Distributed Document Search Service

Architecture Design Document

Objective

Design a horizontally scalable, multi-tenant document search platform capable of indexing and searching 10M+ documents with sub-500ms P95 latency while supporting 1000+ concurrent queries per second and maintaining tenant isolation.

The system prioritizes availability, performance, and operational simplicity while remaining extensible for enterprise requirements.

---

1. High-Level Architecture

Logical Architecture Diagram

```
                   ┌──────────────┐
                   │   Clients    │
                   └──────┬───────┘
                          │
                   ┌──────▼───────┐
                   │ API Gateway  │
                   │ Auth + Rate  │
                   │ Limiting     │
                   └──────┬───────┘
                          │
             ┌────────────▼─────────────┐
             │ Document Search Service  │
             │ (Spring Boot / REST)     │
             └──────┬────────┬─────────┘
                    │        │
             ┌──────▼────┐ ┌─▼────────────┐
             │ Redis     │ │Elasticsearch│
             │ Cache     │ │ Search Index│
             └───────────┘ └────┬─────────┘
                                │
                      ┌─────────▼────────┐
                      │ PostgreSQL        │
                      │ Document Store   │
                      └─────────┬────────┘
                                │
                        ┌───────▼───────┐
                        │ Kafka         │
                        │ Event Bus     │
                        └───────────────┘
```

---

2. Data Flow

Document Indexing Flow

```
Client → POST /documents
      → API Service
      → PostgreSQL (persist document)
      → Kafka (publish document-created event)
      → Indexer Consumer
      → Elasticsearch (index document)
```

Rationale

- PostgreSQL is the system of record.
- Elasticsearch indexing is asynchronous to keep write latency low.
- Kafka decouples ingestion from indexing and provides retryability.

Search Flow

```
Client → GET /search
      → API Service
      → Redis (cache lookup)
      → Miss
      → Elasticsearch
      → Cache results
      → Client
```

This enables fast responses for repeated queries while preserving freshness via short TTL.

---

3. Database & Storage Strategy

PostgreSQL (Primary Store)

- Stores canonical document data
- ACID compliance for writes
- Tenant metadata and access control

Chosen for reliability, transactional guarantees, and operational maturity.

Elasticsearch (Search Engine)

- Full-text search
- Relevance ranking
- Horizontal scaling via sharding
- Replica-based fault tolerance

Documents are indexed with `tenantId` as a routing key.

Large tenants can optionally be isolated into dedicated indices.

Redis (Caching Layer)

- Hot search query cache
- Frequently accessed documents
- Reduces Elasticsearch load

Kafka (Messaging)

- Asynchronous document indexing
- Event replay capability
- Dead-letter queues for failed processing

---

4. API Design

Index Document

```
POST /documents
Headers:
X-Tenant-Id

Body:
{
  "title": "Sample",
  "content": "Text content"
}
```

Search Documents

```
GET /search?q=distributed&tenant=tenantA
```

Response:

```
{
  "results": [
    {
      "id": "123",
      "title": "Distributed Systems"
    }
  ]
}
```

Retrieve Document

```
GET /documents/{id}
```

Delete Document

```
DELETE /documents/{id}
```

Health Check

```
GET /health
```

Returns dependency status for PostgreSQL, Redis, and Elasticsearch.

---

5. Consistency Model

- PostgreSQL: Strong consistency (source of truth)
- Elasticsearch: Eventual consistency (async indexing)

Trade-off

✔ Fast ingestion
✔ High availability
✔ Search scalability

✘ Small delay between write and search visibility (acceptable for search workloads)

---

6. Caching Strategy

Layer	Purpose	TTL
Redis Query Cache	Repeated searches	~60s
Redis Document Cache	Document reads	~5 min

Cache key format:

```
tenantId:hash(query)
```

Cache invalidation occurs on document update/delete events.

---

7. Message Queue Usage

Kafka topics:

- `document-created`
- `document-deleted`

Indexer consumers:

- Batch index documents
- Retry on transient failures
- Send unrecoverable messages to DLQ

This ensures resilient ingestion and protects Elasticsearch from spikes.

