OPEN ACCESS

A High-Throughput Retrieval-Augmented Generation Backend with Consistent Index Updates and Cache Coherency for Marketplace Product Search

Bagas Wiratama,[1] Fadhil Setiawan[2]

[1] Department of Computer Science, Institut Teknologi Cakrawala, Jalan Cendana Raya No. 18, Bandung, Jawa Barat, Indonesia;
[2] Department of Computer Science, Universitas Mandala Nusantara, Jalan Melati Utama No. 42, Yogyakarta, Daerah Istimewa Yogyakarta, Indonesia;

Abstract

Marketplace product search increasingly blends traditional information retrieval with neural retrieval and large language model generation to support richer query understanding, multi-attribute matching, and response synthesis. In operational settings, however, the retrieval-augmented generation (RAG) backend must handle high query throughput while continuously ingesting catalog changes such as price updates, inventory fluctuations, and policy-driven content edits. These changes complicate index maintenance because retrieval quality and user-facing correctness depend on coherent visibility across dense vector indexes, sparse lexical indexes, feature stores, and caches. This paper describes a high-throughput RAG backend design for marketplace product search that emphasizes consistent index updates and cache coherency under frequent, fine-grained catalog mutations. The architecture combines event-sourced ingestion, deterministic embedding generation, hybrid retrieval with shard-local fusion, and a read path that is explicitly versioned by index epochs. Consistency is treated as a systems property rather than an application afterthought: updates are sequenced through a commit log, materialized into multi-index snapshots using write-ahead metadata, and made visible via atomic epoch advances that align caches, rerank features, and retrieval results. Cache coherency is implemented using generational keys, epoch-tagged entries, and selective invalidation driven by product-level dependency tracking. Performance is analyzed through queueing-based capacity models and staleness bounds derived from update propagation delays. The discussion focuses on practical tradeoffs among latency, freshness, and operational complexity in a production marketplace setting.

1. Introduction

Marketplace search differs from many document retrieval workloads because the corpus is dominated by short, structured items whose meaning depends on rapidly changing attributes [1]. A product listing is not only a textual description but also a bundle of constraints such as price, currency, shipping region, inventory status, seller reputation, compliance flags, and taxonomy placement. Many of these constraints change frequently, and user expectations for correctness tend to be strict for certain attributes. If an item is out of stock or cannot ship to a user's region, retrieving and presenting it as a primary candidate can be perceived as an error even if the textual match is strong. The introduction of retrieval-augmented generation amplifies these challenges because the generated response, whether it is a conversational answer, a synthesized comparison, or an explanation of why an item matches, is only as faithful as the retrieved context. When the retrieval layer is stale or inconsistent across indexes, generation can appear confidently incorrect because it composes statements from partial or conflicting views of the catalog [2].

At the same time, the backend must satisfy stringent latency and throughput constraints. Product search is frequently placed on the critical path of user interaction, and the tail latency distribution matters as much as the median. The RAG pipeline adds computational stages: dense embedding of the query, approximate nearest neighbor retrieval, lexical retrieval, candidate fusion, learned

CTASC Copyright by the Authors.
Licensed as an open access article using a CC BY 4.0 license.

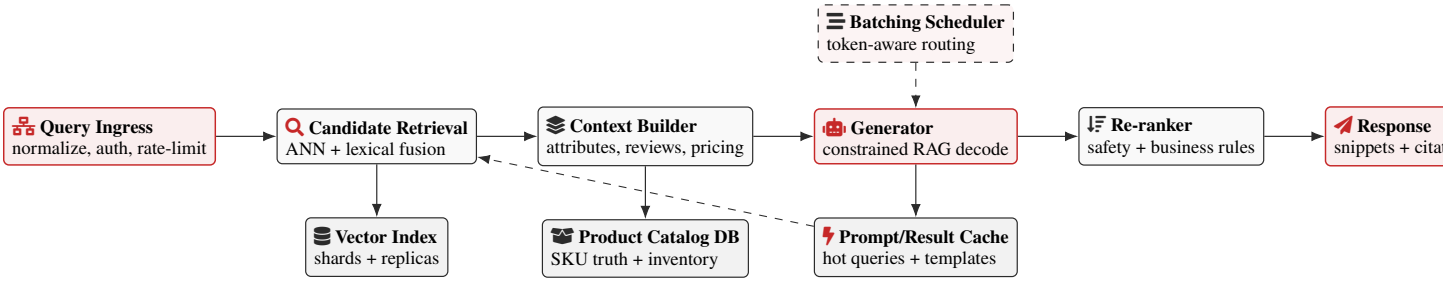


Figure 1: High-throughput RAG backend for marketplace product search: a tight online path (ingress → retrieval → context → generation → ranking) with separate stores for the vector index, catalog truth, and hot-path caching.

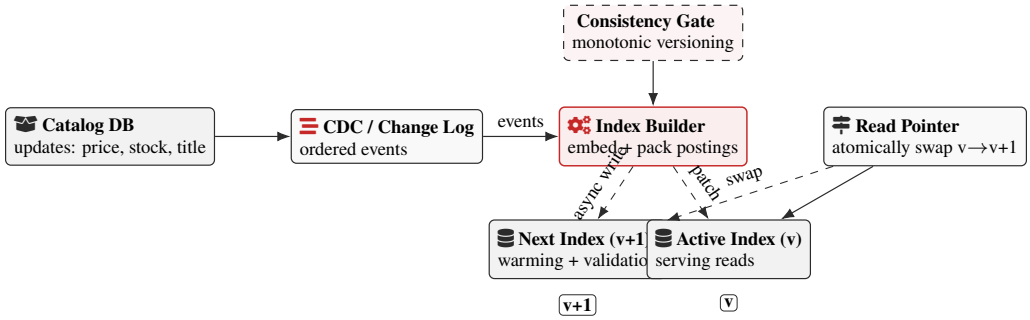


Figure 2: Consistent index updates via ordered change capture, versioned build artifacts, and an atomic read-pointer swap: reads remain on a stable active version while the next version is warmed and validated.

reranking, context packaging, and generation. Each stage consumes time and introduces opportunities for caching. Caching is often the difference between feasible and infeasible cost profiles at scale, but caching can also become a source of inconsistency when catalog mutations are frequent [3]. If retrieval results are cached without regard to index version, then a query can return candidates that no longer satisfy constraints. If generated answers are cached aggressively, then answers can reference items whose price has changed or whose availability has shifted. In a marketplace, these kinds of inconsistencies are not merely cosmetic; they can degrade trust and may create policy and compliance concerns depending on the domain.

A common response is to relax freshness and accept eventual consistency, but the operational question is not whether eventual consistency exists, but how it manifests. If different subsystems converge at different rates, then the user experience can oscillate between states in which some attributes are fresh and others are stale [4]. For example, a sparse index might be updated quickly with inventory changes, while a dense index lags because embeddings are regenerated asynchronously. A reranker feature store might contain the latest seller metrics, while cached retrieval sets still reflect an older taxonomy mapping. Such partial freshness can be worse than uniformly stale data because it produces contradictions that are hard to reason about and hard to debug. A robust backend therefore needs a principled notion of visibility and a mechanism to align read-time components with a coherent version of the corpus.

This paper presents a backend design that treats the RAG system as a versioned dataflow. Catalog changes enter through a commit log that defines a total order per product and a global progression through watermarks [5]. Derived artifacts, including lexical postings updates, vector index updates, feature store updates, and cache updates, are materialized as functions of the log. Read-time components then select a consistent snapshot, expressed as an index epoch, and the system enforces that all retrieval and generation decisions are tied to that epoch. The goal is not to enforce strict serializability across all products and all reads, which would be costly and unnecessary for search, but to enforce internal coherency such that a single request does not mix incompatible versions of the same product across stages. This coherency is also used to manage caching: caches are not simply key-value stores but become version-aware layers that can safely reuse work within a defined epoch boundary.

