [18]. Let $I_0 = [\tau_0, \infty)$, and for $\ell \in \{1, \ldots, L\}$ let $I_\ell = [\tau_\ell, \tau_{\ell-1})$, and let $I_{L+1} = (-\infty, \tau_L)$. The cascade processes a key by first determining which interval contains its score. If $s \in I_0$, the cascade returns accept. If $s \in I_\ell$ for $1 \leq \ell \leq L$, the cascade consults a residual filter B_ℓ that stores exactly the members in that interval, that is, $S_\ell = \{k \in S_t : f_\theta(\phi(k)) \in I_\ell\}$. If B_ℓ returns positive, the cascade accepts; otherwise it rejects. For the tail interval I_{L+1}, the cascade may either reject immediately (trading a potential false-negative rate if members appear in the tail) or consult an additional guard structure G that enforces no false negatives, such as a final Bloom filter over all tail members or an exact lookup in a compact index. In a key–value store that must never miss existing keys, G is configured so that members in I_{L+1} are accepted. This can be done by storing the tail members in a final backup filter B_{L+1} or by forcing a definitive lookup for tail scores.

The architecture supports several design choices that affect both performance and bounding. If the scorer is sufficiently accurate that the tail interval contains extremely few members, then B_{L+1} can be small. If the scorer is less reliable, the cascade can allocate more intervals so that each residual filter covers a narrower score band, allowing better tailoring of filter sizes to the local

Table 6. Distributed key value store throughput scaling with shards.

Shard count	Baseline throughput (million qps)	Learned cascade (million qps)	Relative gain (%)
4	1.8	2.2	22.2
8	3.4	4.3	26.5
16	6.1	8.0	31.1
32	11.7	15.9	35.9
64	21.5	30.4	41.4

Table 7. Impact of query selectivity on cascade effectiveness.

Selectivity class	Baseline FPR (%)	Cascade FPR (%)	Relative reduction (%)
Very low (0.01)	0.95	0.10	89.5
Low (0.05)	1.10	0.15	86.4
Medium (0.10)	1.25	0.25	80.0
High (0.50)	1.60	0.60	62.5
Very high (1.00)	2.10	1.10	47.6

density of members and non-members [19]. The key principle is that the false positives contributed by each interval are weighted by the probability that a non-member key falls into that interval under the query distribution. If the query distribution concentrates non-members in certain regions, the cascade can allocate additional memory there to reduce false positives most effectively.

Training the scorer f_θ depends on the available features. In many key–value stores, the key itself may be a hash or an opaque identifier, in which case little structure is available. However, in many practical deployments, keys contain prefixes, namespaces, tenant identifiers, or structured encodings [20]. Additionally, shard-local metadata such as key length, prefix, or creation time bucket may correlate with membership in the current resident set. The feature map $\phi(k)$ can include such derived attributes, along with learned embeddings for categorical components. The scorer can be a linear model, a gradient-boosted tree, or a compact neural network, but its choice is constrained by latency and the need for stable calibration. A high-throughput shard front door often requires microsecond-scale scoring, favoring models that are fast and can be vectorized [21].

The scorer is trained to separate members and non-members, but the cascade objective is not standard classification accuracy. The downstream cost of errors is asymmetric: false negatives may be disallowed, while false positives incur a backend lookup cost that depends on the storage tier. Furthermore, the cascade will impose thresholds that operate in the extreme tail of the non-member score distribution when targeting very small false-positive rates. Therefore, it is beneficial to train the scorer with a loss that emphasizes ranking and calibration in the high-score tail. One approach is to train with a weighted logistic loss where non-member examples with high predicted scores are weighted more heavily [22]. Another approach is to optimize for an approximation of the area under the precision–recall curve in the relevant operating region. A practical method is to train a model to produce a well-calibrated estimate of $\Pr[Y = 1 \mid \phi(k)]$, and then select thresholds by directly measuring tail probabilities on a held-out calibration set.

Calibration is critical because the cascade uses thresholds to allocate members to residual filters. Even if the model is not perfectly probabilistic, it must produce a score whose empirical distributions for members and non-members are stable enough that the measured quantiles remain meaningful in deployment. Calibration can be achieved by post-hoc methods such as temperature scaling or isotonic regression applied to the raw model outputs, using a calibration dataset that reflects the deployed query distribution as closely as possible [23]. In a distributed store, the query distribution may be difficult to sample because non-members are not stored. However, the system naturally observes non-member queries during operation, and can log their scores to build an empirical non-member score distribution. For member keys, the system can sample from S_t or from positive queries. The calibration process then aligns the score scale so that thresholds chosen to achieve a desired non-member tail probability remain stable over time [24].

The cascade thresholds $\{\tau_\ell\}$ and filter sizes $\{m_\ell\}$ (bits) for each residual filter B_ℓ must be chosen jointly. A residual Bloom filter that stores $n_\ell = |S_\ell|$ keys with m_ℓ bits and h_ℓ hash functions has an approximate false-positive probability $\varepsilon_\ell \approx (1 - e^{-h_\ell n_\ell / m_\ell})^{h_\ell}$ under standard assumptions. For a fixed m_ℓ and n_ℓ, the optimal h_ℓ is approximately $(m_\ell / n_\ell) \ln 2$, yielding $\varepsilon_\ell \approx (0.6185)^{m_\ell / n_\ell}$. In practice, implementations may use blocked Bloom filters or SIMD-friendly variants that slightly alter the formula, but the dependence on bits per key remains similar, and the cascade framework can incorporate the

Table 8. Ablation on the number of cascade levels for a fixed memory budget.

