Multi-tenant Knowledge Indexing Platform – System Design

- Author: Vinod Dhomse
- Role: Director, Platform Engineering Candidate

1. Overview

This document proposes the architecture for a multi-tenant knowledge indexing platform that supports enterprise customers ingesting large volumes of documents, indexing them efficiently, and retrieving them with fast, relevant search.

The platform is designed with four priorities:

1. Strong tenant isolation – every tenant's data must be logically protected and appropriately separated.
2. Performance at scale – handle high ingestion volume and large numbers of concurrent queries.
3. Enterprise readiness – authentication, authorization, encryption, auditing, and high availability.
4. Operational simplicity – a design that is scalable but not unnecessarily complicated.

2. Requirements Summary

Functional Requirements:

- Ingest documents (PDF, text, structured data), metadata, and tags
- Provide text/semantic search with ranked results and pagination
- Track audit events for compliance
- Tenant-scoped APIs:
  POST /api/v1/tenants/{tenantId}/documents
  GET /api/v1/tenants/{tenantId}/documents/search

Non-Functional:

- Scale – 100K documents/day ingestion, 10K concurrent queries
- Performance – P95 < 100ms (assignment scale)
- Reliability – Highly available, no SPOF
- Security – Isolation, encryption, auth
- Observability – health, metrics, logs

