SYSTEM DESIGN CAPSTONE · COHORT 4

Multi-Tenant Enterprise RAG Platform

10M	1B+	100K	<3s
daily active users	indexed chunks	isolated tenants	P95 full answer

Group Members:
Zeeshan Hamid
Moneebb Yasin Chaudhry

OVERVIEW

What we are building

- **The product:** a shared question-answering platform that enterprises plug their internal knowledge into. Each company is a tenant, and each tenant connects the sources its teams already use: SharePoint, PDFs, wikis, Slack exports.
- **How it feels to use:** an employee asks a question in plain language and gets an answer built only from their own company's documents, with citations linking every claim back to the exact source passage.
- **It respects the org chart:** access is role-based, so an intern and a VP asking the same question can get different answers, because they are allowed to see different documents. Conversations are multi-turn, so follow-ups keep their context.
- **It handles a living corpus:** enterprise documents change constantly. Updates re-index incrementally, and deletions propagate to every index and cache, so a removed document stops answering questions.
- **Two hard problems shape every decision:** first, tenant isolation is structural. Data partitioning itself guarantees one company never sees another's data, instead of trusting a filter clause. Second, the dominant cost is LLM tokens, not servers, so caching and context control are core architecture rather than afterthoughts.
- **Where it came from and where it goes:** this generalizes a single-tenant SharePoint RAG build into a platform sized for 100,000 tenants, 10 million daily users, and over 1 billion indexed chunks.

REQUIREMENTS

Functional & Non-Functional Requirements

Functional

- Tenants upload / connect document sources (SharePoint, PDF, wikis, Slack exports)
- Chunk, embed, and index per tenant, strict cross-tenant isolation
- Natural-language Q&A grounded in tenant documents, with citations
- Multi-turn conversations, follow-ups retain context
- Admin-managed permissions; role-restricted documents
- Incremental re-indexing on update; deletion propagates to every index and cache

Non-Functional

- Latency: P95 first token < 1 s · P95 full answer < 3 s (budgeted per stage)
- Tenant isolation: zero cross-tenant leakage, the standout NFR
- Permissions: strongly consistent at query admission (one-request revocation window, stated)
- Indexing: eventual, searchable within minutes
- High availability on the query path; ingestion degrades to backlog, never data loss
- Scale: 100M+ docs, 10M DAU, +20%/year · Cost: LLM spend bounded & measurable

ASSUMPTIONS & SCALE

Who uses it, how much data, how hard it gets hit

100M users · 10M DAU

10% daily-active, realistic for a B2B tool used during work hours

100K tenants

~1,000 users each on average, sizes are skewed; a few “whale” tenants hold millions of docs

100M docs → 1B chunks

~10 chunks per document at Year 1, growing 20%/year

Clears the assignment scale floor (≥ 20K QPS or ≥ 10M DAU) on all three counts:

10M DAU

30,371 peak read QPS

37,454 peak write QPS

- 20 queries / active user / day -> 200M queries/day. Enterprise load concentrates in ~8 business hours -> 6,940 QPS average.
- Peak-within-peak × 2.5 -> 17,350 QPS peak query submissions (Year 1), the number that sizes the retrieval pipeline.
- Ingestion is write-light: 0.5% of corpus/day ≈ 6 docs/sec, a read-heavy query path and an async ingestion path, by the numbers.

QPS ESTIMATION

Per-action load

Read action	Total/day	Avg QPS	Peak ×2.5
Submit query	200M	6,940	17,350
Open past conversation (UI)	100M	3,472	8,681
View citation / source doc	50M	1,736	4,340
Total reads	**350M**	**12,148**	**30,371**

Write action	Vol/day	Avg QPS	Peak ×2.5
Store message (Q + A = 2 rows)	400M	13,889	34,722
Create conversation	30M	1,042	2,604
Document ingestion (24h)	500K	5.8	~12
Permission / metadata updates	5M	58	~116
Total writes		**~14,995**	**~37,454**

What the breakdown surfaces

- Message writes are the largest QPS figure in the system (34.7K peak Y1, ~72K Y5), bigger than query reads. This justifies Cassandra.
- History reads and citation views never touch the retrieval funnel, only “submit query” drives embed / search / rerank / LLM sizing.
- Hidden internal read: every query loads recent turns before the cache check -> 17,350 QPS (Y1) on Cassandra, sized on the scaling slide.

GROWTH ESTIMATES

Uniform 20%/year across users, documents, and queries

Year	DAU	Docs	Chunks	Queries/Day	Avg QPS	Peak QPS ×2.5
1	10.00M	100.0M	1.00B	200.0M	6,940	17,350
2	12.00M	120.0M	1.20B	240.0M	8,328	20,820
3	14.40M	144.0M	1.44B	288.0M	9,994	24,984
4	17.28M	172.8M	1.73B	345.6M	11,992	29,981
5	20.74M	207.4M	2.07B	414.7M	14,391	**35,977**

Why one growth knob

A single 20% rate keeps every derived number internally consistent; independent per-dimension rates would each need evidence that doesn't exist at design time. Stated as a simplification, not hidden.

Peak crosses 20K QPS in Year 2 and reaches ~36K by Year 5; this line decides when each layer must scale out.

