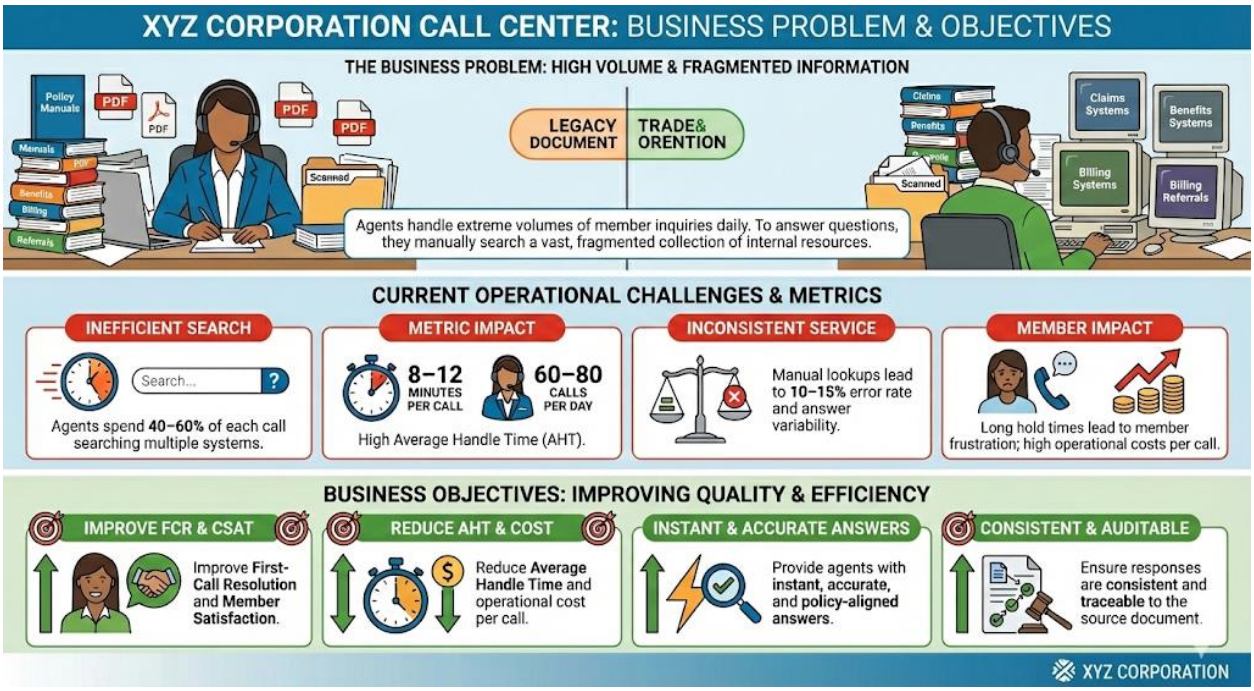
RAG (Retrieval Augmented Generation) – Customer Call Center Assistant Use Case

1. Business Problem & Objectives

The Business Problem

Kaiser Permanente (KP) call center agents handle an extremely high volume of member inquiries daily (e.g., claims status, benefits eligibility, billing, referrals). To answer these questions, agents must manually search across a vast and fragmented collection of internal resources, including policy manuals, PDF guidelines, scanned documents, and legacy systems.



Current Operational Challenges

- **Inefficient Search:** Agents spend 40–60% of each call searching across multiple systems to find the correct policy.
    - *Metric Impact:* Average Handle Time (AHT) is high (8–12 minutes per call), limiting an agent to only 60–80 calls per day.
- **Inconsistent Service:** Manual lookups and misinterpretations lead to a 10–15% error rate and answer variability depending on agent experience.

- **Member Impact:** Long hold times lead to member frustration, and high operational costs are incurred per call.

Business Objectives

The core goals are to improve service quality and operational efficiency by:

- Improving First-Call Resolution (FCR) and Member Satisfaction (CSAT).
- Reducing Average Handle Time (AHT) and operational cost per call.
- Providing agents with **instant, accurate, and policy-aligned** answers.
- Ensuring responses are **consistent and auditable** (traceable to the source document).

2. Existing Knowledge Ecosystem Challenges

KP's internal document library is highly problematic for automated search, consisting mainly of **unstructured, fragmented data**:

- Large PDF policy handbooks, scanned claim documents, and email attachments.
- Provider network materials, benefit summaries, and legacy SharePoint pages.

Limitations of Current ElasticSearch Solution

The existing system, which relies on indexing and **keyword-only search**, is insufficient for complex policy questions:

- It lacks **semantic understanding**, failing to match a member's question (e.g., "Is this service covered?") to the policy's true meaning.
- It frequently returns too many irrelevant hits, forcing agents to manually scan long documents for the exact answer.
- Inconsistent formatting across PDFs, scanned files, and HTML further degrades search quality.