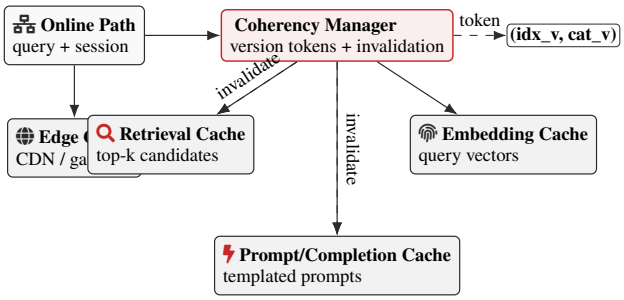


Figure 3: Cache coherency with lightweight version tokens (index and catalog versions) and targeted invalidation: hot-path caches remain fast while stale entries are rejected or evicted without global flushes.

Table 1: Core components of the high-throughput RAG backend

Component	Role in Pipeline	Implementation Highlights
Query Router	Routes search and generative queries to appropriate shards and models	Consistent hashing with latency-aware load balancing
Dual Index (BM25 + Vector)	Provides lexical and semantic retrieval for marketplace products	Sharded inverted index; HNSW-based ANN vector index
RAG Orchestrator	Coordinates retrieval, ranking, and generation for answers	Asynchronous fan-out with deadline-aware cancellation
Cache Layer	Caches retrieved contexts and generation results	Hierarchical key-value cache with versioned entries
Update Stream Processor	Applies product catalog changes to indexes and caches	Exactly-once processing using ordered change streams

The specific focus is on marketplace product search, which motivates several design choices [6]. Hybrid retrieval is typically required because lexical retrieval handles rare terms, codes, and exact attribute matches, while dense retrieval captures semantic similarity, paraphrases, and attribute inference. The catalog is often large, and indexing is distributed across shards for both sparse and dense representations. Updates are frequent and skewed: a small subset of products may receive many updates due to promotions, while the long tail changes rarely. The backend must therefore handle high update rates without forcing constant full index rebuilds. Approximate nearest neighbor structures such as HNSW or IVF-like schemes can support incremental updates, but their behavior under heavy churn must be managed to avoid recall degradation and fragmentation [7]. Similarly, sparse indexes often use segment-based approaches and merge policies that must be tuned to avoid write amplification.

A further complication is that RAG introduces a second-order dependency: the retrieval set influences generation, and generation can be cached independently. If a cached generation response is served after the retrieval set is no longer valid, the system can present a narrative that does not match the current catalog. One approach is to disable caching of generated outputs, but this increases cost and latency. Another approach is to cache with strict invalidation, but invalidation is hard when the dependency graph is large and dynamic. The approach here is to treat generation outputs as epoch-tagged artifacts, and to use dependency-aware invalidation for a narrow subset of changes that are known to affect output correctness materially, while relying on epoch advances to bound staleness [8].

The remainder of the paper is organized as a sequence of system concerns rather than a purely algorithmic narrative. The architecture section describes the end-to-end dataflow from ingestion to serving, emphasizing how epochs are produced and consumed. The data and indexing section describes hybrid retrieval, metadata filtering, and the representation choices that enable incre-

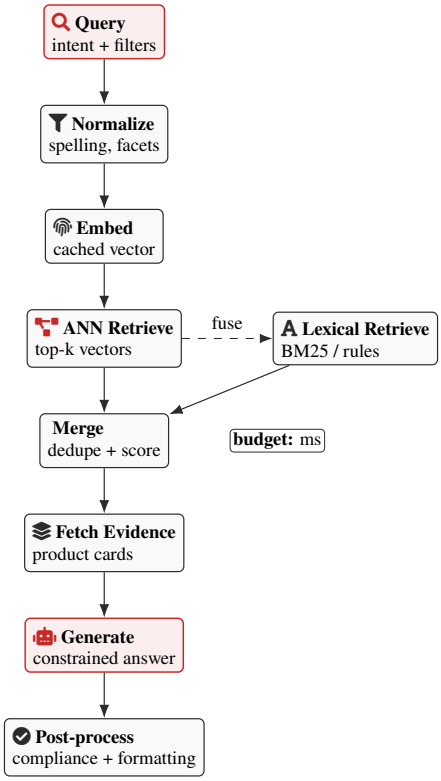


Figure 4: Online RAG execution pipeline for product search: normalized intent is embedded, retrieved through ANN and lexical channels, merged into evidence cards, then used for constrained generation and safe post-processing.

mental updates. The consistent index update section explains how the system maintains multi-index coherency using log sequencing, write-ahead metadata, and atomic epoch publication, including approaches for handling embedding regeneration and backfills. The cache coherency section details versioned caching, selective invalidation, and mechanisms to prevent thundering herds during epoch transitions [9]. The performance modeling and evaluation section presents analytical models that connect throughput, latency, cache hit rate, and staleness, and it discusses practical measurement strategies. The conclusion summarizes tradeoffs and outlines directions for refinement within the constraints of production marketplaces.

2. System Architecture

The backend is structured as two coupled planes: a write plane that ingests and materializes catalog state into multiple retrieval and ranking artifacts, and a read plane that serves queries using a consistent snapshot of those artifacts. The coupling is mediated by an epoch manager, which is responsible for publishing a monotonic sequence of epochs, each corresponding to a set of artifact versions that are mutually compatible. An epoch is not necessarily a single global snapshot in the database sense, because search infrastructure is distributed and often partitioned by shard, but the epoch defines a contract: for any request served at epoch e, all components that contribute to retrieval, ranking, filtering, and generation must either use data produced for epoch e or data that is guaranteed to be backward-compatible with it under explicitly defined rules [10].

On the write plane, catalog events are captured from the source of truth, typically via change data capture from transactional systems or via an application-level event stream. Each event is normalized into a canonical product mutation record that includes a product identifier, an event timestamp, a source sequence number when available, and a set of changed fields. The event stream is appended to a commit log with partitioning by product identifier to preserve per-product ordering. The log may be implemented via a distributed messaging system, but the key property is that consumers can deterministically replay events and that offsets can be checkpointed. The system includes multiple consumers: a materializer that constructs the current product state, an embedding pipeline that produces dense

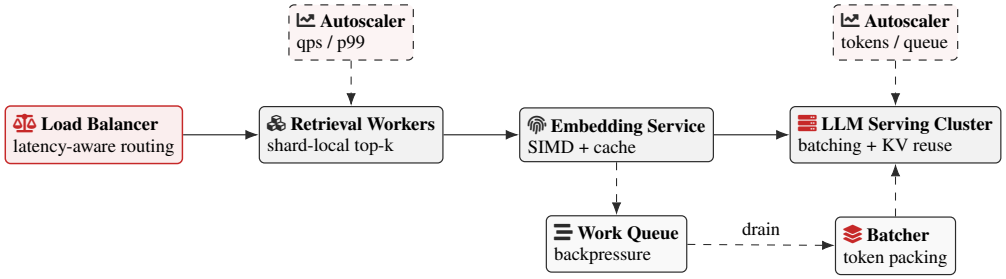


Figure 5: Throughput-oriented scaling: shard-local retrieval workers feed an embedding service and a batched LLM cluster, with queue-based backpressure and token-aware autoscaling to protect p99 latency.

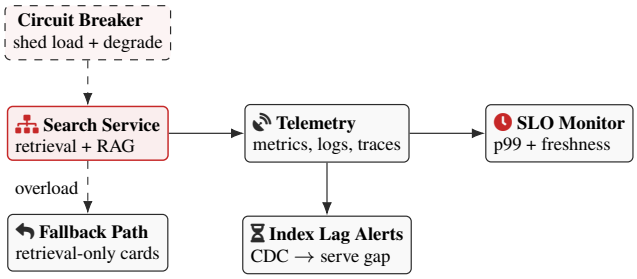


Figure 6: Operational safety loop: telemetry drives SLO monitoring and index-lag alerts, while a circuit breaker enables graceful degradation to retrieval-only results during overload or cache/index inconsistencies.

vectors and textual features, a lexical indexer that updates sparse postings, a feature pipeline that updates ranking features, and a cache invalidation pipeline that produces signals for selective invalidation [11].