STORAGE

One engine per access pattern

Data	Engine	Why this engine
Tenant / User / Document / ACL groups / Chunk metadata	PostgreSQL	ACID for permission changes; B-tree indexing fits equality + range access
Chunk embeddings (768-dim)	Vector store (HNSW)	B-trees compare ordered scalars — similarity search needs a graph-based ANN index
Keyword / exact-match index	BM25 (OpenSearch)	Catches exact tokens (error codes, SKUs, acronyms) that dense embeddings miss
Conversations / Messages	Cassandra	400M append-only rows/day, always read as one time-ordered range — LSM fits, B-tree fights
Raw blobs · cold messages · snapshots	Object storage (S3-class)	Elastic, provider-managed; serves citation views via short-lived signed URLs
Document parsing (PDF/Office/HTML)	Apache Tika	One format-agnostic extractor for every FR source type — not a bespoke parser per format

Decision pattern: the engine follows the access pattern, relational where ACID matters, ANN where geometry matters, LSM where write volume matters.

STORAGE REQUIREMENTS

1 Year Calculations

Row sizes × row counts (Year 1)

- tenant: 172 B × 100K ≈ 17 MB
- user: 268 B × 100M ≈ 26.8 GB, B-tree (tenant_id, last_active_at)
- document: 792 B × 100M ≈ 79 GB, B-tree (tenant_id, status, updated_at)
- ACL groups + memberships: ~300M small rows ≈ 12 GB, clustered on user_id
- chunk_metadata: 3,064 B × 1B ≈ 3.06 TB, BRIN on doc_id (a B-tree here costs ~400 GB alone)
- Vectors: 3.1 TB raw × 2 (HNSW graph) × 3 (RF) ≈ 18.6 TB
- BM25: 2.6 TB × 3 ≈ 7.8 TB · Messages: 855 GB/day raw

Store	Year 1
PostgreSQL (metadata)	3.19 TB
Vector store (×3)	18.60 TB
BM25 (×3)	7.80 TB
Messages (tiered 30d hot + 60d cold)	93.70 TB
Platform total	**123.3 TB**

Tiered retention pays: 30-day hot (Cassandra RF=3) + 60-day cold (4:1 compressed, erasure-coded) = 93.7 TB vs 231 TB flat, a 59% cut on the platform's largest storage line.

STORAGE GROWTH

5-year calculations

Year	PostgreSQL	Vector	BM25	Messages	Platform
1	3.19 TB	18.60 TB	7.80 TB	93.70 TB	123.3 TB
2	3.83 TB	22.32 TB	9.36 TB	112.44 TB	147.9 TB
3	4.59 TB	26.78 TB	11.23 TB	134.93 TB	177.5 TB
4	5.51 TB	32.14 TB	13.48 TB	161.92 TB	213.0 TB
5	6.61 TB	38.56 TB	16.18 TB	194.30 TB	**255.6 TB**

Key finding

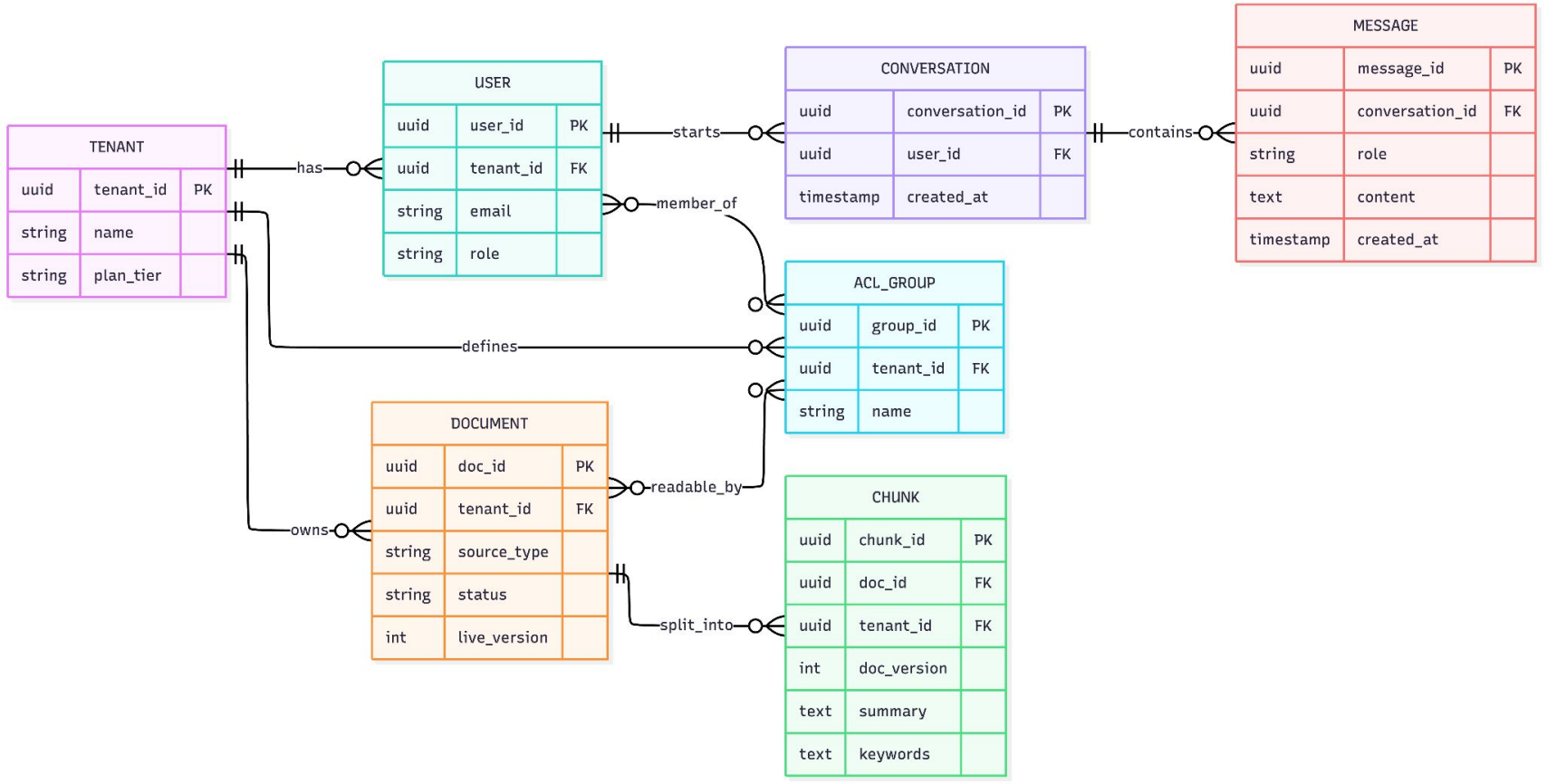
Message history dominates storage every year, not embeddings. The intuitive worry (“vectors will explode”) is wrong for this system. PostgreSQL grows only 3.19 -> 6.61 TB in five years and never needs sharding; the vector store, BM25, and Cassandra are the only horizontal-scaling stores. Messages are a rolling 90-day window (per-tenant retention policy), so each year's figure is steady state.

