

Part 1 (Architecture & System Design)

- High-level architecture diagram
 - Frontend / API / Service / Data layers
 - Caching + Security + Observability
 - Scalability plan for 10k+ concurrent users + 500k+ articles
 - AI suggested answers using customer context
 - Trade-off analysis with at least 3 key architectural decisions
-

Part 1 — Architecture & System Design

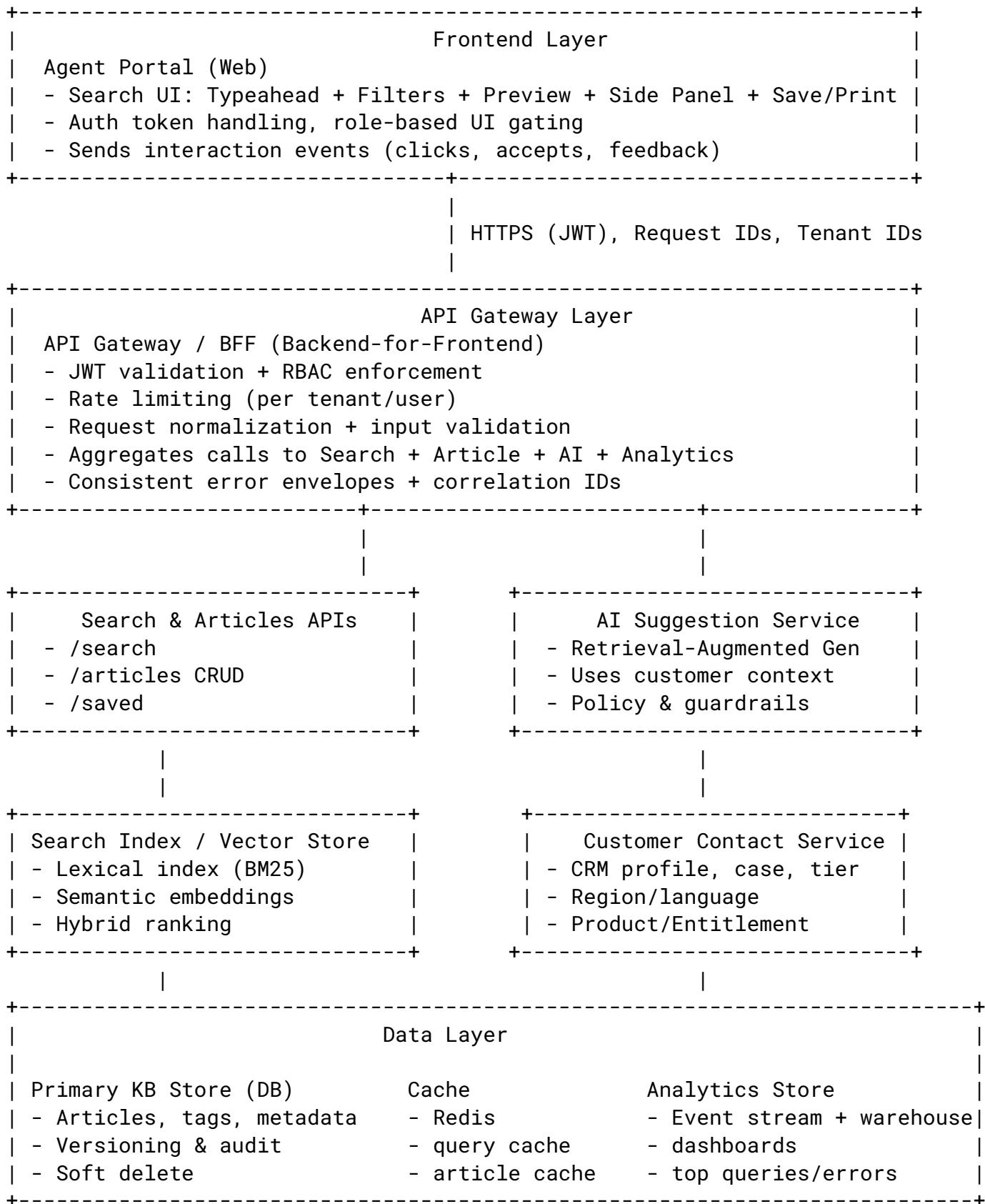
eGain Knowledge Search (AI-Powered) — System Design Proposal