Materialization proceeds in two stages. The first stage maintains a compacted product state store that is keyed by product identifier and holds the latest authoritative representation required for retrieval and ranking. This store is updated by applying mutations from the commit log in order, resolving conflicts using source ordering and explicit precedence rules. The store is written with a version counter per product, which increments on each applied mutation. The second stage produces derived artifacts. For lexical indexing, the derived artifact includes tokenization outputs, attribute normalization, and postings updates; for dense indexing, it includes embeddings derived from a chosen model and preprocessing pipeline; for features, it includes computed statistics and join results with seller and behavioral data [12]. Importantly, each derived artifact is annotated with the product version and the log offset that produced it, enabling downstream components to reason about freshness and to detect gaps.

The epoch manager consumes watermark information from the artifact pipelines. Each pipeline reports the highest log offset for which it has fully materialized artifacts into a serving-ready form. Serving-ready here means that the data has been written to the appropriate storage and that the storage has acknowledged durability. The epoch manager then computes an epoch candidate as the minimum of these offsets across pipelines, potentially after applying policy constraints [13]. For example, the system may require that lexical and dense indexes are within a bounded offset distance of each other, or it may allow the dense index to lag slightly if query-time fusion can tolerate it for certain query classes. The epoch manager publishes an epoch record that includes the epoch identifier, the log offset boundary, and pointers to the specific index segments or shard versions that correspond to that boundary. Publication is performed atomically, typically by writing to a strongly consistent metadata store. Read nodes watch this metadata and update their local epoch view.

The read plane consists of a query gateway, retrieval services, reranking services, and a generation service [14]. The query gateway performs authentication, request shaping, and admission control. It attaches an epoch to each request, either by using the latest published epoch or by selecting an older epoch for canarying or for graceful degradation when some components are recovering. The retrieval services execute hybrid retrieval. A sparse retriever queries a lexical index and applies attribute filters, while a dense retriever computes a query embedding and performs

Table 2: Marketplace datasets used for evaluation

Dataset	#Queries (M)	#Products (M)	Avg. Description Length
Electronics	12.4	38.1	84 tokens
Home & Kitchen	9.7	25.3	71 tokens
Fashion	15.2	44.8	63 tokens
Long-tail Niche	3.1	6.5	129 tokens
All Categories	40.4	114.7	78 tokens (weighted)

Table 3: Latency breakdown of the product search and RAG stack

Stage	Baseline p95 (ms)	Proposed p95 (ms)	Relative Improvement
Query Parsing & Routing	12.4	9.1	26.6%
Index Lookup (BM25)	41.0	29.7	27.6%
Vector Retrieval	63.5	38.2	39.8%
Reranking	54.3	47.9	11.8%
Generation (RAG)	188.6	162.3	13.9%
End-to-end (Search+RAG)	359.8	282.1	21.6%

approximate nearest neighbor search against a vector index. The candidate sets are fused, often by score normalization and rank aggregation, and passed to the reranker. The reranker queries a feature store and applies a learned model to produce a final ranking [15]. The generation service selects a subset of top candidates, constructs a context with product attributes and snippets, and invokes an LLM to produce a structured output, such as a conversational explanation or a comparison. The generation stage can be optional depending on the product surface, and in some flows the system returns ranked products without generation, but the backend design supports both.

A key architectural decision is that the epoch is treated as a first-class parameter across all service calls. Retrieval calls include the epoch so that the sparse and dense retrievers can select the correct index segments and so that caches can be partitioned by epoch. Feature store reads include the epoch so that feature values align with the same product versions used in retrieval [16]. Generation calls include the epoch so that any cached generation outputs can be validated. This approach shifts complexity from ad hoc invalidation to systematic versioning. The tradeoff is that every storage and cache layer must understand epochs, but the resulting behavior is easier to reason about and easier to debug because inconsistent mixes of versions become detectable.

To support high throughput, the architecture uses horizontal scaling, shard-local execution, and batching. Query embedding computation is batched at the dense retriever, which exposes a micro-batching interface and uses GPU or specialized inference accelerators when available [17]. Approximate nearest neighbor search is executed in shard-parallel fashion, and results are merged at the caller. Lexical retrieval is similarly shard-parallel. Reranking is often the most CPU-intensive stage if the model is complex; therefore, it is implemented as a stateless service with vectorized feature extraction and model inference. Generation is typically the most expensive stage; therefore, it is rate-limited and often gated by intent classification so that only queries that benefit from synthesis trigger generation.

Observability is integrated into the architecture with per-epoch metrics. The system logs the epoch used for each request, along with retrieval statistics such as candidate counts and score distributions, cache hit rates per layer, and generation usage [18]. The write plane logs epoch publication frequency and lag metrics per pipeline. These logs enable correlation between user-visible issues and specific epoch transitions. When an issue is detected, operators can compare behavior across epochs, roll back to an earlier epoch, or temporarily decouple certain pipelines by adjusting epoch policy, such as allowing the dense in-

Table 4: Throughput scaling with sharding and parallelism

Configuration	Shards	Peak QPS (Baseline)	Peak QPS (Proposed)
Single-shard	1	1.1k	2.0k
Shard-per-category	8	5.4k	10.2k
Fine-grained sharding	32	8.7k	17.5k
Fine-grained + RAG caching	32	9.3k	21.8k
Global autoscaling	64	10.1k	25.4k

Table 5: Cache strategy comparison for RAG contexts and generations

Cache Strategy	Hit Rate	p95 Latency Reduction	Avg. Invalidation Lag
No Cache	0.0%	–	–
Query-keyed LRU	42.7%	15.3%	18.2 s
Context-keyed LFU	58.9%	21.4%	11.7 s
Versioned Product Cache	64.3%	24.8%	2.9 s
Hybrid Versioned Cache	71.5%	28.6%	3.1 s

dex to lag within a threshold while disabling generation caching.

The architecture also accounts for multi-region deployments. In a global marketplace, serving often occurs in multiple regions, and write pipelines may be centralized or regionally replicated [19]. The epoch manager can operate per region, publishing region-specific epochs, or it can publish a global epoch that all regions follow. A global epoch simplifies coherency but can amplify the impact of a slow region; a regional epoch reduces coupling but introduces cross-region divergence. The design presented here supports regional epochs with a shared commit log replicated across regions, enabling each region to advance its epoch according to local materialization status while maintaining per-product ordering via partitioned replication. Cross-region consistency is then a matter of bounding divergence in epoch offsets and ensuring that user flows that cross regions, such as seller dashboards, do not rely on the same coherency contract as search.

3. Data Representation and Hybrid Indexing

Marketplace product data is heterogeneous, blending free-form text with structured attributes [20]. A retrieval system that relies solely on one representation tends to exhibit systematic failures. Sparse lexical retrieval excels when queries include specific model numbers, brand names, SKU-like tokens, and exact attribute constraints, but it can struggle with paraphrases and implicit intent. Dense retrieval can capture semantic similarity and can handle synonymy and paraphrase, but it can be sensitive to distribution shifts and may retrieve semantically related items that violate hard constraints. A practical backend therefore uses hybrid retrieval, but hybrid retrieval is not a single algorithm; it is an interplay of representation choices, filtering mechanisms, and fusion strategies, all of which interact with update consistency.

The product representation used for lexical indexing is constructed from multiple fields: title, description, category labels, brand, and selected attributes that are meaningful to users, such as size or material [21]. Structured fields are normalized into text in a controlled manner, using stable serialization rules so that tokenization is deterministic. Determinism matters for incremental indexing because nondeterministic serialization can create spurious diffs that increase churn. The lexical index is stored as a set of immutable segments, each built from a batch of updates, with background merging to control segment count. Each posting entry references a product identifier and stores term frequency and field-aware boosts. Attribute filters, such as price range and region, are not encoded as postings when they change frequently; instead, they are

Table 6: Consistency of index and cache updates under catalog churn

Scenario	Avg. Index Lag	Requests with Stale Results
Baseline nightly rebuild	6 h 12 m	7.8%
Hourly micro-batches	38 m	3.1%
Streaming without versioning	9.4 m	2.6%
Streaming with versioning	4.1 m	0.9%
Streaming + cache invalidation hints	3.6 m	0.6%

Table 7: RAG answer quality for marketplace product queries

Configuration	Recall@10	nDCG@10	Human Helpfulness (0–5)
BM25-only	0.71	0.62	3.2
Vector-only	0.78	0.66	3.6
BM25 + Vector (no RAG)	0.82	0.71	3.9
RAG w/o freshness guarantees	0.86	0.75	4.1
RAG with consistent updates	0.88	0.78	4.4
RAG + cache coherency	0.88	0.79	4.6

handled via a sidecar metadata store or via doc values that are optimized for filtering.