ENTITIES

ER Diagram, ACL groups replace per-user permission rows

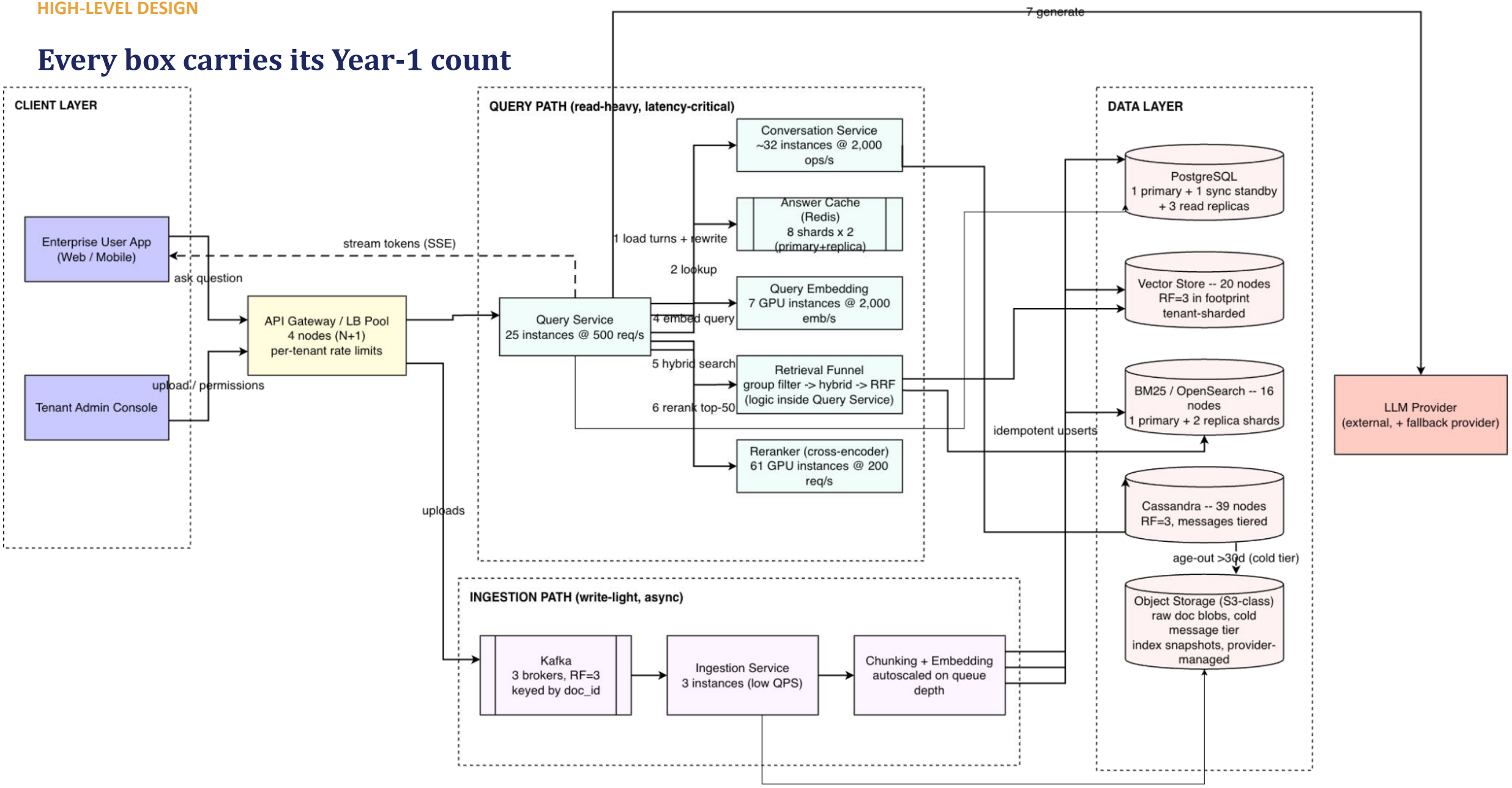
Entities

- Tenant
- User
- Conversation
- Message
- ACL Group
- Chunk
- Document



Why ACL groups, not PERMISSION(doc_id, user_id) rows: a whale-tenant user with access to 4M of 5M docs would need 4M doc_ids as a search filter, not feasible. Documents carry a short list of group tags indexed in the search stores; the query path resolves the user's few group memberships instead. Strong permissions, tiny filter payloads, any tenant size.