3. Solution Evaluation and RAG Selection

Kaiser Permanente evaluated four main approaches before selecting the final architecture.

Solution 1: LLMs with Direct Prompting (No Retrieval)

- **Description:** Embed the entire policy or relevant excerpts directly into the LLM prompt.

- **Key Risks: Cannot scale** to large document libraries due to prompt size limits. High chance of **hallucinations** (answering without factual grounding). Compliance risk due to sending policy text externally.
- **Status: Rejected** for large, complex, and compliance-sensitive documents.

Solution 2: Fine-Tuning a Model on Policy Documents

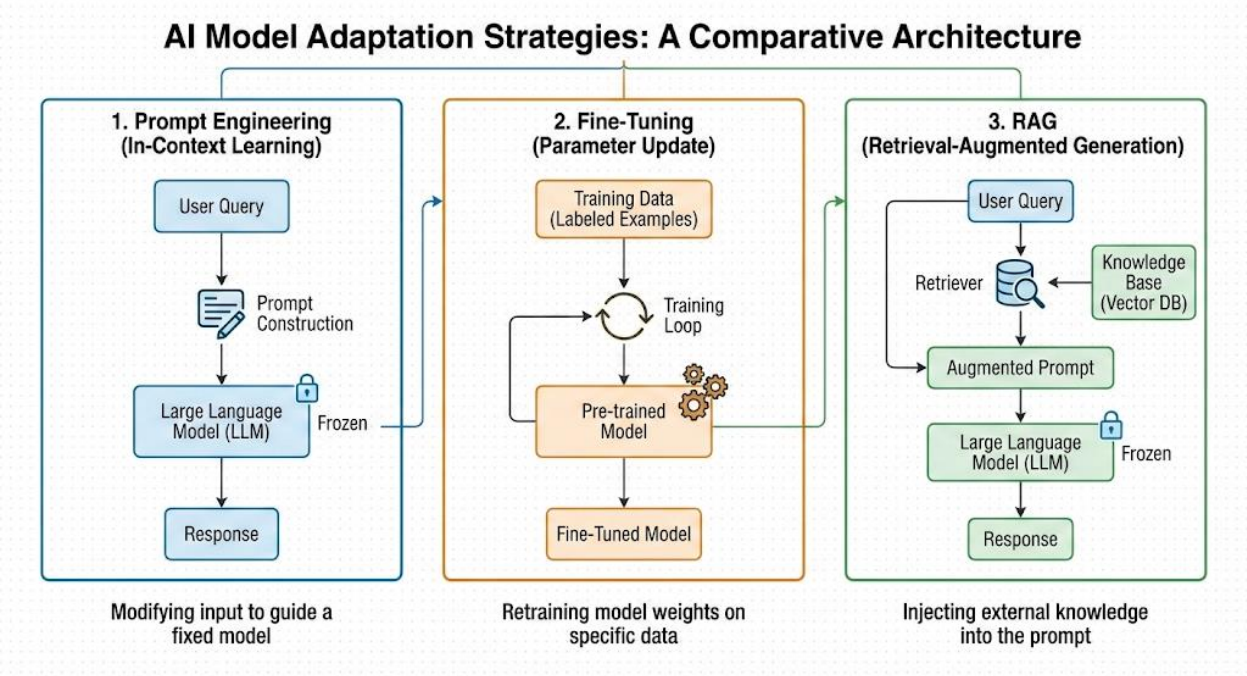
- **Description:** Train or fine-tune an LLM on Kaiser's proprietary benefits and coverage data.
- **Key Risks: Expensive, continuous retraining** required for frequently updated healthcare policies. High risk of storing PHI/PII or proprietary content within the model. Version control and auditability are difficult.
- **Status: Rejected** due to the dynamic nature of healthcare policies and compliance risks.

Solution 3: Third-Party AI Solutions (COTS Products)

- **Description:** Evaluate enterprise AI platforms offering built-in Q&A capabilities.
- **Key Risks:** Requires data sharing with external vendors, raising **HIPAA and data residency concerns**. Limited flexibility and customization. Difficult to integrate with legacy claims systems. High licensing cost.
- **Status: Rejected** as KP requires full control, internal hosting, and the ability to customize retrieval logic for their complex document structure.