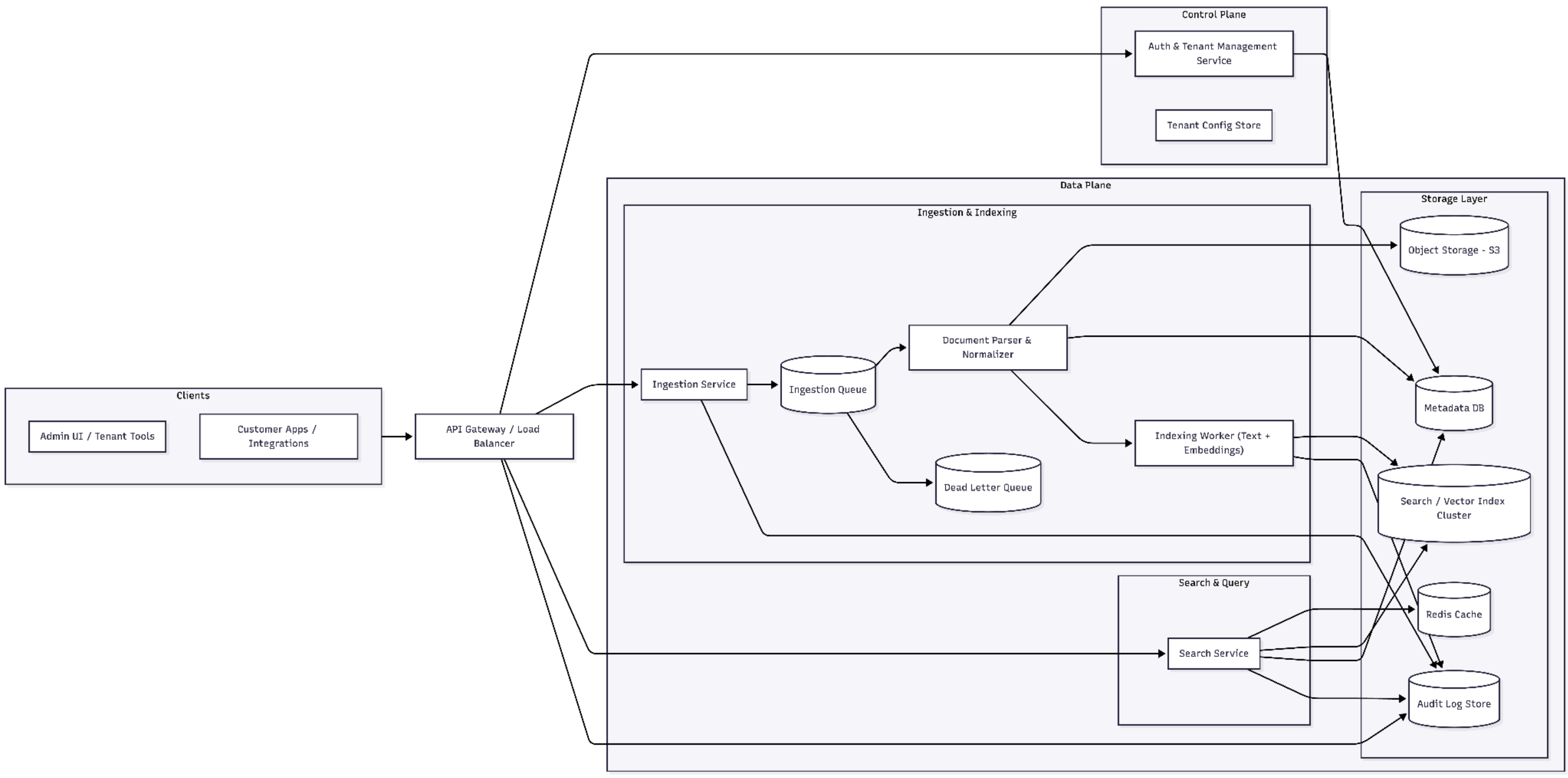
3. High-Level Architecture

The platform is structured into three planes:

- Control Plane – tenant management, authentication, configuration
- Ingestion + Indexing Plane – document intake, normalization, indexing
- Search Plane – fast retrieval and ranking

Documents are stored durably, indexed asynchronously, and retrieved via scalable stateless services and search layer.

Architecture Diagram



4. Multi-Tenancy Strategy

The system enforces tenant isolation across database, storage, search, and APIs.

Database: Shared relational DB with tenant_id mandatory in schema. Row-level security optionally enabled. Large tenants may move to dedicated DB without architectural change.

Object Storage for documents:
Tenant-namespaced storage structure (/tenant/document/version) with IAM enforcement.

Search: Support either shared index with per-tenant aliasing (default) or dedicated tenant index when necessary.

API Enforcement: Tenant-scoped URLs and authentication tokens linked to tenant identity.

5. Data Flow

Ingestion Flow:

1. Client submits document via API
2. Auth + authorization in API
3. Metadata stored in DB, raw file persisted in Object Storage
4. Message queued for asynchronous processing
5. Worker extracts item from queue + builds embeddings
6. Search index created
7. Record audit info in DB

Search Flow:

1. Client calls search API
2. Validate tenant
3. Execute tenant-scoped search
4. Use Redis cache to store search results for efficiency (optional)
5. Add Meta data info to results from DB
6. Return ranked results
7. Record audit info in DB

6. Scalability Strategy

Base strategy: Use horizontal scaling, distributed platforms, asynchronous processing and stateless services.

Ingestion:

- Distributed upload service (stateless)
- Async queue based processing of documents for search indexing
- Autoscaling workers for load distribution

Search:

- Distributed search services (stateless)
- Scalable search cluster
- Redis cache
- Sharding + replica strategy

Performance:

- Production: OpenSearch/Elastic
- Assignment: SQLite FTS sufficient

7. Security & Compliance

- **Authentication**: Use API key for this assignment, convert to OAuth/OIDC in real-world. API key header maps to tenant; middleware validates path tenantId matches key's tenant.
- **Authorization**: Tenant role based access control (RBAC) enforcement at service layer
- **Audit Logging**: what you log (ingest + search + admin actions) and retention (hot in DB, archive in S3).
- **Data encryption**: TLS in transit; at rest via RDS/S3/KMS in prod.

8. Reliability & Availability

- Multi-AZ services for API
- Multi-AZ High Availability database
- Replicated search clusters
- S3 object storage durability
- Blue/green deployment strategy ensuring zero downtime
- Circuit breakers + rate limiting (future)
- 100K/day is ~1.2 docs/sec avg; design scales for peaks via queue depth + worker autoscaling

9. Technology Rationale

- **FastAPI + Python**: rapid development, async model, clear structure
- **MySQL** (prod) / **SQLite** (assignment): strong tenant modeling
- **AWS OpenSearch** (prod) / **SQLite FTS** (assignment): mature full-text + vector.
- **S3**: durable storage
- **Redis**: caching + counters
- **Amazon SQS** (prod) / in-process queue (assignment): reliable distributed queuing in production, lightweight async processing for local execution

This stack balances capability and simplicity

Tradeoffs:

- **Shared DB + tenant_id**: cheaper/easier vs noisy-neighbor risk; option to move large tenants to dedicated DB.
- **SQS vs Kafka**: SQS simpler/managed vs Kafka better for streaming/replay (more ops).

- **OpenSearch vs SQLite FTS**: OpenSearch scalable/distributed vs SQLite FTS local/simple.
- **API-key vs OIDC**: API key simplest for assignment vs OIDC best for enterprise.

10. Summary

The proposed design delivers reliable ingestion, fast scalable search, strong tenant isolation, enterprise security posture, and operational simplicity while enabling future evolution to a production SaaS platform.