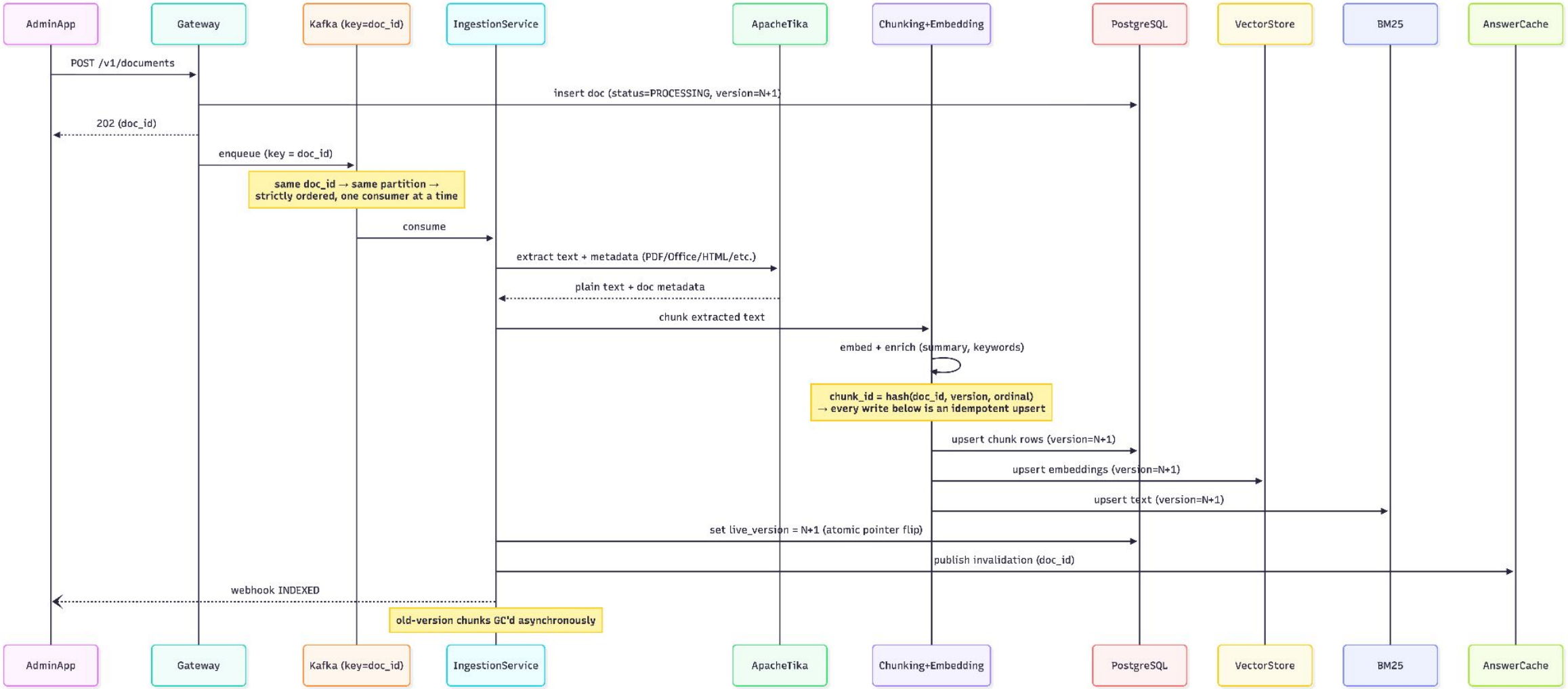
Every box carries its Year-1 count



View Diagram here

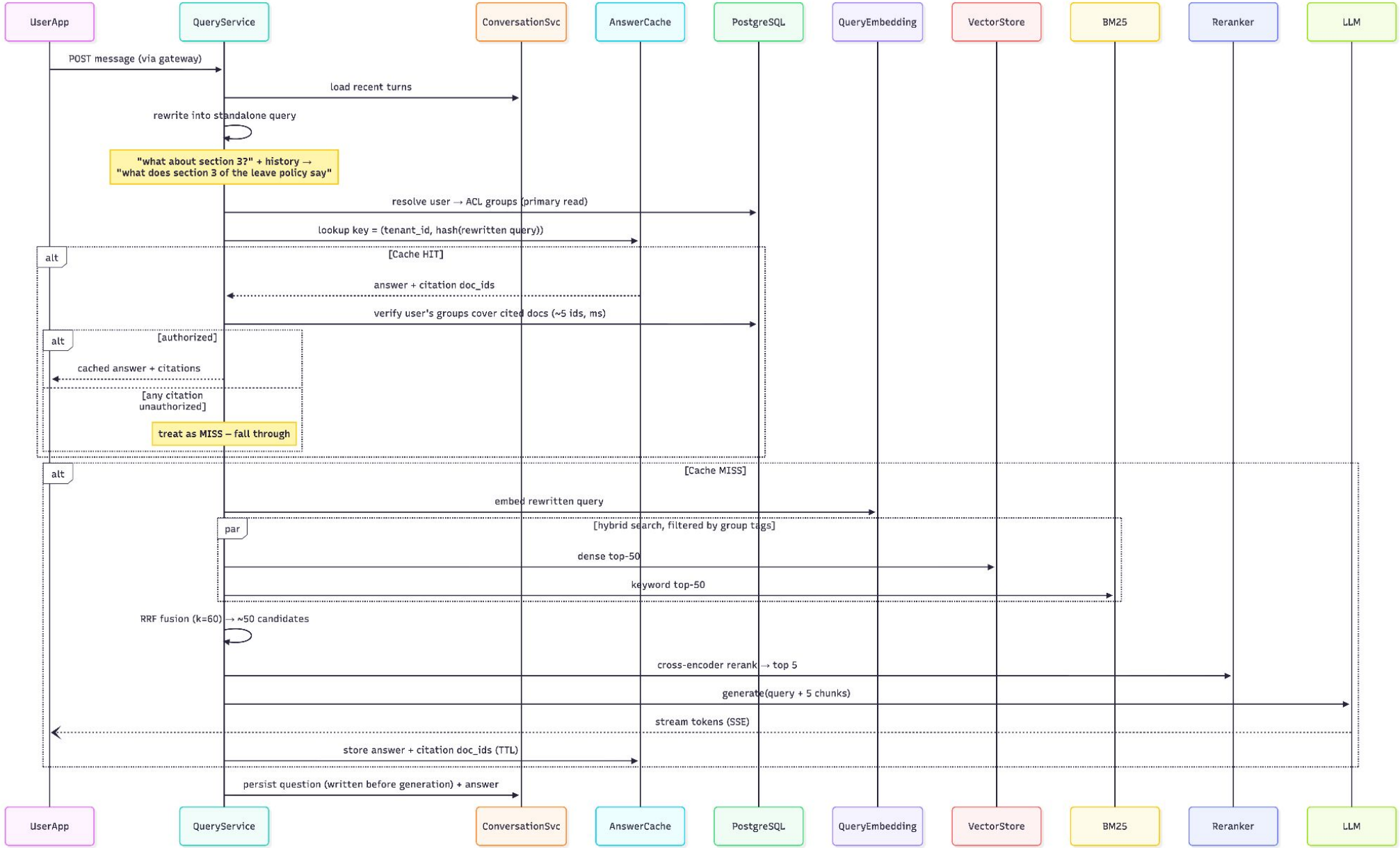
Flow Diagram

Data Ingestion



Flow Diagram

Query and Retrieval



API DESIGN

Versioned /v1 — tenant_id comes from the JWT, never the client

Ask a question (SSE stream)

```
POST /v1/conversations/
     {id}/messages
Auth: Bearer <jwt>
{ "content": "What is our
  parental leave policy?" }

→ 200 text/event-stream
data:{"delta":"Full-time..."}
data:{"done":true,
 "citations":[{doc_id,
  chunk_id,title,page}]}
```

Upload document (202 async)

```
POST /v1/documents
{ "source_type": "pdf",
  "title": "HR Policy 2025",
  "acl_groups":
   ["hr-all","managers"] }
→ 202 { doc_id,
   status: PROCESSING }

GET /v1/documents/{id}
→ { status: PROCESSING |
  INDEXED | FAILED, live_version }
```

Permissions & deletion

```
PATCH /v1/documents/{id}/
      permissions
{ "acl_groups": ["hr-all"] }
→ 200

DELETE /v1/documents/{id}
→ 202 (async purge across
  PG, vectors, BM25, cache)

POST /v1/conversations → 201
GET  .../messages?limit=50
```

202 on writes matches the NFRs: clients never block on embedding compute — status is polled or pushed via webhook.

FLOW 1 · INGESTION

Upload → parse → index — with three races designed out

1. Upload

202 + doc row (v = N+1)

2. Kafka

key = doc_id → per-doc ordering

3. Tika parse

any format → text + metadata

4. Chunk + enrich

embed · summary · keywords

5. Upsert ×3

PG · vectors · BM25 (v = N+1)

6. Pointer flip

live_version = N+1 · invalidate cache

Race: two updates to one doc

Kafka key = doc_id serializes updates per document; different docs still run in parallel. Index never holds interleaved v1/v2 chunks.

Race: crash + Kafka redelivery

chunk_id = hash(doc_id, version, ordinal) → every write is an idempotent upsert. Retries converge instead of duplicating chunks.

Race: partial writes across 3 stores

No distributed transaction. New-version chunks are invisible until one atomic PG pointer flip; readers never see a mixed state. GC sweeps the rest.