Cascade levels	Index memory (megabytes)	FPR (%)	Throughput (thousands qps)
1	420	1.30	540
2	420	0.70	610
3	420	0.35	670
4	420	0.18	700
5	420	0.10	710

Table 9. Resource usage breakdown for different indexing strategies.

Method	CPU time per lookup (nanoseconds)	Cache miss rate (%)	Index memory
Classic Bloom filter	180	7.8	512
Cuckoo filter	165	7.1	480
Blocked Bloom filter	150	6.0	430
Learned Bloom filter	140	5.2	360
Learned cascade (proposed)	128	4.5	340

precise implementation-specific false-positive curve.

A subtlety arises because the residual filters are not consulted for all queries, only for those whose score lies in the corresponding interval. Therefore, the contribution of B_ℓ to the end-to-end false-positive probability is the product of the probability that a non-member query lands in interval I_ℓ and the filter's conditional false-positive probability. This suggests that filter bits should be allocated in proportion to how frequently non-members land in the interval and how many members must be stored in that interval. The cascade can exploit this by using more bits per key in intervals that receive more non-member queries, because reducing false positives there yields larger global benefit [25]. Conversely, intervals that are rarely visited by non-members can use fewer bits per key without violating the global false-positive budget.

Training also includes selecting the partitioning of the score axis. One method is to choose thresholds so that each interval contains an equal number of member keys, which equalizes n_ℓ and simplifies sizing. Another method is to choose thresholds so that each interval contains an equal probability mass of non-member queries, which equalizes the weight of each interval in the global false-positive rate. For high-selectivity lookups, non-member mass is often much larger than member mass, and it can be advantageous to partition based on non-member mass to better control the dominant false-positive contribution [26]. A hybrid strategy selects thresholds using both member and non-member histograms so that intervals isolate regions where member density is high relative to non-member density, enabling small residual filters with strong effect.

Because the store evolves, the cascade must support updates. The residual filters are defined in terms of membership in S_t and score interval membership. When S_t changes, keys are inserted and deleted, and their scores may drift if the scorer depends on time-varying features [27]. If the residual filters are standard Bloom filters, deletions are not supported; if deletions are required, the system can use counting Bloom filters, cuckoo filters, or stable Bloom variants, or it can rebuild filters periodically. The cascade framework is compatible with any approximate membership filter that offers a tunable false-positive curve and an update model, but the error bounds must reflect the chosen filter type. For simplicity, the subsequent analysis will treat the residual filters as Bloom-like filters with a known upper bound on false positives, and it will incorporate rebuild windows via an operational staleness budget.

4. Tight Error Bounds and Resource Optimization

The central requirement for high-selectivity deployment is an explicit bound on the cascade's end-to-end false-positive probability for non-member queries, and an explicit bound on false negatives for member queries under the chosen correctness policy. The cascade is designed so that these bounds can be computed from measurable quantities: score interval probabilities and filter parameters [28]. Tightness is achieved by avoiding unnecessary union bounds when independence structure is available, and by conditioning on the score interval selection, which is deterministic given the score.

Let $A(k)$ denote the event that the cascade returns accept for key k. For a non-member query $k \notin S_t$, a false positive occurs when $A(k) = 1$. For a member query $k \in S_t$, a false negative occurs when $A(k) = 0$ if the system treats rejection as definitive. In a key–value store that always performs a backend lookup upon accept, the cascade's output controls whether the store proceeds with a lookup [29]. Therefore, the relevant non-member false-

positive rate is $\Pr[A(k) = 1 \mid Y(k) = 0]$ under the query distribution conditioned on non-members, and the relevant member false-negative rate is $\Pr[A(k) = 0 \mid Y(k) = 1]$ under the query distribution conditioned on members.

Under the interval-based cascade described earlier, for any key k the interval index $\ell(k)$ is a deterministic function of the score. For non-members, define $p_\ell = \Pr[\ell(k) = \ell \mid Y(k) = 0]$ for $\ell \in \{0, 1, \ldots, L+1\}$. Define $q_\ell = \Pr[\ell(k) = \ell \mid Y(k) = 1]$ similarly for members. The interval 0 corresponds to immediate acceptance, and intervals 1 through L correspond to residual filters B_ℓ [30]. The tail interval $L+1$ corresponds to either immediate rejection or a guard.

Assume first the no-false-negative configuration in which members in every interval are accepted by construction, either via residual filters that contain all members in their interval or via an exact guard for the tail. In this case, the algorithmic false-negative probability is zero at build time, because for any member $k \in S_t$, either $s \in I_0$ and the cascade accepts immediately, or $s \in I_\ell$ and the key is included in B_ℓ so the filter returns positive with probability 1 under idealization, or $s \in I_{L+1}$ and the guard accepts. In practice, approximate filters can have false negatives due to implementation errors, hash collisions in certain structures, or staleness. These are treated as operational risks and can be mitigated by using Bloom filters for residuals and ensuring update consistency. The analysis therefore focuses on bounding false positives [31].