---

8. Multi-Tenancy & Data Isolation

Primary Approach

- Shared infrastructure
- Logical isolation using `tenantId`
- Elasticsearch routing + filtered aliases
- Mandatory tenant validation at API layer

Security Enforcement

- API Gateway validates JWT + tenant claims
- Service layer verifies tenant ownership
- Elasticsearch queries always filtered by tenantId

Enterprise Upgrade Path

- Dedicated indices for large tenants
- Separate Kafka topics
- Optional per-tenant resource quotas

Summary

This architecture provides:

- Horizontal scalability through stateless services and sharded search
- Sub-500ms search via Elasticsearch + Redis
- Strong write consistency with eventual search consistency
- Secure multi-tenant isolation
- Resilient async processing with Kafka

It balances performance, operational simplicity, and enterprise extensibility while supporting future growth to hundreds of millions of documents.

Enterprise Experience Showcase

1. Distributed System Built at Scale

At Armorcode, I worked on a distributed application security data ingestion and analytics platform that aggregated findings from 20+ external scanners (SAST, DAST, SCA, cloud security tools) into a unified risk management system. The platform processed millions of security records per month across multiple enterprise tenants. The architecture was built using Java microservices, Kafka for event streaming, Elasticsearch for search and analytics, and PostgreSQL as the system of record, deployed on AWS.

I was responsible for designing ingestion pipelines, tenant isolation strategies, and scaling Elasticsearch indices. The system supported thousands of concurrent users with sub-second search latency. By introducing async processing via Kafka and separating write and read paths, we achieved predictable performance while allowing independent horizontal scaling of ingestion and search workloads.

2. Performance Optimization with Measurable Impact

In a previous role, we faced severe latency issues in a high-traffic search API where P95 response times exceeded 2 seconds during peak hours. After profiling, I identified excessive database joins and repetitive Elasticsearch queries as the primary bottlenecks. I introduced Redis-based query caching for hot searches, optimized ES mappings, added proper routing keys, and replaced synchronous enrichment calls with async batch processing.

These changes reduced P95 latency from ~2 seconds to under 400ms and decreased Elasticsearch load by nearly 45%. The optimization also improved system stability during traffic spikes and significantly reduced infrastructure costs by allowing us to scale down ES nodes.

3. Critical Production Incident in a Distributed System

We encountered a production incident involving AWS Elasticsearch where, despite having hourly automated snapshots enabled, there was still a risk of data loss between backup windows during a cluster instability event. This created a potential gap where recently indexed documents could not be recovered if the cluster required restoration, impacting data integrity for multiple tenants.

To mitigate this, I designed and implemented an Elasticsearch query logging mechanism at the application layer, capturing all write operations (index, update, delete) into a durable store. This effectively created an application-level change log that allowed us to replay operations after restoring from snapshots. The solution acted as a lightweight event-sourcing layer for

Elasticsearch, enabling near-complete recovery even beyond the last snapshot. As a result, we eliminated blind data-loss windows, improved recovery confidence, and strengthened our overall disaster recovery strategy without introducing significant write latency.

---

4. Architectural Decision Balancing Competing Concerns

While designing our multi-tenant Elasticsearch architecture, we initially adopted a shared-index model using `tenantId` as the routing key. This approach provided efficient shard locality, simplified operations, and allowed us to onboard new tenants quickly while keeping infrastructure costs low. However, as some customers grew significantly larger, their indexing and query patterns began to impact cluster performance for smaller tenants.

To address this, I proposed a tiered isolation strategy. Most tenants continued using the shared index with routing, large tenants were migrated to dedicated indices within the same cluster, and ultra-large enterprise customers were provisioned on fully dedicated Elasticsearch clusters. This progressive isolation model balanced cost efficiency, operational complexity, and performance guarantees. It allowed us to scale selectively based on tenant size while maintaining strong isolation for high-volume customers, and it significantly reduced noisy-neighbor issues without sacrificing platform agility.