Deletion runs the same machinery: PG tombstone (instantly invisible) -> cache invalidation -> async purge of BM25, vector tombstones + HNSW compaction, chunk rows. Tenant offboarding = same purge scoped by tenant_id.

FLOW 2 · QUERY

The retrieval funnel, cache skips the expensive path, never the security path

1. Load turns + rewrite

“what about section 3?” → standalone query (every query, ~150 ms)

2. Resolve ACL groups

PG primary, strongly consistent

3. Cache lookup

key = tenant + hash(rewritten query); HIT → verify cited doc_ids vs user's groups → serve (~180 ms)

4. Embed + hybrid search

dense top-50 // BM25 top-50, filtered by group tags

5. RRF fusion → rerank

k=60 merge → cross-encoder picks top 5

6. Generate + stream

LLM over 5 chunks → SSE tokens → store answer + citations

Why this ordering is the security design

- Rewrite before cache: the same words mean different things in different conversations, keying on the rewritten query stops cross-conversation answer leaks and raises hit rate.
- Citation-verified hits: a tenant-keyed cache stops cross-tenant leaks but not intra-tenant ones (VP's answer served to an intern). Every hit re-checks its ~5 cited doc_ids — milliseconds.
- Event-driven invalidation on update / delete / permission change; TTL is only the backstop. 30% hit rate is a stated assumption with a 15% sensitivity case.

PERMISSION MODEL

ACL group tags. This is where naive RAG designs fail at scale

1 Documents carry group tags

e.g. [“hr-all”, “managers”] written into the vector store and BM25 as filterable chunk metadata at ingestion.

2 Queries resolve memberships

The user's few group IDs come from the PG primary, strongly consistent, a point lookup on a ~12 GB hot table.

3 Search filters on the tags

Vector + BM25 filter on tenant_id + groups. Payload stays tiny whether the tenant has 100 docs or 5 million.

Consistency split, stated precisely

- Membership changes (someone leaves the company) take effect on the next query, resolved fresh from the primary every time. Strong where it matters most.
- Document retag events propagate to the search indices within minutes (same bound as indexing), with cache invalidation fired immediately.