For a non-member k, if $\ell(k) = 0$ then the cascade accepts, contributing probability p_0. If $\ell(k) = \ell$ for $1 \le \ell \le L$, then the cascade accepts if and only if B_ℓ returns positive on k. Let ε_ℓ denote an upper bound on the false-positive probability of B_ℓ for keys not in its stored set, under the hash assumptions and any implementation-specific constraints. Conditioned on $\ell(k) = \ell$ and $Y(k) = 0$, the probability that B_ℓ returns positive is at most ε_ℓ. Therefore the cascade false-positive probability satisfies [32]

$$\begin{aligned}\Pr[A(k) = 1 \mid Y(k) = 0] &= \Pr[\ell(k) = 0 \mid Y(k) = 0] \\ &\quad + \sum_{\ell=1}^{L} \Pr[\ell(k) = \ell \mid Y(k) = 0] \Pr[B_\ell(k) = 1 \mid \ell(k) = \ell, Y(k) = 0] \\ &\quad + \Pr[\ell(k) = L+1 \mid Y(k) = 0] \Pr[\text{tail accepts} \mid \ell(k) = L+1, Y(k) = 0] \\ &\le p_0 + \sum_{\ell=1}^{L} p_\ell \varepsilon_\ell + p_{L+1} \alpha, \quad (1)\end{aligned}$$

where α is the acceptance probability in the tail for non-members, determined by the guard policy. If the tail is rejected immediately, then $\alpha = 0$, reducing false positives but introducing false negatives for members in the tail. If the tail is always accepted to preserve no false negatives, then $\alpha = 1$, which may be unacceptable for false positives unless the tail probability p_{L+1} is negligible. In practice, a no-false-negative cascade sets α close to 0 by using a tail residual filter B_{L+1} with small false-positive probability, yielding $\alpha = \varepsilon_{L+1}$, or by using an exact guard with $\alpha = 0$ but a backend lookup cost. The analysis can incorporate either [33].

The expression above is already close to tight because it conditions on the interval event, which is deterministic given the score and does not require independence assumptions between the scorer and the residual filter hashes. The only slack arises from using an upper bound ε_ℓ rather than the exact false-positive rate conditioned on the query distribution restricted to the interval. Tightness can be improved by measuring the realized false-positive rate of each residual filter on a held-out non-member sample restricted to the interval. However, for tight and auditable bounds, it is useful to retain the worst-case ε_ℓ implied by filter parameters, and then separately maintain monitoring to validate that realized rates are below bound [34].

A key requirement is to enforce a global false-positive budget ε such that

$$p_0 + \sum_{\ell=1}^{L} p_\ell \varepsilon_\ell + p_{L+1} \alpha \le \varepsilon. \quad (2)$$

Given measured p_ℓ and chosen α, the task is to choose thresholds (which determine p_ℓ and the member counts n_ℓ) and filter sizes (which determine ε_ℓ) to satisfy the inequality while minimizing total memory and compute overhead. Since p_0 is purely a function of the scorer and τ_0, it can be controlled by moving τ_0 upward or downward. Increasing τ_0 reduces p_0 but increases the number of members that fall into residual intervals, potentially increasing total residual memory. This introduces the central trade-off in learned filters: shifting work from the scorer acceptance region into residual filters [35].

To incorporate memory, consider a Bloom-like residual filter with false-positive curve approximately $\varepsilon_\ell \approx \exp(-c\, m_\ell / n_\ell)$ for a constant $c > 0$ depending on the filter design, where m_ℓ is bits and n_ℓ is stored keys. For classic Bloom filters at optimal hashing, $c \approx (\ln 2)^2$. Using this approximation yields an optimization problem

$$\min_{\{\tau_\ell\}, \{m_\ell\}} \sum_{\ell=1}^{L} m_\ell + m_{\text{tail}} \quad (3)$$

$$\text{s.t.} \quad p_0(\tau_0) + \sum_{\ell=1}^{L} p_\ell(\{\tau\}) \exp\left(-c\, \frac{m_\ell}{n_\ell(\{\tau\})}\right) + p_{L+1}(\{\tau\})\, \alpha \le \varepsilon \quad (4)$$

$$m_\ell \ge 0 \quad \text{for all } \ell, \quad (5)$$

where m_{tail} and α represent the tail policy, and where p_ℓ and n_ℓ depend on the thresholds through the score distributions for non-members and members. This formulation highlights that the marginal benefit of allocating additional bits to interval ℓ is proportional to p_ℓ and to the current ε_ℓ, and inversely related to n_ℓ. For

fixed thresholds, the optimal allocation of m_ℓ to meet a constraint resembles a water-filling solution [36]. Using Lagrange multipliers, for intervals where $m_\ell > 0$ the optimality condition satisfies

$$p_\ell \exp\left(-c\,\frac{m_\ell}{n_\ell}\right)\frac{c}{n_\ell} = \lambda, \tag{6}$$

for some $\lambda > 0$ chosen to satisfy the global constraint with equality. Solving for m_ℓ gives

$$m_\ell = \frac{n_\ell}{c}\left(\ln\left(\frac{p_\ell c}{\lambda n_\ell}\right)\right), \tag{7}$$

with the understanding that intervals with $\frac{p_\ell c}{\lambda n_\ell} \le 1$ receive $m_\ell = 0$ because allocating bits there yields insufficient global benefit. This expression is useful operationally because it links bits-per-key in each interval to the ratio p_ℓ/n_ℓ, which can be interpreted as the non-member query mass per stored member key in that interval [37]. Intervals that attract many non-member queries relative to their stored members should receive more bits per key, because they are visited frequently and contain relatively few members, making them efficient to filter.