Solution 4: Retrieval-Augmented Generation (RAG) AI Assistant

- **Description:** Create an AI assistant using a vector database to retrieve the most relevant policy sections (semantic search), and then use an LLM to generate an answer **grounded** in that source text.
- **Key Benefits:**
    - Ensures answers come from **source-of-truth documents**, drastically reducing hallucinations.
    - Provides **traceable and auditable** responses by citing the source text.
    - **Easy to update** when policies change (by updating the vector index, not the model).
    - Fully deployable on in secure environment, ensuring compliance and control.
    - Works well with diverse, complex, and long unstructured documents.



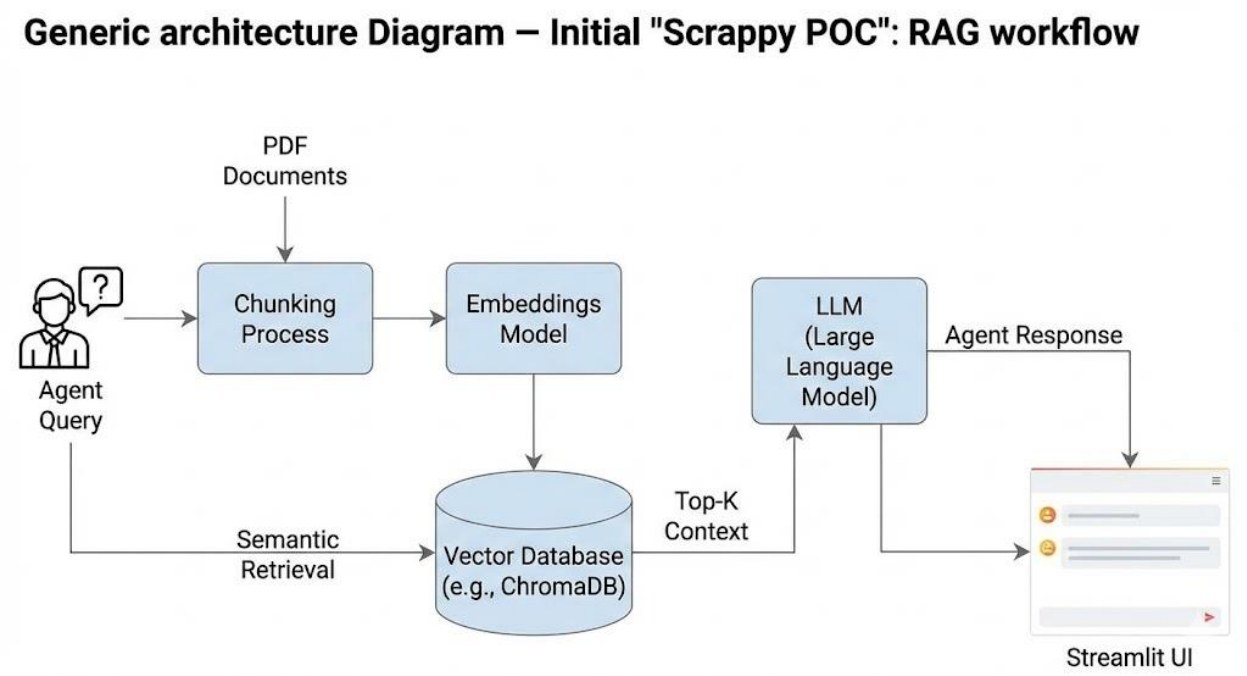
POC:

Team: My director, product manager, Myself (AI Engineer)

Stakeholders: Customer Care Leadership, government agencies, patients, healthcare providers

Validation: Two Customer care human agents

Knowledge corpus : KP pdfs



How the POC Idea Started

The idea began during a discussion between me, my director, and our product partner. We kept hearing the same recurring issue from Customer Care leadership:

- Agents were spending too much time searching through manuals and PDFs
- Average call time was increasing
- Knowledge systems only supported keyword-based text search
- Important policy information was buried inside large documents

To address these challenges, we decided to run a **small, fast, and scrappy POC** over **2–3 weeks** to validate whether RAG could help reduce search time for agents

What We Built in the POC

We selected a limited subset of internal KP PDFs and built a simple RAG system:

1. Converted documents into embeddings
2. Stored them in ChromaDB (vector database)
3. Implemented semantic retrieval
4. Connected the results to an LLM for grounded answer generation
5. Built a lightweight Streamlit UI with a chat interface for quick testing

The goal was not to build a perfect system, but to understand feasibility, value, and agent response.

We Presented the POC during our company's AIML Day and then worked closely with two customer care agents. We asked them to use the tool for a couple of weeks, and they found it helpful.

There were still many issues limited relevancy and accuracy, slow latency, and even some hallucinated responses. But despite the problems, the agents saw clear value and felt it saved them time.