Filter before search — never post-filter

Group filtering executes inside the vector/BM25 query, so an unauthorized chunk never appears in any candidate list, the production pattern where permissions propagate into the retrieval engine, not a UI-layer check. Post-filtering briefly holds unauthorized chunks in memory and silently returns fewer than k results.

RETRIEVAL ENGINE

How 1 billion chunks become the 5 the model reads

1. Scope

1B chunks ⟶ one tenant + ACL groups

2. Dense search

HNSW top-50, meaning-based

3. Keyword search

BM25 top-50, exact tokens, in parallel

4. RRF merge

rank-based fusion, k = 60

5. Rerank

cross-encoder scores 50, keeps 5

6. Generate

LLM reads 5 chunks, streams the answer

Why two search arms

Embeddings understand meaning but miss exact tokens like error codes and SKUs. BM25 catches those. Each arm covers the other's blind spot, the same hybrid pattern Salesforce runs in production.

Why merge by rank, not score

Cosine similarity and BM25 scores live on incomparable numeric scales. RRF fuses the two lists by rank position, so it needs no re-tuning as data grows and score distributions drift.

Why a reranker exists at all

It is a cost gate. Sending 5 precise chunks instead of 20 loose ones cuts LLM input tokens by 4x, the platform's largest cost line. The GPU pool pays for itself roughly 100 times over.

Enrichment feeds the funnel: every chunk also carries an LLM-written summary, keywords, and hypothetical questions, written once at ingestion -> a vague question can match a chunk whose raw text never used those words -> recall improves before a single query-time token is spent. Group filtering always runs inside the search itself, never as a post-filter.

KEY DECISIONS

Chosen approaches with trade-offs

Decision	Why	Trade-off accepted
HNSW vector index	O(log N) approximate NN vs O(N) brute force at 1B+ chunks	Small tunable recall loss (ef_search); RAM-priced nodes
Hybrid search + RRF (k=60)	Embeddings miss exact tokens; rank-based fusion needs no re-tuning as scales drift	One extra store (OpenSearch) to operate and keep in sync
Rerank top-50, not top-100	Halves the priciest GPU pool (126 vs 252 instances @ Y5)	Tail queries may lose a candidate ranked 51–100
Tenant-sharded vectors + whale sub-shard	Isolation becomes structural; shard failure blast radius = a tenant list	Whale queries fan out to a few shards and merge
BM25: shared indices + routing	100K indices would blow OpenSearch cluster state (ceiling: a few thousand)	Shared tenants are one filter apart; whales get dedicated indices
Cassandra (conversation_id, created_at)	34.7K⟶72K writes/s; thread = one pre-sorted partition read	Weak at ad-hoc queries, messages are never queried that way
Apache Tika extraction	One parser for every FR source format instead of four bespoke ones	Complex table layouts occasionally lose structure; enrichment recovers most

SHARDING

Do we need database sharding?

Store	Sharded?	Shard key	What forces it
Vector store	**Yes — day one**	tenant_id (consistent hashing) + chunk_id sub-shard for whales	18.6 → 38.6 TB of RAM-resident HNSW can't live on one node
BM25 / OpenSearch	**Yes — day one**	index routing by tenant_id	7.8 → 16.2 TB; shared indices dodge the cluster-state ceiling
Cassandra	**Yes — inherent**	conversation_id (partition key IS the shard key)	77 → 160 TB hot + 34.7K → 72K writes/s
Redis cache	**Yes**	hash slots, 8 shards × 2	~420 GB working set exceeds any sensible node
PostgreSQL	**No — deliberate**	—	6.6 TB @ Y5; hot ACL set ~12 GB in RAM; ~25K point reads/s fits one primary

- Query load never forced a single sharding decision in this platform, data volume forced four (the storage-bound finding).
- Every shard key does double duty: tenant_id = the isolation boundary; conversation_id = the access pattern. Security model and read pattern physically meet here.
- PostgreSQL is a decision, not an omission: revisit trigger named (primary saturation / ~20+ TB), exit pre-paved, every table already carries tenant_id, so Citus-by-tenant is schema-compatible.

SCALING & LOAD

Every number derived - Year 1 and Year 5 side by side

Component	Sizing basis	Year 1	Year 5	Bound by
Pipeline QPS after 30% cache	peak × 0.7	12,145	25,184	—
Query Service	500 req/s per instance	25	51	throughput
Query Embedding (GPU)	2,000 embeds/s per GPU	7	13	throughput
Reranker (GPU)	200 req/s = 10K pair-scores/s (top-50, fp16, batch 64)	61	126	throughput
Vector store	1 TB per shard	20	39	**storage (QPS alone: 3→6)**
BM25	0.5 TB per node	16	33	**storage (QPS alone: ~3→6)**
Cassandra	2 TB per node, hot tier	39	80	**storage (writes alone: 18→36)**
PostgreSQL	vertical + 3 replicas + sync standby	3.19 TB	6.61 TB	neither. Stays vertical

Key finding: the data layer is storage-bound, not throughput-bound. Every store needs 2–6× more nodes for capacity than query load alone would demand; throughput headroom comes free. Cassandra's reads (26K ⟶ 54K/s incl. per-query turn-loads) land at ~670 reads/s/node — trivial at CL=ONE on row-cached hot partitions.