The thresholds influence both p_ℓ and n_ℓ, and their selection can be approached by discretizing the score into bins and searching for partitions that minimize memory under the bound. Because the score distribution can be estimated as histograms for members and non-members, the optimization can operate on these histograms. Let the score range be binned into B bins, and let u_b denote the fraction of non-members in bin b, and v_b denote the count of members in bin b. A partition into intervals corresponds to grouping consecutive bins [38]. For any interval ℓ consisting of bins $b \in \mathscr{B}_\ell$, we have $p_\ell = \sum_{b\in\mathscr{B}_\ell} u_b$ and $n_\ell = \sum_{b\in\mathscr{B}_\ell} v_b$. The acceptance region I_0 corresponds to bins above τ_0, contributing p_0. The residual intervals contribute $p_\ell \varepsilon_\ell$ to the bound. This discrete representation makes the optimization implementable and auditable: the store can maintain the histograms online and recompute thresholds and allocations when drift is detected.

The above bound is tight with respect to the cascade structure, but it assumes that ε_ℓ is independent of the restricted query distribution within interval ℓ. For Bloom-like filters, the false-positive rate is largely independent of the query distribution because it depends on hash outputs, provided keys are hashed uniformly [39]. In practice, keys may be structured, and hash functions may not perfectly randomize, especially if performance constraints lead to reduced hashing. To maintain tight bounds, the system can incorporate conservative hash families and validate uniformity by measuring bit occupancy and empirical false-positive rates on sampled non-members. Additionally, the cascade can be built over hashed representations of keys, using a high-quality hash as a preprocessing step, so that the residual filters see uniformly distributed inputs even if original keys are structured. This shifts the distributional assumptions away from the original key space and improves bound reliability [40].

The cascade may also permit a controlled false-negative probability if the application allows it, for example in caches where a missed member can be recovered by a slower path. In that case, the tail interval can be rejected without a guard, and the member false-negative probability becomes $q_{L+1} = \Pr[\ell(k) = L+1 \mid Y(k) = 1]$. The system can then impose a constraint $q_{L+1} \le \delta$ for a target δ, and the optimization includes both constraints:

$$p_0 + \sum_{\ell=1}^{L} p_\ell \varepsilon_\ell \le \varepsilon, \qquad q_{L+1} \le \delta. \tag{8}$$

This variant illustrates how the cascade framework supports explicit trade-offs rather than implicit assumptions. In strongly consistent key–value stores, δ is typically set to 0, implemented by ensuring that all members are accepted by some stage, but the formalism remains useful because it separates algorithmic and operational contributions to misses.

A final aspect of tightness concerns composition across shards. Suppose each shard m has its own cascade with false-positive bound $\varepsilon^{(m)}$ and receives a fraction w_m of non-member queries. Then the global false-positive rate is bounded by $\sum_m w_m \varepsilon^{(m)}$. If the system enforces a global budget $\varepsilon_{\text{global}}$, it can allocate per-shard budgets such that $\sum_m w_m \varepsilon^{(m)} \le \varepsilon_{\text{global}}$. This allocation can itself be optimized, giving more budget to shards where achieving a small $\varepsilon^{(m)}$ is expensive in memory due to weaker separability, and less budget to shards where separability is strong. Because w_m can be measured online, the system can adjust budgets and thresholds without retraining the global scorer, preserving operational flexibility while maintaining a quantitative global bound [41].

5. Distributed Integration and Operational Considerations

Deploying a learned Bloom filter cascade in a distributed key–value store requires integrating model scoring, residual filters, calibration, and update handling into an existing lookup pipeline without harming tail latency or correctness. The cascade must also be observable: it should export metrics that validate the bound assumptions and detect drift. This section discusses how to implement shard-local cascades, manage updates, and maintain tight bounds under realistic constraints.

The primary deployment point for a high-selectivity cascade is at the shard front door, before expensive index operations. Upon receiving a request, the shard computes the feature map $\phi(k)$ and evaluates the scorer $f_\theta(\phi(k))$ to obtain a score s [42]. The feature map must

be fast to compute and stable across software versions. In many stores, the key is a byte string; feature extraction may include prefix hashes, length, namespace identifiers, and tenant tags that are available from request context. A critical design constraint is that the scorer must be deterministic and consistent across replicas, so that the same key yields the same score regardless of which replica serves the request. Determinism is important for debugging and for ensuring that residual filter membership is well-defined [43]. Therefore, any features derived from mutable state must be carefully controlled, and any non-deterministic operations such as floating-point reductions must be made consistent via fixed precision or stable kernels.