Scenario

eGain needs a new **AI-powered knowledge search interface** for customer service agents. The system must:

- Search across **500,000+ knowledge articles** in near real-time
- Handle **10,000+ concurrent users** across multiple time zones
- Provide **AI-suggested answers** based on **customer context**
- Track **search analytics + agent interactions**
- Support **RBAC (role-based access control)**
- Integrate with **existing backend APIs**

High-Level Architecture (Layered)



Component Breakdown

1) Frontend Layer (Agent Portal)

Goals: fast search UX, predictable states, low-latency perception.

Key features:

- Typeahead suggestions (</search/suggestions>)
- Filters (category/tags/date), sorting, pagination
- Result preview cards + side detail panel
- Save article, print, open in new tab
- Interaction tracking (search executed, opened, saved, agent accepted/rejected)

Technology choices:

- **React** (fast iteration, component model)
- **TypeScript** (safer UI state + API contracts)
- Optional later: speech input, image attachments, agent co-pilot panel
- Personalization: the end user personalize the UX

2) API / Service Layer (BFF + Domain Services)

BFF responsibilities

- JWT validation and RBAC enforcement
- Rate limiting, throttling (per-tenant/user, burst control, cost-based limits)
- Request shaping (payload caps, pagination bounds)
- Routing + API governance (service dispatch, version routing, schema validation)
- Centralized security controls
- One consistent response format + error envelope
- Aggregation: search + article fetch + AI suggestion + analytics ingest

Domain services

- Search service (hybrid lexical + semantic)
- Articles service (CRUD + versioned retrieval)
- Saved service (per-user list)
- Analytics service (high-volume event ingestion)
- AI suggestion service (RAG, customer-context aware)

3) Data Layer

- **Knowledge store** (source of truth): relational DB or document DB
 - Supports versioning, soft-delete, tags, category, timestamps
- **Search index**: hybrid approach
 - Lexical index (BM25) for precision and exact matches
 - Vector store for semantic retrieval
 - Hybrid ranker merges both
- **Cache (Redis)**
 - Query cache for hot searches
 - Article cache for frequently opened articles
 - Suggestions cache (very high QPS)
- **Analytics store**
 - Event stream + warehouse/dashboard tables
 - Used for top queries, error rate, agent adoption metrics

API Design Principles

1. Consistency

- Standard headers: `X-Request-Id`, `X-Tenant-Id`
- Standard error envelope:

```
{ "error": { "code": "...", "message": "...", "requestId": "..." } }
```

2. Versioning

- `/api/v1` current stable
- `/api/v2` for breaking changes only
- Additive changes remain in v1

3. Pagination

- Page-based pagination for predictable UI and caching:

```
"pagination": { "page": 1, "pageSize": 20 }
```

4. Rate limiting

- Per tenant + per user
- Separate budgets for:
 - suggestions (high QPS)
 - search (moderate)
 - analytics ingest (high volume batch)

- Return 429 with `Retry-After` and `X-RateLimit-*` headers

Scalability Approach (10k+ Concurrent Users)

Key strategies

- Stateless API services behind load balancers (horizontal scaling)
- Cache aggressively for hot queries/articles/suggestions
- Use hybrid search index optimized for reads
- Make analytics ingestion **async** with batching and idempotency
- Keep AI generation optional and decoupled from core search latency

Performance targets (example)

- Suggestions p95 < 150ms (cached)
- Search p95 < 600ms (hybrid, cached hot queries)
- Article fetch p95 < 250ms (cache first)
- AI suggestion p95 < 2500ms (async/streamed), not blocking search results
- Expensive CRM user profile data (cached)