LATENCY BUDGET

The 3-second NFR

Stage (cache-miss path, P95)	Budget
Gateway + auth	10 ms
Load recent turns + rewrite (small model)	150 ms
Cache lookup (miss)	2 ms
ACL group resolution (PG primary)	5 ms
Query embedding (GPU, batched)	15 ms
Hybrid search (vector // BM25, parallel)	80 ms
RRF fusion + rerank 50 candidates	61 ms
Pre-generation total	**~325 ms**

First token < 1 s ✓

325 ms pipeline + ~500 ms LLM time-to-first-token

Full answer < 3 s ✓

+ ~2,000 ms generation → ~675 ms of headroom

Cache hit ≈ 180 ms

Dominated by the rewrite step, which runs on every query by design — the cache skips the expensive steps, never the cheap or the security ones

The reranker is the stage to watch: last synchronous step before the LLM — any queueing there eats generation headroom one-for-one.

FAILURE SURFACE

Bottlenecks vs single points of failure — different problems, different fixes

Bottlenecks → add capacity

- Reranker is heaviest synchronous hot-path stage (126 Y5 instances vs 51 Query Service); under-provisioning inflates P95 directly.
- LLM provider is the one capacity we don't own; most likely visible degradation point under a spike.
- Chunk + Embed is heaviest ingestion compute; the Kafka queue turns a bulk-upload spike into a backlog, not an outage.

SPOFs → add redundancy (all four named)

SPOF	Mitigation
LLM provider	Circuit breaker + fallback provider; last resort = chunks-only answer
PostgreSQL primary	Sync standby, ~30 s managed failover; fails closed (never serves unauthorized data)
API Gateway	Redundant pool behind DNS/anycast — one box on the diagram, never one node
Kafka	3 brokers, replicated partitions, built-in redundancy, stated not assumed

Reranker and Chunk+Embed are bottlenecks but NOT SPOFs — stateless pools where one instance failing reduces capacity, not availability.

DURABILITY & RECOVERY

Replication, consistency, and backup for every store

Store	Replication	Consistency	Backup
PostgreSQL	**primary + sync standby + 3 read replicas**	permission reads and writes hit the primary only; fails closed during failover (~30 s)	continuous WAL archive + daily snapshot; point-in-time restore
Vector store	**RF = 3**	writes at quorum; reads eventual, latency first	nightly snapshot to object storage
BM25 / OpenSearch	**1 primary + 2 replica shards**	writes to the primary, async replication; reads from any replica	daily snapshot API to object storage
Cassandra	**RF = 3, peer-to-peer ring**	writes QUORUM, reads ONE	cold tier doubles as the archive; no separate backup job
Redis answer cache	**8 primaries + 8 replicas**	best effort by design; a cost optimization, never a correctness dependency	none needed; entries rebuild from live traffic

- Consistency is spent where it buys safety, strong on the permission path, eventual everywhere latency matters more.
- Restore story per store: PostgreSQL restores from WAL to any point in time; search indices restore from snapshots, then re-converge from PostgreSQL, the source of truth for chunk metadata.
- The cheapest backup is the one already paid for, the Cassandra cold tier is simultaneously the retention policy, the archive, and the backup.

RACES, NAMED

Systems fail in the seams

Failure / race	Without mitigation	Design answer
Cache hit bypasses permissions	Intern gets an answer from a VP-only doc	Every hit re-verifies cited doc_ids vs the requester's groups
Same words, different conversations	Conversation B served A's contextual answer	Cache keyed on the rewritten standalone query
Concurrent re-index of one doc	Interleaved v1/v2 chunks in the index	Kafka key = doc_id + versioned chunks + atomic pointer flip
Crash mid-ingestion / redelivery	Duplicated chunks double retrieval weight	Deterministic chunk IDs ⟶ idempotent upserts converge
Redis cluster failure	36K QPS hits a 25.2K-sized pipeline (−43%)	Gateway admission control sheds load while the reranker autoscales
Revocation mid-generation	One in-flight answer under revoked access	Accepted + stated: bounded to one request; cache afterlife closed by event invalidation

COST MODEL

Additional: Where the money goes - (OUT OF SCOPE)

$2.5M/mo

LLM generation, Year 1 (API mini-tier): 140M cache-miss calls/day × 2K in + 500 out tokens

≈ 10× all infrastructure combined ($200–250K/mo)

Biggest levers (Y1)

- Answer cache @30%: −$1.1M/mo vs ~$10K of Redis
- Top-5 context: −$126K/day vs naive top-20 stuffing
- Tiered retention: 137 TB of hot storage not bought

	A · API only	B · Self-host 8B	C · Hybrid ★
Y1 / Y5 monthly	$2.5M / $5.2M	~$750K / ~$1.55M	**~$1.0M / ~$2.1M**
Quality	Best models on tap	Capped at 8B strengths	8B bulk + API escalation (~10%)
Data boundary	3rd party (zero-retention)	Fully in-house — a sales feature	In-house for ~90%
SPOF posture	External provider	Own fleet	Each side is the other's fallback

Decision: phase it

- Phase 1 (launch → ~3M DAU): pure API — below the crossover, self-hosting pays for idle GPUs. Telemetry accumulates.
- Phase 2: bulk traffic to self-hosted 8B (~1,000 H100s at peak, ~40% avg fleet) with API escalation — a ~60% cut, ~$18M/yr at Y1 scale.
- 70B self-hosting loses to the API on arithmetic (~8,000 GPUs) — the top-5 funnel is what makes the small model viable.

OPERATE & PROTECT

Observability, tenant fairness, and isolation tested in production

Per-tenant rate limiting

Token bucket at the gateway, quota by plan tier. A noisy neighbor becomes a 429 for that tenant, not P95 degradation for everyone. Doubles as the monetization enforcement point.

Metrics that test the design's own claims

Per-stage latency histograms matching the budget table · cache hit rate per tenant (the 30% assumption is monitored, not trusted) · reranker queue depth as the autoscaling signal · ingestion lag vs the “minutes” NFR.