The residual filters B_ℓ are shard-local data structures. They can be stored in memory and updated via rebuilds or incremental updates. A practical approach is to rebuild the residual filters periodically from the authoritative set $S_t^{(m)}$, using the current model version and thresholds. Rebuild frequency depends on update rate and on the acceptable staleness risk. If the store supports compaction or segment creation, residual filters can be built per segment and combined logically; however, for the cascade defined by score intervals, the natural unit is the shard's full set because the interval assignment depends on the model score [44]. Still, segment-level builds can be achieved by storing, for each segment, residual filters restricted to the segment's keys in each interval, and combining them by querying the appropriate segment filters. This increases query overhead because multiple segments may need to be checked, so shard-level filters are preferable when the primary goal is front-door rejection.

To maintain no false negatives, updates must ensure that a newly inserted key is not rejected due to stale filters. One operational strategy is to treat the cascade as an optimization hint rather than a definitive decision for writes or for reads that must be correct. For a read, if the cascade rejects, the system may still perform a cheap verification in a small exact index, or it may route the request to a replica that holds a more up-to-date filter [45]. Another strategy is to update the residual filters synchronously on writes, at least for the intervals that could reject the key. Because the cascade is designed to reject negatives aggressively, synchronous updates can be limited to a small structure: the store can maintain a write buffer of recently inserted keys that are always accepted, either by placing them in an always-accept cache or by inserting them into a small auxiliary filter that is checked before rejection. This auxiliary filter can be exact or approximate but must avoid false negatives for recently inserted keys. Periodic rebuild then incorporates these keys into the main residual filters and clears the buffer [46].

Deletions are more challenging because Bloom filters do not support them. In a key–value store, deletions occur due to explicit deletes, TTL expiration, or compaction drops. If a residual filter retains deleted keys, it will produce additional false positives, increasing backend load but not violating correctness. Therefore, one acceptable policy is to allow residual filters to be deletion-stale and rebuild periodically to remove deleted keys. The bound must then incorporate the effect of stale keys on ε_ℓ [47]. For Bloom filters, adding extra keys increases bit occupancy and thus increases false-positive probability. If the store can bound the maximum number of stale keys between rebuilds, it can compute an upper bound on the false-positive rate by treating n_ℓ as the maximum possible stored count including stale keys. Because this is a worst-case bound, it may be conservative; however, it remains auditable. Alternatively, counting Bloom filters or cuckoo filters support deletions, reducing staleness effects but increasing memory and complexity. The cascade framework supports either choice by parameterizing each residual structure with an explicit false-positive bound function of its current load [48].

Model versioning is another operational constraint. If the scorer changes, the score intervals and thus the definition of S_ℓ change, requiring residual filters to be rebuilt for the new model. During a rolling upgrade, different replicas might run different model versions, leading to inconsistency. To avoid this, a shard can store the model version used to build the filters and reject using only that version. Requests can be routed to replicas with matching versions, or the shard can keep multiple cascades in parallel during transition [49]. Keeping parallel cascades increases memory but simplifies safety. A conservative approach is to maintain a baseline classical Bloom filter in parallel as a fallback during transitions, ensuring that the system can reject negatives safely even if the learned cascade is temporarily disabled.

Calibration drift is addressed by online monitoring of score distributions. The shard can maintain histograms of s for a sample of non-member queries and for observed member queries [50]. Because non-member queries are plentiful in high-selectivity regimes, the non-member histogram can be estimated accurately with modest sampling. For member keys, the histogram can be built from the stored set via periodic sampling, or from positive queries. The thresholds τ_ℓ can then be adjusted to maintain target p_0 and to control p_{L+1}. However, adjusting thresholds changes the membership assignment of keys to residual filters. Therefore, threshold adjustments without rebuilding filters can violate the no-false-negative guarantee if a member key moves into an interval whose filter does not contain it. To support online adjustment while preserving correctness, the cascade can restrict threshold changes to be monotone in a safe direction [51]. For example, increasing τ_0 moves keys from

immediate-accept into residual intervals, which is safe if the residual filters contain all members below the old τ_0. But the residual filters were built with a specific partition, so arbitrary threshold changes are unsafe. A practical solution is to decouple score partitioning used for residual filters from the acceptance rule by defining residual filters on fixed bins and allowing the acceptance threshold to move only at bin boundaries that preserve membership containment. If residual filters are built for a fine-grained binning, then thresholds can move across bins by activating or deactivating entire bins, which can be done safely if the corresponding residual filters exist.

Another solution is to use a two-layer logic: the scorer decides whether to accept immediately; if it would reject, the system consults a backup structure that contains all members that might be rejected under any allowed threshold [52]. This is equivalent to choosing a conservative lower threshold for backup inclusion and allowing the operational threshold to vary above it. Concretely, pick a build-time threshold τ_{build} such that all keys with $s < \tau_{\text{build}}$ are included in a backup filter. Then at runtime choose an operational threshold $\tau_{\text{op}} \geq \tau_{\text{build}}$. Any member with $s < \tau_{\text{op}}$ is also in the backup because $s < \tau_{\text{op}}$ implies $s < \tau_{\text{build}}$ is not necessarily true, so to guarantee containment one instead requires $\tau_{\text{op}} \leq \tau_{\text{build}}$. Therefore, to allow threshold increases, the backup must include members below the maximum possible operational threshold. This suggests building with a high threshold and allowing only decreases. In many systems, decreasing τ_0 increases immediate accepts and thus increases false positives but reduces residual load; this can be used as a safety valve under load, while preserving correctness [53]. The cascade can be designed so that the safe direction of threshold movement corresponds to graceful degradation under overload.