With their positive feedback, we got approval from leadership, including the Vice President, to move forward with our own solution instead of using a third-party option. Once approved, we kicked off phase one

- Here is the git repo Link

Technologies Used

Backend

- Open AI for LLM Open AI
- **Sentence Transformer MiniLM** (for generating embeddings)
- **Chroma DB-** (as an easy-to-use local vector database)

Frontend

- **Streamlit** - Simple chat Interface

POC Learnings

What Worked Well

- Customer Care agents quickly saw the potential of semantic search over keyword based systems
- Natural language questions reduced time spent manually searching large policy documents
- Agents reported faster access to relevant policy sections despite imperfect answers

What Failed or Underperformed

- High and inconsistent latency made the system unsuitable for live call handling
- Lack of structured evaluations made it difficult to measure accuracy or hallucinations
- Feedback was largely subjective and not scalable

What Surprised the Team

- Answer quality depended more on retrieval accuracy than model selection
- Minor retrieval errors often led to confident but incorrect responses
- Even simple evaluations revealed hallucination patterns and improvement areas

What Changed in the Final Design

- Stability and low latency became primary system requirements
- Retrieval quality was improved through better chunking, metadata, and ranking
- A high scale vector database was selected for performance and reliability
- Automated evaluation pipelines were added to track quality and regressions
- Safer response behavior was introduced when grounding was insufficient

Phase one was approved by the customer care team to go ahead.

Phase one

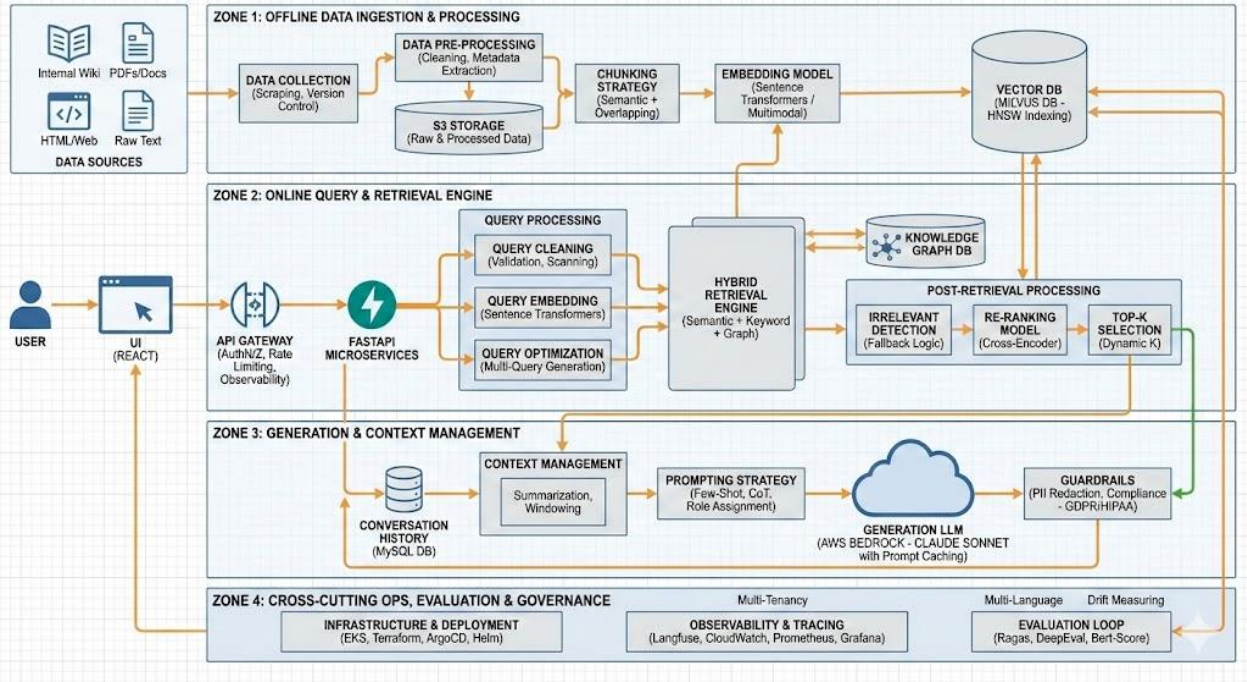
- **Team**: My director, product manager, Myself (Senior AI Engineer), 2 ML Engineer, 2 DS took help from them
- **Stakeholders**: Customer Care Leadership, AIML leadership, CEO, CTO