The dense representation is produced by an embedding model applied to a curated text view of the product [22]. This view may include the title and key attributes, possibly with templated text that exposes structured information to the model. Embedding computation is expensive relative to lexical tokenization, so the pipeline supports incremental updates by recomputing embeddings only when fields that influence the embedding change. This is done by maintaining a field-level dependency map for the embedding view. For example, a price change may not require embedding recomputation if price is not included in the embedding view, whereas a title edit does. To avoid silent drift when the embedding model changes, the system treats the model version and preprocessing version as part of the artifact identity [23]. A product can therefore have multiple embeddings in flight during a rollout, each associated with a model version and an epoch policy.

The vector index is typically sharded by product identifier hash to balance load and to support parallel updates. Within each shard, the index uses an approximate nearest neighbor structure that supports incremental inserts and deletes. Deletion is particularly important in marketplaces because products can be removed or become ineligible. Many ANN structures support lazy deletion via tombstones, which can accumulate and degrade performance; therefore, the system includes periodic compaction or rebuild mechanisms that are coordinated with epochs [24]. Inserts are handled by adding new vectors and associating them with the product identifier and product version. If a product's embedding is updated, the system either inserts a new vector and tombstones the old one or performs an in-place update if supported. In-place updates can reduce fragmentation but can complicate concurrency; tombstone-based updates are simpler but require careful handling to avoid retrieving stale vectors.

Hybrid retrieval proceeds by executing both sparse and dense retrieval and then fusing results. Fusion can be performed by score normalization and weighted combination, by reciprocal rank fusion, or by learned fusion. In a marketplace setting, fusion must also respect hard constraints, which often means that filtering is applied after candidate generation [25]. However, filtering after retrieval can waste capacity if many retrieved candidates are filtered out. The system therefore uses a two-level approach: coarse filtering is applied during retrieval using

Table 8: Ablation study of update and cache coherency mechanisms

Configuration	Consistent Index Updates	Coherent Cache	Throughput (QPS)
A: Baseline search			5.4k
B: + streaming updates			5.0k
C: + naive cache			6.1k
D: + versioned cache			7.8k
E: + adaptive invalidation			8.4k

Table 9: Online A/B test results on marketplace product search

Metric	Control (Legacy)	Treatment (Proposed)	Relative Change
Search click-through rate	18.6%	20.9%	+12.4%
Add-to-cart rate	6.3%	7.1%	+12.7%
Time-to-first-result (p95)	412 ms	323 ms	−21.6%
Query abandonment rate	9.8%	8.1%	−17.3%
GMV from search traffic	1.00x	1.06x	+6.1%

shard-local metadata, and fine-grained filtering is applied after fusion using a richer eligibility evaluator. Coarse filters include region eligibility, inventory availability, and policy flags that can be represented compactly. Fine filters include seller restrictions and dynamic promotions. Coarse filtering reduces waste and stabilizes latency at high load because it reduces the number of candidates passed downstream [26].

A crucial aspect of data representation is the separation of frequently changing fields from retrieval-sensitive fields. If inventory and price change frequently, embedding and lexical indexing should avoid incorporating them directly, and they should instead be enforced as filters or reranking features. This reduces index churn and improves stability. However, some user queries explicitly include price intent, and some marketplaces support queries such as “under 20 dollars.” This intent is best handled by query understanding and filter application rather than by embedding the price into the dense vector. The system includes a query understanding module that detects numeric constraints and maps them to structured filters [27]. Similarly, category and brand constraints can be extracted and applied as filters, which improves retrieval precision and reduces reliance on semantic similarity for hard constraints.

The feature store used for reranking holds both static and dynamic features. Static features include product quality signals, textual features, and category embeddings. Dynamic features include recent click-through rates, seller performance metrics, and inventory velocity. Dynamic features often update more frequently than indexes, so the feature store is designed for rapid updates and for serving low-latency reads keyed by product identifier [28]. To maintain coherency with retrieval, features are also versioned by epoch. This does not mean that every feature is recomputed per epoch; rather, the feature store exposes a read API that can return values as of a watermark associated with the epoch, ensuring that features are not drawn from a future state relative to the index snapshot. For some dynamic features, exact as-of semantics can be expensive; therefore, the system defines a set of features that must be epoch-aligned and a set that are allowed to be slightly ahead because they do not affect correctness, only ranking.

Context construction for generation requires careful selection of product fields. The system uses a context schema that includes identifiers, titles, short descriptions, prices, availability, and key attributes relevant to the query intent. The schema is designed to be stable across epochs and to support partial updates [29]. For example, if price

changes, the context generator should pull the latest price as of the epoch, not necessarily the latest absolute price. This ensures that the generated text is consistent with the retrieved set. The schema also includes provenance markers that allow downstream systems to detect if a generated response is out of date relative to the current epoch and to decide whether to reuse or regenerate.

Because marketplaces have long-tail diversity, multilingual content, and noisy seller-provided text, normalization is a substantial part of indexing. Normalization includes language detection, transliteration rules for certain scripts, and controlled handling of special tokens [30]. These steps are performed deterministically and are versioned so that index updates are reproducible. When normalization rules change, the system treats it similarly to a model rollout: it builds a new set of artifacts and advances epochs accordingly. This is important for cache coherency because a normalization change can invalidate cached retrieval results even if the underlying product data did not change.

Finally, the system includes a policy and eligibility layer that intersects with indexing. Certain products may be ineligible for certain users due to regulatory constraints or marketplace policies [31]. Eligibility changes can occur due to policy updates, seller status changes, or region-specific restrictions. The system handles eligibility in a manner similar to inventory: it is enforced primarily via filtering and metadata rather than via embedding updates. Eligibility metadata is stored in a low-latency store and is included in coarse filters when possible. When eligibility changes are widespread, such as a policy rollout affecting many products, the system may publish a new epoch that includes updated eligibility metadata while keeping retrieval indexes stable, thereby minimizing churn.

4. Consistent Index Updates and Visibility Control

Consistent index updates in a hybrid RAG backend require alignment across multiple derived artifacts. The central difficulty is that updates do not propagate uniformly: lexical updates are relatively cheap, feature updates can be fast but involve joins, and embedding updates are expensive and may be batched [32]. If the system simply updates each artifact independently and serves from the latest state of each, then a single request can observe a mixture of versions. The approach in this paper is to enforce a visibility boundary by using epochs that correspond to a set of offsets in the commit log. An epoch becomes the unit of coherency for serving, and index updates are organized to produce epoch-compatible artifact versions.

The commit log defines an ordered stream of product mutations. Each mutation is applied to the product state store, producing a new product version [33]. Derived pipelines then consume the state store or the mutation stream and produce artifacts. To support replay and determinism, the system stores the exact input used for artifact generation: for embeddings, this includes the serialized text view; for lexical indexing, the token list and field weights; for features, the input feature keys and join results where feasible. This does not require storing large intermediate data for all products, but it requires storing enough metadata to detect when two runs would differ. This metadata is also used for cache invalidation because it provides a stable fingerprint for the retrieval-relevant representation of a product.

Index updates can be applied in streaming or micro-batch modes [34]. Streaming updates reduce staleness but can be costly for ANN structures that are sensitive to frequent insertions. Micro-batching reduces overhead by grouping updates, which also aligns naturally with epoch publication. The system uses micro-batches per shard, typically with batch sizes determined by a target time window and a maximum update count. Each micro-batch produces a new lexical segment and a set of vector insertions and tombstones. The segment and vector updates are written to shard-local storage, but they are not immediately visible to serving. Instead, they are staged and associated with a pending epoch candidate [35].