Tight error bounds require that the system can estimate p_ℓ, the probability that a non-member query falls into interval ℓ. In a distributed store, non-member queries are observed directly, so the shard can estimate p_ℓ by logging scores for rejected or accepted non-member queries. The challenge is determining whether a query is a non-member without performing a lookup, which would defeat the purpose. This can be handled by sampling: for a small fraction of queries, the store performs the full lookup regardless of the cascade decision and records whether the key was present [54]. This yields unbiased estimates of p_ℓ and of realized false positives. The sampling rate can be small because high throughput yields many samples, and the sampling overhead can be bounded. The system can then compute confidence intervals for p_ℓ and adjust the bound conservatively by using upper confidence bounds for p_0 and p_ℓ when enforcing an auditable guarantee.

Shard heterogeneity implies that a single global model may have different separability across shards. The cascade design accommodates this by allowing shard-local thresholds and residual allocations [55]. Each shard measures its own score histograms and chooses thresholds and filter sizes to meet a shard-local budget. If a global false-positive budget must be enforced, the system can allocate budgets dynamically based on observed query rates and on memory availability. This is important in multi-tenant environments where certain tenants may generate disproportionate negative queries. The cascade can incorporate tenant identifiers into features and can maintain per-tenant histograms, enabling tenant-specific thresholds or even tenant-specific residual filters [56]. However, per-tenant filters increase complexity and memory fragmentation, so a practical design aggregates tenants into classes based on observed score distributions.

Evaluation of the cascade in a distributed setting must consider both micro-level filter metrics and macro-level system metrics. Micro-level metrics include the per-interval member counts n_ℓ, the per-interval non-member probabilities p_ℓ, the configured ε_ℓ from filter parameters, and the measured empirical false-positive rates. Macro-level metrics include backend lookup reduction, CPU utilization, cache hit rate impact, and tail latency changes. For high-selectivity lookups, a small increase in false positives can have large cost, so evaluation must focus on the operating region corresponding to the target ε [57]. This often requires large-scale sampling or replay of production traces to estimate rare events. The tight bound framework supports this by enabling operators to reason in terms of measured p_ℓ and configured ε_ℓ, reducing reliance on observing extremely rare global false positives directly.

An important practical consideration is compute overhead. The learned scorer adds CPU cost compared to a pure Bloom filter, and residual filters add additional memory accesses. The cascade is beneficial if the cost of scoring and residual checks is less than the cost saved by avoiding backend lookups [58]. In high-selectivity regimes, backend lookups are expensive relative to scoring, but the scorer must still be highly optimized. Techniques include feature extraction with fixed offsets, model quantization, and vectorized evaluation. Additionally, the cascade can exploit early exits: if the score falls in the immediate accept region, no residual filter is consulted; if it falls in a region with a very small residual filter, the check may be cache-resident. The cascade design can also be aligned with cache locality by ordering intervals by expected query mass and placing their filters in contiguous memory [59].

Finally, fault tolerance requires that the cascade does not become a single point of failure. If the model or filters are unavailable, the shard must fall back to a safe behavior, typically performing normal lookups without filtering or using a baseline Bloom filter. Because the

cascade is an optimization, such fallback is acceptable, though it increases load. The system should therefore maintain the ability to disable the cascade quickly and to revert to a known-safe configuration. The tight bound framework aids this by making the cascade behavior explicit and by enabling operators to set conservative thresholds when uncertainty arises, preserving bounded false positives at the cost of additional backend work [60].

6. Conclusion

This paper presented a learned Bloom filter cascade tailored to high-selectivity lookups in distributed key–value stores, with an emphasis on tight, composable error bounds suitable for operational deployment. The cascade combines a calibrated learned scorer with a sequence of residual approximate filters over score intervals and, when required, a guard mechanism that enforces a no-false-negative policy. By conditioning the analysis on deterministic score intervals, the end-to-end false-positive probability decomposes into a weighted sum of per-interval contributions, enabling tight bounds that depend on measurable non-member interval probabilities and known filter false-positive parameters. This structure supports explicit budgeting, shard-local tuning, and global composition across heterogeneous shards.

The resource optimization perspective highlighted that memory should be allocated preferentially to intervals that attract substantial non-member query mass relative to their stored members, because these intervals dominate global false positives [61]. A histogram-based representation of score distributions provides an implementable pathway to selecting thresholds and allocating filter bits under a prescribed global false-positive budget. The distributed integration discussion addressed practical constraints including updates, deletions, model versioning, calibration drift, and observability, emphasizing mechanisms that preserve bounded behavior under drift and rolling upgrades while maintaining the ability to fall back safely.