Cross-tenant isolation canary

Synthetic tenants continuously query for each other's planted documents; any hit pages immediately. Isolation is tested in production, not assumed from design.

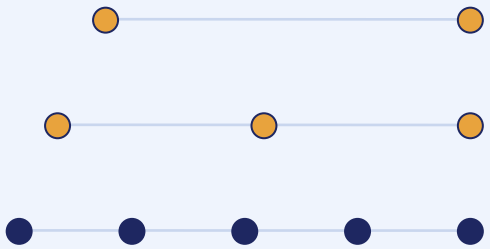
One trace ID from gateway through rewrite → retrieval → rerank → generation, tagged with tenant_id — per-tenant cost attribution becomes a query, not a project.

Deliberately out of scope: multi-agent orchestration (no FR/NFR moves) and multi-region (single-region at launch; tenant-sharding extends naturally to region-per-tenant for data residency).

APPENDIX

Algorithms under the hood

HNSW



sparse top layers → dense base: search descends greedily

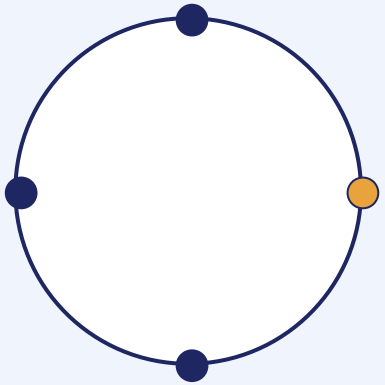
- O(log N) approximate nearest-neighbor vs O(N) brute force at 1B chunks
- Recall ↔ latency tunable via ef_search

Reciprocal Rank Fusion

$$\text{score}(d) = \sum_{k=60} \frac{1}{(k + \text{rank}(d))}$$

- Merges the dense and BM25 result lists by rank, not by score
- Cosine similarity and BM25 scores live on incomparable numeric scales — rank-based fusion needs no re-tuning as distributions drift with data growth
- Discards score magnitude — acceptable: the cross-encoder re-scores the merged top-50 immediately after

Consistent Hashing



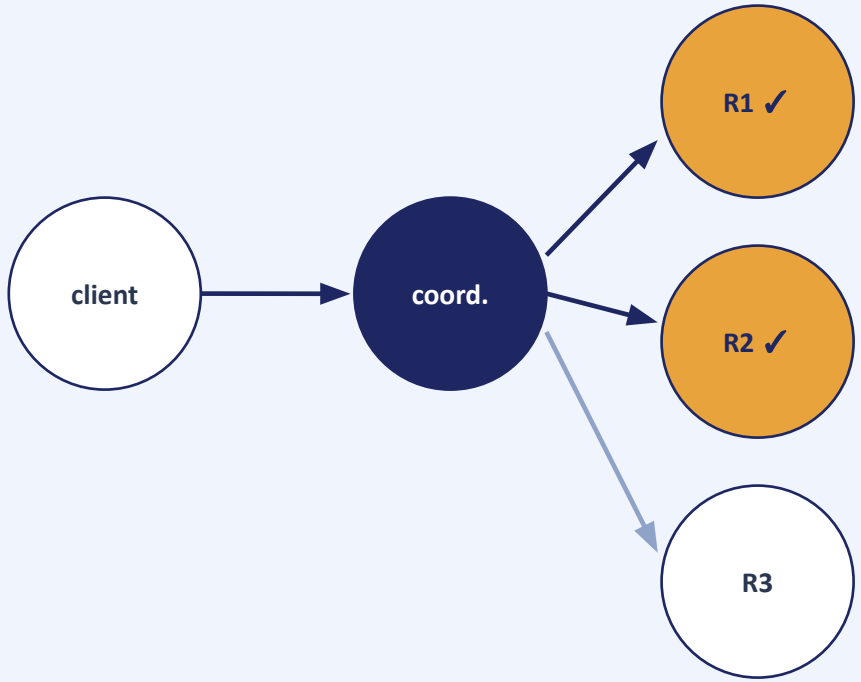
tenants hash onto a ring of vector-store nodes

- Adding / removing a node remaps only the affected ring segment
- Plain modulo (tenant_id % N) would reshuffle nearly every tenant on any node change
- Whale tenants (>1M chunks) sub-shard by chunk_id across the ring

APPENDIX

Cassandra replication & the tiered-retention math

RF = 3 · write QUORUM · read ONE



Write acks from 2 of 3 replicas (quorum) = durable; the third converges. Reads at ONE favor latency, acceptable for append-only chat history. Peer-to-peer ring: no primary, no SPOF, rebalancing on node add.

Tiered retention: 231 TB → 93.7 TB (−59%)

- Hot: Cassandra RF=3 → 855 GB/day × 30 × 3 = 77.0 TB
- Cold: object storage, 4:1 compressed, 1.3× erasure coding → 16.7 TB (vs 3× full replication)
- Purge at 90 days under a per-tenant retention policy; compliance export available before purge
- Cold tier doubles as the backup — no separate backup job for messages
- Access is recency-biased: conversations older than 30 days are rarely reopened, so slower cold reads land on almost nobody

Citations Used

- Valintry360, "RAG on Salesforce: Trustworthy Enterprise AI Patterns Guide" https://valintry360.com/blogs/rag-on-salesforce-done-right-trustworthy-enterprise-ai-patterns

- Salesforce, "How Einstein Copilot Search Uses Retrieval Augmented Generation to Make AI More Trusted and Relevant" https://www.salesforce.com/uk/news/stories/retrieval-augmented-generation-explained/

Thank you

Multi-Tenant Enterprise RAG Platform · System Design Capstone