Visibility control is implemented via a two-phase publication process. In the first phase, each shard writes its updated index artifacts and records a manifest entry that includes the shard identifier, the artifact pointers, and the highest commit log offset included. The manifest entry is written to a metadata store with conditional updates to prevent reordering. In the second phase, once all shards for all pipelines have reported readiness beyond a candidate offset, the epoch manager publishes a new epoch record that references the manifests. Read nodes then switch to the new epoch by loading the manifests and updating their in-memory routing [36]. The switch is designed to be atomic from the perspective of request handling: a request is assigned an epoch at admission and uses that epoch consistently across internal calls. During the transition, both the old and new epochs may be served concurrently for different requests, but each request sees only one.

A challenge arises when some shards lag. If the system waits for the slowest shard, epoch advancement can stall. If it advances without lagging shards, the epoch

may no longer represent a coherent snapshot [37]. The design addresses this by allowing bounded shard skew under explicit policy. The policy defines acceptable lag in terms of commit log offsets or time. For example, the system may allow a small fraction of shards to lag by a limited offset while serving them from the previous epoch, as long as cross-shard merges can tolerate partial staleness and as long as the lagging shards are not responsible for high-impact categories. This is not a general solution but a practical operational compromise. To prevent silent inconsistencies, the epoch record includes a per-shard epoch mapping, and the query gateway attaches both a global epoch and a shard map [38]. Retrieval services then merge results while tracking which shard epochs contributed. If shard skew is present, the system can adjust fusion weights or increase candidate counts to mitigate recall loss.

Embedding generation introduces additional complexity because embedding updates can be delayed and because the embedding model may change. The system treats embedding generation as an asynchronous pipeline that consumes product state versions and produces embedding artifacts. If a product update does not affect embedding inputs, the embedding pipeline can skip recomputation and simply carry forward the previous embedding artifact, but it must still record the association between the product version and the embedding artifact for traceability. If a product update affects embedding inputs, the pipeline computes a new embedding, stores it, and emits an update to the vector index [39]. To align with epochs, the pipeline emits readiness signals only when both the embedding artifact and the vector index update are durable.

When the embedding model changes, the system may need to recompute embeddings for a large portion of the catalog. A naive approach would rebuild the entire vector index and then switch, but this can be resource-intensive. Instead, the system supports dual index operation. During a rollout, the system maintains two vector indexes per shard: one for the old model and one for the new [40]. Queries can be served using either, controlled by epoch policy. The epoch manager can publish epochs that gradually shift traffic to the new model by associating the dense retriever with a model-specific index pointer. Because the lexical index and other artifacts may remain unchanged, the rollout can be localized to dense retrieval while maintaining coherency. However, caches must be aware of the model version because dense retrieval results differ. Therefore, the epoch identity includes the dense model version, and cache keys incorporate it [41].

Backfills and repair operations are inevitable in production. An indexing pipeline may miss updates due to transient failures, or a shard may become corrupted and require rebuild. The epoch mechanism provides a controlled way to recover because the system can replay the commit log from a known offset and regenerate artifacts deterministically. During repair, the shard can rejoin the latest epoch by catching up to the current watermark and then publishing readiness. If the shard cannot catch up quickly, the system may isolate it by routing a subset of traffic away or by using fallback retrieval. A common fallback is to use lexical retrieval only for that shard's product range, sacrificing some semantic recall but maintaining availability [42]. Importantly, such fallbacks are also tied to epochs and explicitly represented in routing so that caching and debugging remain consistent.

Deletes and eligibility removals require careful handling to avoid serving removed items from caches. The system represents deletes as mutations that set an eligibility flag and may remove text and attributes. The lexical index marks the product as deleted via a doc value or via a deletion bitset, and vector index updates tombstone the embedding. Because tombstones may not immediately remove the item from ANN candidates, the eligibility filter enforces removal at read time [43]. This filter is epoch-aligned, ensuring that a product that is deleted before an epoch boundary is not returned in that epoch. If an item is deleted after an epoch boundary, it may still appear for requests served under the previous epoch, which is an expected consequence of snapshot serving. The system bounds this window by controlling epoch publication frequency, balancing freshness against overhead.

Atomicity across artifacts is achieved by treating epoch publication as the commit point. Before publication, artifacts can be written in any order, and partial updates can exist in storage, but they are not referenced by a published epoch [44]. Once an epoch is published, it becomes visible to serving. This approach resembles multi-version concurrency control, but implemented at the level of search artifacts rather than database rows. The advantage is that serving never observes partially updated indexes, and caches can safely reuse results within an epoch. The cost is increased storage because multiple epochs may coexist, and the system must manage retention by deleting old segments after a grace period.

A subtle issue is the alignment of reranker features with retrieval candidates. If features are updated more quickly than indexes, then features could reflect a future state for a product that is not yet visible in retrieval [45]. This can be acceptable if the features do not contradict retrieval-relevant constraints, but it can also produce inconsistent explanations in generation. The system there-

fore defines a feature watermark per epoch. Feature updates are stored with timestamps or offsets, and the feature store read path supports an as-of query. In practice, exact as-of reads can be expensive, so the system may implement as-of by partitioning features into coarse time buckets and selecting the bucket that corresponds to the epoch watermark. This yields bounded inconsistency and predictable behavior [46].

5. Cache Coherency for Retrieval, Ranking, and Generation

Caching in a RAG backend spans multiple layers: query embedding caches, retrieval result caches, reranker feature caches, and generation output caches. Without coherency controls, caching can amplify inconsistency because caches persist results beyond the lifetime of the underlying index versions. The design presented here uses epoch-tagged caching and selective invalidation. Epoch-tagged caching ensures that cached entries are only reused within the epoch in which they were computed, or across epochs when the system can prove backward compatibility. Selective invalidation handles cases where reusing across epochs would violate correctness or would be undesirable due to policy changes [47].

The foundational mechanism is generational cache keying. Each cache entry key is augmented with an epoch identifier and, when relevant, additional artifact identifiers such as dense model version, normalization version, and eligibility policy version. For example, a retrieval cache key might include the normalized query, filter set, user region, and epoch. This ensures that a retrieval set computed for epoch e is not served for epoch e 1 unless explicitly allowed. Similarly, a query embedding cache key includes the query normalization version and the embedding model version [48]. If the normalization or model changes, embeddings are recomputed rather than reused.

Epoch-tagged caches reduce the risk of stale results but can reduce hit rates because entries are not reused across epochs. To mitigate this, the system supports controlled cross-epoch reuse using compatibility windows. A compatibility window is defined when the system can assert that certain artifacts did not change between epochs for a given class of keys. For instance, if the lexical index did not change for a shard between two epochs, then lexical retrieval results for that shard could be reused. The system encodes this by including shard-level artifact hashes in epoch metadata [49]. When a read node receives an epoch, it also receives the artifact hashes, and it can decide to reuse cache entries whose hashes match. This approach is particularly useful when epochs advance due to feature updates or eligibility metadata updates while the core retrieval indexes remain unchanged.

Selective invalidation is necessary for caches that cannot be fully partitioned by epoch due to cost or due to desired longevity. Feature caches, for example, may cache product-level features that are reused across many queries and many epochs. These caches can be made coherent by attaching a per-product version counter [50]. When a product mutation occurs, the system emits an invalidation event for that product's feature cache entries. The feature cache key includes the product identifier and the product version, so reading features for a product at a given epoch uses the version corresponding to that epoch watermark. If the feature store does not support exact as-of semantics, the cache uses the latest version not exceeding the epoch watermark.