Overall, the learned cascade framework is most applicable when the key distribution exhibits exploitable structure and when negative lookups dominate enough that avoiding backend work amortizes the additional scoring cost. The tight bound formulation provides a basis for auditable deployment decisions and for monitoring-driven recalibration. Future extensions within the same bounding framework include incorporating richer residual structures with deletions, adapting thresholds under drift using safe monotone adjustments, and optimizing jointly for memory and compute under explicit latency constraints in multi-tier storage pipelines [62].

References

[1] J. Jaskolka, “Evaluating the exploitability of implicit interactions in distributed systems.,” 6 2020.

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

[3] I. Siddique, “Libguides: Distributed systems: Home,” 7 2018.

[4] C. B. Jones, T. Haines, and R. Darbali-Zamora, “Hosting capacity considerations for the combination of wind and solar on distribution electric power systems subject to different levels of coincident operations,” Journal of Renewable and Sustainable Energy, vol. 17, 11 2025.

[5] J.-H. Syu, J. C.-W. Lin, P. Biernacki, and A. Ziebinski, “Machine learning-based calibration approaches for single-beam and multiple-beam distance sensors,” IEEE Sensors Journal, vol. 24, pp. 975–983, 1 2024.

[6] L. Mariani, M. Pezzè, O. Riganelli, and R. Xin, “Predicting failures in multi-tier distributed systems,” 11 2019.

[7] 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.

[8] F. A. Wolf and P. Müller, “Verifiable security policies for distributed systems,” in Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, pp. 4–18, ACM, 12 2024.

[9] “Demystifying distributed systems in cloud-native environments,” Journal of Computational Analysis and Applications, vol. 34, 10 2025.

[10] X. Wei, Z. Huang, T. Sun, Y. Hao, R. Chen, M. Han, J. Gu, and H. Chen, “Phoenixos: Concurrent os-level gpu checkpoint and restore with validated speculation,” in Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, pp. 996–1013, ACM, 10 2025.

[11] M. Krzysztoń, B. Bok, P. Żakieta, and J. Kołodziej, “Evasion attacks on ml in domains with nonlinear constraints,” in 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW), pp. 112–119, IEEE, 5 2024.

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

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

[14] C. Zhu, W. Zheng, X. Fan, X. Deng, S. Liu, L. Yi, W. Xi, and Y.-S. Jeong, "Graph-empowered multi-dimensional target full-coverage reliability for internet of everything," IEEE Internet of Things Journal, vol. 12, pp. 3707–3719, 2 2025.

[15] D. Marek, P. Biernacki, J. Szygula, and A. Domanski, "General concepts of a simulation method for automated guided vehicle in industry 4.0," in 2022 IEEE International Conference on Big Data (Big Data), pp. 6306–6314, IEEE, 12 2022.

[16] E. Becks, P. Zdankin, V. Matkovic, and T. Weis, "Complexity of smart home setups: A qualitative user study on smart home assistance and implications on technical requirements," Technologies, vol. 11, pp. 9–9, 1 2023.

[17] R. Alfonso, C. Daher, M. Arzamendia, K. Cikel, D. Gregor, D. Gutierrez, S. Toral, and M. Villagra, "A motorcyclist helmet detection system through a two-stage cnn approach," in 2021 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON), pp. 1–6, IEEE, 12 2021.

[18] R. K. Ghosh and H. Ghosh, "Global states and termination detection," 2 2023.

[19] A. Rashidov, A. Akhatov, I. Aminov, D. Mardonov, and A. Dagur, Distribution of data flows in distributed systems using hierarchical clustering, pp. 207–212. CRC Press, 6 2024.

[20] 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 Conferaence on Advanced Computing and Communications, pp. 628–629, IEEE, 2006.

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

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

[23] E. Becks, M. Josten, V. Matkovic, and T. Weis, "Revising poor man's eye tracker for crowd-sourced studies," in 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops), pp. 328–330, IEEE, 3 2023.

[24] K. Cui, S. Liu, W. Feng, X. Deng, L. Gao, M. Cheng, H. Lu, and L. T. Yang, "Correlation-aware cross-modal attention network for fashion compatibility modeling in ugc systems," ACM Transactions on Multimedia Computing, Communications, and Applications, vol. 21, pp. 1–24, 11 2025.

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

[26] "Reliable networked and distributed systems," 4 2024.

[27] 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.

[28] M. A. Helcig and S. Nastic, "Fedccl: Federated clustered continual learning framework for privacy-focused energy forecasting," in 2025 IEEE 9th International Conference on Fog and Edge Computing (ICFEC), pp. 50–57, IEEE, 5 2025.

[29] I. Argyroulis, "Recent advancements in distributed system communications," 1 2021.

[30] K. Agarwal, O. Khare, A. Sharma, A. Prakash, and A. K. Shukla, "Artificial intelligence in a distributed system of the future," 7 2024.

[31] R. S. Chowhan, Evolution and Paradigm Shift in Distributed System Architecture. IntechOpen, 4 2019.

[32] OSDI - Bringing Decentralized Search to Decentralized Services, 6 2021.

[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] J. Chen, P. Wang, S. Du, and W. Wang, "Log pattern mining for distributed system maintenance," Complexity, vol. 2020, pp. 1–12, 12 2020.