Performance Optimization Strategies

- **Caching**
 - Redis for query + article + suggestions
 - Cache key includes tenant + query + filters + sort + page
- **Index optimization**
 - Precompute embeddings offline / incremental updates
 - Use hybrid ranking rather than “vector-only”
- **Backpressure**
 - Rate limit + queue analytics ingestion
 - Circuit breaker AI dependency (fallback to top KB hits)

Security Implementation

Authentication

- JWT bearer auth (`Authorization: Bearer <token>`)
- Signed tokens; short expiry; rotate secrets

Authorization (RBAC)

- Roles in JWT claims (viewer/editor/admin)
- Enforced at BFF + service

- “Least privilege” defaults

Data safety

- Tenant isolation (tenantId scoping in queries + cache keys)
- Policy agent to guard what tools, APIs can be called based on the user profile and scopes
- Audit logs for article edits/deletes
- Soft delete by default, hard delete restricted

Abuse prevention

- Rate limiting per tenant/user
- Payload size limits for analytics events
- Validation for all inputs (query, filters, ids)
- CSRF, Allowed Websites

Monitoring & Observability

What we measure

- Search latency (p50/p95/p99), suggestion latency, AI latency
- Error rates by endpoint + code (4xx vs 5xx)
- Top queries, zero-result queries
- Agent interaction funnel:
 - suggestions shown → clicked → accepted/rejected
 - answers generated → copied → feedback

Implementation

- Structured logs with requestId + tenantId + userId
- Metrics (Prometheus/OpenTelemetry style) + dashboards
- Traces across BFF → search → AI → analytics
- Alerting:
 - p95 latency regression
 - spike in 5xx
 - spike in zero-results (index or ingestion issue)

Trade-off Analysis (Key Architectural Decisions)

Trade-off 1: Page-based vs Cursor-based Pagination

Decision: page-based for v1.

Why: simpler UI + easy caching + predictable analytics aggregation.

Cost: deep paging can be slower/less stable if dataset changes rapidly.

Mitigation: cap page depth; for v2, consider cursor-based for large/rapidly changing corpuses.

Trade-off 2: Hybrid Search (BM25 + Vectors) vs Vector-only

Decision: hybrid retrieval + ranking.

Why: customer support knowledge often needs exact term precision (“SSO”, error codes), while semantic helps with paraphrases. Hybrid wins on both.

Cost: more infra complexity (two indexes + rank merge).

Mitigation: keep BM25 as primary and add vectors incrementally; maintain clear evaluation metrics (NDCG, recall@k).

Trade-off 3: AI Answers Synchronous vs Decoupled

Decision: decouple AI answer generation from core search results.

Why: agents must always get fast results even if AI is slow/unavailable.

Cost: more orchestration (async job, streaming, or background generation).

Mitigation: show KB results immediately; AI panel loads progressively; fallback to “top articles + highlights.”

Trade-off 4: Real-time Index Updates vs Batch

Decision: near real-time incremental updates for critical articles; batch for bulk.

Why: reduces operational load while still allowing urgent KB updates.

Cost: two ingestion paths.

Mitigation: unify ingestion interface with priority flags.

Technology Stack Choices

- **Frontend:** React + TypeScript (fast UX iteration, safe state management)
- **BFF/API:** Node.js/Next.js API routes or dedicated service (rapid development + easy Vercel deploy)
- **Cache:** Redis (low-latency shared cache, rate limit counters)
- **Search:** BM25 engine + vector DB (hybrid search quality)
- **Analytics:** event ingestion + warehouse tables (dashboards, funnels, top queries)
- **Observability:** OpenTelemetry + metrics + tracing (end-to-end visibility)

Summary

This architecture meets the requirements by:

- Delivering **fast, reliable search** at scale (500k+ articles, 10k+ concurrent agents)
- Supporting **secure, role-aware access** (JWT + RBAC + tenant isolation)
- Providing **AI suggestions grounded in customer context** (RAG + context service)
- Enabling **analytics + observability** as first-class features (events + dashboards + tracing)
- Maintaining **clear API design contracts** with versioning, pagination, and consistent errors