Retrieval result caching is more challenging because a query can return a set of many products, and invalidating all queries that include a changed product is not feasible. The system addresses this by designing retrieval caches to be short-lived and epoch-scoped, and by caching at intermediate levels where dependencies are simpler [51]. Instead of caching the final fused ranked list, the system caches shard-local candidate sets for sparse retrieval and for dense retrieval separately. Shard-local caches have smaller dependency surfaces because they depend mainly on the shard's index artifacts. When a shard's artifact changes, the shard-local cache can be invalidated by epoch keying without needing per-product dependency tracking. The fusion step can also be cached but typically has lower reuse because it depends on combining across shards and on the filter set.

Generation output caching introduces distinct coherency concerns because the output can contain specific factual statements. Caching generation outputs can be cost-effective, especially for popular queries, but a cached output can become misleading if any referenced product attribute changes [52]. The system therefore uses a two-level approach. First, generation caches are epoch-tagged, so outputs are reused only within the epoch by default. Second, for limited cross-epoch reuse, the system attaches a dependency fingerprint to each generated output. The fingerprint is computed from the product identifiers included in the context and from a compact hash of the attributes that the generation output is allowed to mention, such as price and availability. When serving a cached output in a later epoch, the system checks whether the fingerprint matches the current state as of that epoch [53]. If it does, the output can be reused. If not, the system re-

generates. This check is intentionally lightweight: it uses precomputed per-product attribute hashes stored in a fast metadata store. It does not attempt to verify every detail, but it provides a practical guardrail.

Cache coherency must also handle concurrent epoch transitions [54]. If many read nodes switch epochs at the same time, they can simultaneously miss caches and overload retrieval and generation services. The system mitigates this with staggered epoch adoption and with background warming. Staggered adoption means that read nodes introduce a randomized delay before switching to a new epoch, within a bounded window. Requests arriving during the window are still served by the old epoch on some nodes, which preserves cache warmth and smooths load. Background warming means that, upon receiving a new epoch, a node precomputes cache entries for a set of hot queries and hot filters, based on recent traffic [55]. Warming is performed with low priority and is rate-limited, and it can be disabled under high load.

A related issue is cache coherency across regions. If epochs are regional, then a query served in one region at epoch e_r may have a different epoch in another region $e_{r'}$. If caches are shared globally, this can cause confusion. The system therefore scopes caches to regions or includes region identifiers in keys. For globally shared caches, such as a shared embedding cache, the key includes the preprocessing and model version and can often be reused across regions because query embeddings do not depend on region, but retrieval results and generation outputs typically must be region-scoped because of availability and eligibility differences.

The system also includes negative caching to improve throughput, such as caching that a query yields no results under certain filters [56]. Negative caching must be coherent because catalog updates can turn a previously empty result into a non-empty one, for example when new inventory arrives. Negative cache entries are therefore given short time-to-live values and are epoch-scoped. If the system detects that a query is empty due to a transient constraint such as inventory, it may choose not to negative-cache at all, or it may include a dependency on the relevant category inventory watermark.

Coherency extends to rate limits and admission control. Generation is often subject to quotas, and caching can interact with quotas in subtle ways [57]. If cached generation outputs are served, they should not consume generation quota, but if cache misses spike during an epoch transition, quota exhaustion can occur. The system addresses this by introducing a generation budget per epoch and by reserving capacity for regeneration during transitions. Additionally, the system may degrade gracefully by serving retrieval-only results when generation is unavailable. This degradation is coherent because it is tied to the request epoch and because it does not mix partial generated text with new retrieval candidates.

Finally, cache observability is treated as a first-class requirement [58]. Cache entries include metadata such as the epoch, artifact hashes, and dependency fingerprints, and the system samples cache hits and misses for logging. When a user-visible inconsistency is reported, operators can inspect whether the response came from cache, which epoch it used, and whether dependency checks were performed. This reduces debugging time and provides confidence that coherency mechanisms are functioning as designed.

6. Performance Modeling and Empirical Evaluation Framework

A high-throughput backend must be engineered with an explicit understanding of how latency, throughput, cache hit rate, and staleness interact. While empirical benchmarking is essential, analytical models are useful for capacity planning and for reasoning about tradeoffs without exhaustive experimentation. This section presents a modeling framework that connects the epoch-based consistency mechanism with queueing behavior and with update propagation delays [59]. The goal is not to produce a single closed-form expression for system behavior, but to provide interpretable relationships that can guide design choices.

Consider the read path as a sequence of service stages: query understanding, sparse retrieval, dense retrieval, fusion, reranking, and optional generation. Each stage i can be modeled as a queue with k_i parallel workers and service time distribution S_i. Let λ be the request arrival rate. Under a simplified independence assumption, the end-to-end latency distribution is influenced by the convolution of stage latencies, but tail latency is typically dominated by the slowest stage or by correlated congestion [60]. For capacity planning, a common approximation is to treat each stage as an MGk queue with utilization $\rho_i = \lambda E S_i k_i$. Maintaining ρ_i below a threshold is necessary to control tail latency. When caching is present, the effective service time becomes a mixture distribution because cache hits bypass some stages. Let h_i be the cache hit probability for stage i such that a hit reduces the stage

service time to a smaller value S_i^{hit}, potentially near zero, while a miss uses S_i^{miss}. Then

$$ES_i \quad = \quad h_i ES_i^{hit} \quad 1 - h_i ES_i^{miss} \quad . \qquad (6.1)$$

For dense retrieval, S_i^{miss} includes embedding computation and ANN search, while S_i^{hit} may include only lookup of a cached embedding or cached candidates. For generation, S_i^{miss} includes the LLM call, while S_i^{hit} may include only serialization. In practice, h_i is epoch-dependent because epoch-tagged caching reduces cross-epoch reuse. Therefore, after an epoch transition, h_i can drop temporarily and then recover as caches warm [61]. This dynamic can be modeled by a time-dependent hit rate $h_i t$, where t is time since epoch adoption. Warming mechanisms aim to accelerate the recovery of $h_i t$ for hot keys.

The epoch publication frequency affects both freshness and cache performance. Let epochs be published every T_e seconds on average. If caches are strictly epoch-scoped with no cross-epoch reuse, then the maximum lifetime of a cache entry is approximately T_e plus any stagger window. Shorter T_e improves freshness because new updates become visible sooner, but it can reduce cache hit rate because entries expire more frequently [62]. The tradeoff can be expressed by modeling the probability that a query repeats within the same epoch. If the query arrival process for a specific key is Poisson with rate λ_q, the probability of at least one repeat within an epoch window of length T_e is $1 - e^{-\lambda_q T_e}$. This suggests that for low-frequency queries, epoch-scoped caching provides limited benefit, while for high-frequency queries, it can still be effective even with small T_e. Because marketplaces often have a heavy-tailed query distribution, a small set of queries contributes a large fraction of traffic, and those queries can sustain high hit rates even under frequent epoch changes.

Update propagation delay determines staleness. Let a product mutation occur at time t_0 [63]. The mutation is appended to the commit log and then processed by multiple pipelines. Let D_{lex} be the delay until the lexical index artifact is ready, D_{dense} the delay until the dense index update is ready, D_{feat} the delay until feature updates are ready, and D_{elig} the delay until eligibility metadata is ready. The epoch manager publishes an epoch only when all required pipelines have advanced beyond the mutation's offset. If the epoch boundary is defined by the minimum pipeline watermark, then the mutation becomes visible at time $t_0 \max D_{lex}, D_{dense}, D_{feat}, D_{elig}$ plus any batching and publication delay. Denote the total visibility delay as D_{vis}. Then the staleness of reads under the latest epoch is bounded by D_{vis} in the sense that no read will see a state more recent than $t - D_{vis}$, where t is current time, assuming continuous traffic and steady operation.

To reason about the distribution of D_{vis}, one can model pipeline delays as random variables with distributions determined by batch sizes, compute capacity, and queueing in the write plane. If each pipeline processes micro-batches of size B and each batch takes time C to compute and write, then the delay includes waiting for the batch to fill and processing time. Under a simple model with a constant update rate λ_u, the expected waiting time to fill a batch is approximately $B\lambda_u$ when updates are evenly distributed, but in reality updates are skewed by product. Skew can increase effective waiting time for shards with fewer updates [64]. The system can mitigate this by using time-based flushing in addition to size-based batching, ensuring that batches are emitted at least every T_f seconds. Then the waiting component is bounded by T_f. The total expected delay can be approximated as

$$ED_{vis} \quad \approx \quad E\big[\max D_{lex}, D_{dense}, D_{feat}, D_{elig}\big] \quad ED_{pub} \quad , \qquad (6.2)$$

where D_{pub} captures epoch manager publication and read node adoption delay. Staggered adoption increases D_{pub} slightly but reduces the risk of synchronized cache misses.