[35] S. Agarwal, M. A. Rodriguez, and R. Buyya, "Serv-drishti: An interactive serverless function request simulation engine and visualiser," in 2025 IEEE 35th International Telecommunication Networks and Applications Conference (ITNAC), pp. 1–7, IEEE, 11 2025.

[36] A. Heß, F. J. Hauck, and E. Meißner, "Consensus-agnostic state-machine replication," in Proceedings of the 25th International Middleware Conference, pp. 341–353, ACM, 12 2024.

[37] M. Zaccarini, F. Poltronieri, C. Stefanelli, and M. Tortonesi, "Hybridized hot restart via reinforcement learning for microservice orchestration," in NOMS 2025-2025 IEEE Network Operations and Management Symposium, pp. 1–7, IEEE, 5 2025.

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

[39] A. Samanta, C. Chetri, D. Karnehm, A. Neve, and S. Williamson, "Temporal sensitivity analysis of internal temperature informed charging algorithms and rapid thermal management system for e-mobility," in 2024 IEEE Energy Conversion Congress and Exposition (ECCE), pp. 2302–2304, IEEE, 10 2024.

[40] A. Lapkovskis, B. Sedlak, S. Magnússon, S. Dustdar, and P. K. Donta, "Benchmarking dynamic slo compliance in distributed computing continuum systems," in 2025 IEEE International Conference on Edge Computing and Communications (EDGE), pp. 93–102, IEEE, 7 2025.

[41] A. Ba, F. O'Donncha, J. Ploennigs, and M. Azmat, "Efficient extraction of insights at the edges of distributed systems," in 2023 IEEE International Conference on Big Data (BigData), pp. 1610–1619, IEEE, 12 2023.

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

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

[44] B. V. S. Pinto, D. R. Melo, C. A. Zeferino, E. A. Bezerra, and F. Viel, "Implementation of double sha-256 in hls for fpga using real bitcoin blocks," in 2025 17th Seminar on Power Electronics and Control (SEPOC), pp. 1–7, IEEE, 11 2025.

[45] A. Hennebelle, Q. Dieng, L. Ismail, and R. Buyya, "Smartedge: Smart healthcare end-to-end integrated edge and cloud computing system for diabetes prediction enabled by ensemble machine learning," in 2024 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 127–134, IEEE, 12 2024.

[46] 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.

[47] K. Mak, K. Osuka, and T. Wada, "Development of a multi-master communication platform for mobile distributed systems," Journal of Robotics and Mechatronics, vol. 31, pp. 348–354, 4 2019.

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

[49] 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.

[50] A. Hermann, N. Trkulja, P. Wachter, B. Erb, and F. Kargl, "Quantification methods for trust in cooperative driving," in 2025 IEEE Vehicular Networking Conference (VNC), pp. 1–8, IEEE, 6 2025.

[51] J. Yang, Z. Wang, R. Chen, and H. Chen, "A system-level abstraction and service for flourishing ai-powered applications," in Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems, pp. 106–114, ACM, 10 2025.

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

[53] P. Afanasev, A. Ilyushina, S. Kolesnichenko, P. Komissarov, and E. Zheleznov, "Environmental monitoring using distributed system theory," in SGEM International Multidisciplinary Scientific GeoConference EXPO Proceedings, vol. 21, pp. 247–254, STEF92 Technology, 12 2021.

[54] V. Mahaliyanaarachchi, "Security issues and mitigation mechanisms in distributed systems," in 2023 3rd International Conference on Advanced Research in Computing (ICARC), pp. 172–177, IEEE, 2 2023.

[55] M. Arzamendia, D. Britez, G. Recalde, V. Gomez, M. Santacruz, D. Gregor, D. Gutierrez, S. Toral, and F. Cuellar, "An autonomous surface vehicle for water quality measurements in a lake using mqtt protocol," in 2021 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON), pp. 1–5, IEEE, 12 2021.

[56] M. Mditshwa, M. E. S. Mnguni, and M. Ratshitanga, "The limitations of an automatic generation control in stabilizing power system in the event of load demand increase," in 2022 30th Southern African Universities Power Engineering Conference (SAUPEC), vol. 9, pp. 1–5, IEEE, 1 2022.

[57] M. Chen, M. T. Islam, M. A. Rodriguez, and R. Buyya, "Trade: Network and traffic-aware adaptive scheduling for microservices under dynamics," IEEE Transactions on Parallel and Distributed Systems, pp. 1–14, 1 2025.

[58] X. Zhao, Z. Lei, G. Zhang, Y. Zhang, and C. Xing, WISA - Blockchain and Distributed System, pp. 629–641. Germany: Springer International Publishing, 9 2020.

[59] 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.

[60] A. Gogineni, "Resource management strategies in heterogeneous distributed systems," Journal of Artificial Intelligence, Machine Learning and Data Science, vol. 2, pp. 2183–2189, 7 2024.

[61] S. E, "Logical and vector clocks used in distributed systems," Journal of emerging technologies and innovative research, vol. 8, 5 2021.

[62] K. K. Rout, D. P. Mishra, and S. R. Salkuti, "Deadlock detection in distributed system," Indonesian Journal of Electrical Engineering and Computer Science, vol. 24, pp. 1596–1603, 12 2021.