Cache coherency mechanisms affect effective staleness as perceived by users. If a response is served from cache, it may reflect an older epoch, depending on how the request epoch is assigned [65]. In the design here, requests are assigned the current epoch at admission, and caches are epoch-tagged, so a cache hit implies that the cached entry was computed for that epoch, not an older one. Therefore, caching does not increase staleness relative to epoch serving, but it can increase staleness relative to the absolute latest catalog state, which is already bounded by D_{vis}. Cross-epoch reuse with compatibility windows can reuse entries across epochs when artifacts are unchanged, which does not increase staleness for the relevant artifact, but it can create subtle behavior when only some artifacts are unchanged. The system avoids mixing by requiring that compatibility reuse is only applied when all artifacts that influence the cached computation are unchanged, which is verified by artifact hashes.

Empirical evaluation for such a system should measure not only retrieval quality and latency, but also coherency and freshness. Coherency can be assessed by injecting synthetic mutations and verifying that requests served at a given epoch reflect a consistent state across

sparse retrieval, dense retrieval, and feature reads. One practical approach is to select a set of sentinel products with controlled mutations and to log whether those mutations appear in retrieval contexts and generation outputs at expected epochs [66]. Freshness can be measured by tracking the time between a mutation's commit log offset and the first epoch in which it is visible, producing a distribution of D_{vis}. Additionally, one can measure the incidence of stale cache usage by logging epoch identifiers on cache hits and verifying that they match the request epoch.

Throughput and tail latency evaluation requires load testing under realistic query distributions and update rates. Because query distributions are heavy-tailed, uniform random queries can underestimate cache benefit and overestimate compute requirements. Load tests should replay production-like query traces while simultaneously replaying update traces. The system should be tested under different epoch publication frequencies to observe the cache hit rate versus staleness tradeoff [67]. One expects that reducing T_e will reduce D_{vis} but will also reduce retrieval cache hit rates, particularly for mid-frequency queries, and will increase load on dense retrieval and generation. Warming and stagger mechanisms can be evaluated by comparing tail latency during epoch transitions with and without these mechanisms.

A useful metric for cache effectiveness in the presence of epochs is the hit rate conditioned on epoch age. Define $h_i\tau$ as the hit rate for stage i at time τ since adopting a new epoch. Plotting $h_i\tau$ can reveal whether warming is effective and whether certain layers recover more slowly. For example, query embedding caches may recover quickly because embeddings depend only on query text and model version, whereas retrieval result caches may recover more slowly because they depend on filters and shard routing [68]. Generation caches may recover slowly if generation is triggered only for a subset of queries.

One can also model the cost impact. Let c_{dense} be the cost per dense retrieval miss, c_{gen} the cost per generation miss, and c_{rerank} the cost per rerank evaluation. Then the expected cost per request is

$$EC \quad = \quad 1 - h_{dense}c_{dense} \quad c_{rerank} \quad p_{gen}1 - h_{gen}c_{gen}, \tag{6.3}$$

where p_{gen} is the probability that a request triggers generation. Epoch decisions influence h_{dense} and h_{gen}, so cost can increase when epochs advance more frequently. This can be partially offset by increasing cross-epoch reuse where safe, by improving warming, or by reducing p_{gen} through better intent gating. The model emphasizes that coherency mechanisms have economic consequences and should be tuned with explicit objectives.

Finally, the system's update plane must be evaluated for write amplification and resource usage. Lexical segment merging and ANN tombstone accumulation can consume I/O and memory [69]. If epochs are published frequently, retention policies must ensure that old segments are reclaimed promptly without breaking in-flight requests. A typical approach is to retain at least two epochs and to keep older epochs for a short grace period equal to the maximum request duration plus a buffer. This retention window can be measured and used to size storage. Empirically, one can measure the number of segments and tombstones per shard over time and correlate spikes with update bursts and with model rollouts.

7. Conclusion

This paper described a high-throughput retrieval-augmented generation backend for marketplace product search with an emphasis on consistent index updates and cache coherency under frequent catalog mutations [70]. The central design principle is to treat coherency as a first-class system property by defining an epoch abstraction that aligns hybrid retrieval indexes, feature stores, and generation behavior. Updates flow through an event-sourced commit log into derived artifact pipelines, and visibility is controlled by atomic epoch publication based on pipeline watermarks. This approach allows the system to avoid serving partially updated states within a single request, while still supporting high update rates through micro-batching, shard-parallel indexing, and asynchronous embedding generation.

Hybrid retrieval is supported by deterministic data representations, separation of frequently changing attributes from retrieval-sensitive text, and filtering that combines shard-local coarse constraints with fine-grained eligibility evaluation. Consistent updates are achieved through staged artifact manifests and atomic epoch advancement, with operational mechanisms for lagging shards, backfills, and model rollouts using dual indexes [71]. Cache coherency is implemented via epoch-tagged generational keys, compatibility windows guided by artifact hashes, and selective invalidation for product-level caches and generation outputs using dependency fingerprints. These mechanisms aim to preserve cache utility while bounding staleness and preventing contradictory mixes of versions.

Analytical models based on queueing approximations and update propagation delays provide a framework

for capacity planning and for reasoning about tradeoffs among epoch frequency, cache hit rates, tail latency, and freshness. Empirical evaluation in such systems benefits from measuring not only latency and retrieval effectiveness but also coherency properties, such as whether retrieval and generation observe consistent product states at a given epoch. The overall design does not eliminate tradeoffs; it shifts them into explicit policy choices about epoch advancement, compatibility reuse, and degradation behaviors. In marketplaces where correctness and user trust depend on coherent visibility of rapidly changing product attributes, such explicitness can be operationally useful, even when the system remains eventually consistent with respect to the absolute latest catalog state [72].

References

[1] Z. F. Hamida, A. Refoufi, and A. Drif, *Fake News Detection Methods: A Survey and New Perspectives*, pp. 123–141. Springer International Publishing, 2 2022.

[2] H. H. Ali and A. D. S. H. Shaker, "Techniques for secure distributed systems," *Journal of Physics: Conference Series*, vol. 1530, pp. 012006–, 5 2020.

[3] F. D. Gregorio, A. Gamatié, G. Sassatelli, A. Castelltort, and M. Robert, "Exploration of energy-proportional distributed systems," 6 2019.

[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. Hermann, N. Trkulja, E. Meißner, B. Erb, and F. Kargl, "Demo: Quantifying trust in a trust assessment framework," in *2025 IEEE Vehicular Networking Conference (VNC)*, pp. 1–2, IEEE, 6 2025.

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

[7] "Reasoning about knowledge in byzantine distributed systems," 7 2020.

[8] I. Beschastnikh, P. Liu, A. Xing, P. Wang, Y. Brun, and M. D. Ernst, "Visualizing distributed system executions," *ACM Transactions on Software Engineering and Methodology*, vol. 29, pp. 1–38, 3 2020.

[9] H. Shukur, S. Zeebaree, R. Zebari, O. Ahmed, L. Haji, and D. Abdulqader, "Cache coherence protocols in distributed systems," *Journal of Applied Science and Technology Trends*, vol. 1, pp. 92–97, 6 2020.

[10] M. Rafailescu, "Fault tolerant leader election in distributed systems," *Zenodo (CERN European Organization for Nuclear Research)*, 4 2020.

[11] A. Luntovskyy and D. Gütter, *Definition: Highly-Distributed Systems*, pp. 3–23. Springer International Publishing, 2 2022.

[12] 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.

[13] 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.

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

[15] Z. Yuan, J. Shi, P. Zhou, N. Z. Gong, and L. Sun, "Badtoken: Token-level backdoor attacks to multimodal large language models," in *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 29927–29936, IEEE, 6 2025.

[16] T. Pusztai, S. Nastic, P. Raith, S. Dustdar, D. Vij, and Y. Xiong, "Vela: A 3-phase distributed scheduler for the edge-cloud continuum," in *2023 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 161–172, IEEE, 9 2023.

[17] K. He, "Research on verifiable access technology of mixed database in distributed system under big data," in *2021 International Conference on Aviation Safety and Information Technology*, pp. 636–640, ACM, 12 2021.

[18] M. Farhadi, D. Miorandi, and G. Pierre, "Blockchain enabled fog structure to provide data security in iot applications," 12 2018.

[19] D. Le-Phuoc and M. Hauswirth, *Semantic Stream Processing and Reasoning*, pp. 1–10. Springer International Publishing, 3 2022.

[20] H. Zhou, W. Qian, X. Zhou, Q. Dong, A. Zhou, and W. Tan, "Scalable and adaptive log manager in distributed systems," *Frontiers of Computer Science*, vol. 17, 8 2022.

[21] 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.

[22] Y. Braidiz, D. Efimov, A. Polyakov, and W. Perruquetti, "On robustness of finite-time stability of homogeneous affine nonlinear systems and cascade interconnections," *International Journal of Control*, pp. 1–11, 9 2020.

[23] A. Jan, S. A. A. Shah, Z. Khan, S. Jan, A. Haroon, I. Khan, and H. Tariq, "Upgraded deadlock averting algorithms in distributed systems," 6 2019.

[24] A. Labutkina, A. Selezneva, T. John, and D. Hausheer, "Multiobjective path optimization for deadline-aware multipath over scion," in *2024 IEEE International Conference on Machine Learning for Communication and Networking (ICMLCN)*, pp. 56–62, IEEE, 5 2024.

[25] K. Johari, G. K. Agarwal, and Y. D. Arya, "Server load balancing analysis for client task assignment in distributed systems," *ECS Transactions*, vol. 107, pp. 17605–17621, 4 2022.

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

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

[28] 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.

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

[30] A. K. Tyagi, S. Tiwari, and K. Naithani, "Distributed systems and distributed ledger technology - an introduction," 9 2024.

[31] 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.

[32] 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.

[33] H. B. Sakariya and G. D, "Taxonomy of load balancing strategies in distributed systems," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 12, pp. 1796–1802, 3 2024.

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

[35] I. Zhuzhgina and A. Lazarev, "An intelligently distributed system for controlling information flows," *E3S Web of Conferences*, vol. 431, pp. 5017–05017, 10 2023.

[36] G. L. Stavrinides and H. D. Karatza, "Icfnds - scheduling techniques for complex workloads in distributed systems," in *Proceedings of the 2nd International Conference on Future Networks and Distributed Systems*, pp. 34–6, ACM, 6 2018.

[37] 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.

[38] A. Charu, "Resource allocation of distributed system based on hierarchical clustering algorithm," *Distributed Processing System*, vol. 3, 7 2022.

[39] T. M. Tatarnikova and E. M. Arkhiptsev, "Hybrid time synchronization in distributed systems," in *2025 XXVIII International Conference on Soft Computing and Measurements (SCM)*, pp. 307–310, IEEE, 5 2025.

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

[41] M. Maćkowski, M. Kawulok, P. Brzoza, and D. Spinczyk, *Method and Tools to Supporting Math Learning in Inclusive Education of Blind Students*, pp. 42–53. Germany: Springer Nature Switzerland, 5 2023.

[42] G. Pierre, "Le « fog computing » est l'avenir du cloud – en plus frugal et plus efficace," *The conversation*, pp. 1–4, 8 2021.

[43] I. S. Ochôa, L. A. Silva, G. de Mello, B. D. Silva, J. F. D. Paz, G. V. González, N. M. Garcia, and V. R. Q. Leithardt, "Prichain: A partially decentralized implementation of ubipri middleware using blockchain.," *Sensors (Basel, Switzerland)*, vol. 19, pp. 4483–, 10 2019.

[44] J. L. Clark, "A design approach for a distributed system to handle chinese," *Proceedings of the Annual Conference of CAIS / Actes du congrès annuel de l'ACSI*, 3 2022.

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

[46] H. Zhang, N. Lu, R. Song, and J. Li, "Modelling and optimizing for distributed systems with limited resource migration capacity," *Journal of Physics: Conference Series*, vol. 1670, pp. 012026–, 11 2020.

[47] A. Ramesh, T. Huang, E. Ruppel, D. Dasari, B. Pourmohseni, F. Smirnov, M. Giani, P. Pazzaglia, C. Shelton, N. Pereira, A. Hamann, D. Ziegenbein, and A. Rowe, "Silverline: Lightweight virtualization and orchestration of distributed systems," in *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 375–388, IEEE, 5 2025.

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

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

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

[51] T. Nowak, U. Schmid, and K. Winkler, "Topological characterization of consensus in distributed systems," *Journal of the ACM*, vol. 71, pp. 1–48, 11 2024.

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

[53] 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.

[54] M. A. Saeed and M. A. Saeed, "Real-time diabetes detection using machine learning and apache spark," in *2024 4th International Conference on Emerging Smart Technologies and Applications (eSmarTA)*, pp. 1–6, IEEE, 8 2024.

[55] B. Full, J. Manner, S. Böhm, and G. Wirtz, *MicroStream vs. JPA: An Empirical Investigation*, pp. 99–118. Germany: Springer International Publishing, 10 2022.

[56] W. Liao, L. Han, and P. Chen, "Design and optimization of asymmetric encryption scheme based on blockchain and its application in privacy protection of internet of vehicles," in *2024 IEEE International Conference on High Performance Computing and Communications (HPCC)*, pp. 1090–1097, IEEE, 12 2024.

[57] K. K. Horvath, D. Kimovski, B. Spiess, O. Hohlfeld, and R. Prodan, "Seal-cc: Scalable latency evaluation

methodology for internet-of-things services," in *Proceedings of the 14th International Conference on the Internet of Things*, pp. 117–126, ACM, 11 2024.

[58] C. Stach, J. Bräcker, R. Eichler, C. Giebler, and B. Mitschang, "Demand-driven data provisioning in data lakes," in *The 23rd International Conference on Information Integration and Web Intelligence*, pp. 187–198, ACM, 11 2021.

[59] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Programming open distributed systems in maude," in *Proceedings of the 26th International Symposium on Principles and Practice of Declarative Programming*, pp. 1–12, ACM, 9 2024.

[60] B. Sedlak, A. Morichetta, P. Raith, V. C. Pujol, and S. Dustdar, "Towards multi-dimensional elasticity for pervasive stream processing services," in *2025 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pp. 657–660, IEEE, 3 2025.

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

[62] M. Maidan and A. Melnyk, "Organization of fpga-based devices in distributed systems," *International Journal of Computing*, pp. 352–359, 10 2023.

[63] D. Calvanese, F. D. Cosmo, J. Lobo, and M. Montali, *CILC - Convergence Verification of Declarative Distributed Systems.* 12 2021.

[64] E. Arif, "Implementing a distributed system as a solution to problems in the indosat cooperative," *JURNAL PETIK*, vol. 5, pp. 24–29, 10 2019.

[65] S. Schewe, *Synthesis of Distributed Systems*. 7 2018.

[66] I. Schagaev, H. Cai, and S. Monkman, *On Performance: From Hardware up to Distributed Systems*, pp. 297–324. Springer International Publishing, 7 2024.

[67] F. L. Mouël, "Youpi: retour d'expérience et travaux futurs sur une plateforme fog/edge computing," 7 2019.

[68] L. Alberto, "Pseudospheres: combinatorics, topology and distributed systems," *Journal of Applied and Computational Topology*, vol. 8, pp. 1023–1052, 2 2024.

[69] 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.

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

[71] M. Witter and A. R. D. Vit, "Blockchain e sistemas distribuídos: conceitos básicos e implicações," 3 2024.

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