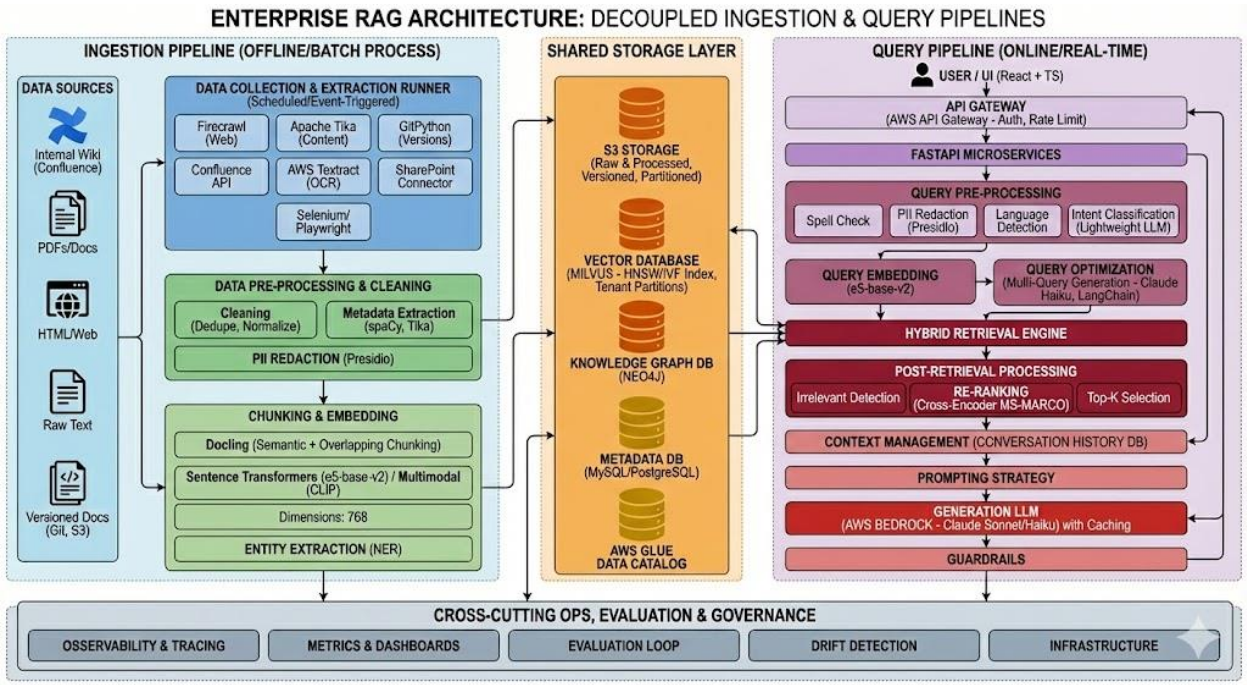
Critical System Requirements

- **Reduce Handle Time (KPI):** Decrease Average Handle Time (AHT) from ~10 minutes to under 5 minutes by reducing search duration to <30 seconds.
- **Service Standardization:** Eliminate variability between agents; ensure a Day-1 junior agent provides the same policy accuracy as a 10-year veteran.
- **Multi-Format Ingestion:** Capable of processing disparate legacy formats—specifically preserving complex **Benefit Tables** in PDFs, scanned claims forms, and dynamic internal HTML wikis.
- **Hybrid Search Precision:** Must support **Dual-Mode Retrieval**: understanding conceptual questions ("Does this cover physiotherapy?") AND precise code lookups ("Status of CPT 99213").
- **Strict Grounding (Zero Hallucinations):** The model is strictly forbidden from using external training knowledge; 100% of answers must be generated *solely* from retrieved Kaiser Permanente context.
- **Verifiable Citations:** Every claim in a response must include a **clickable reference** to the exact source document and page number for immediate agent verification.
- **PII/PHI Redaction (Input Guardrail):** System must automatically detect and mask Member IDs, SSNs, and Dates of Birth *before* the prompt reaches the LLM to prevent data leakage.
- **HIPAA Data Residency:** All vector indices, logs, and document chunks must remain within the internal VPC (Virtual Private Cloud); no data usage for external model training.
- **Multi-Turn Conversation:** The system must maintain context for at least 3 previous turns to allow agents to ask follow-up questions without repeating details.
- **"Unknown" Fallback Logic:** If the retrieval confidence score is below a set threshold (e.g., 0.7), the system must reply "Information not found in policy documents" rather than guessing.
- **Low Latency Performance:** To match the pace of live calls, the system must achieve a **Time-to-First Token (TTFT) of <1.5 seconds** and a full response in <5 seconds.

- **Rapid Data Freshness:** New or updated policy documents must be indexed and searchable within **24 hours** of publication to ensure compliance.

Architecture:





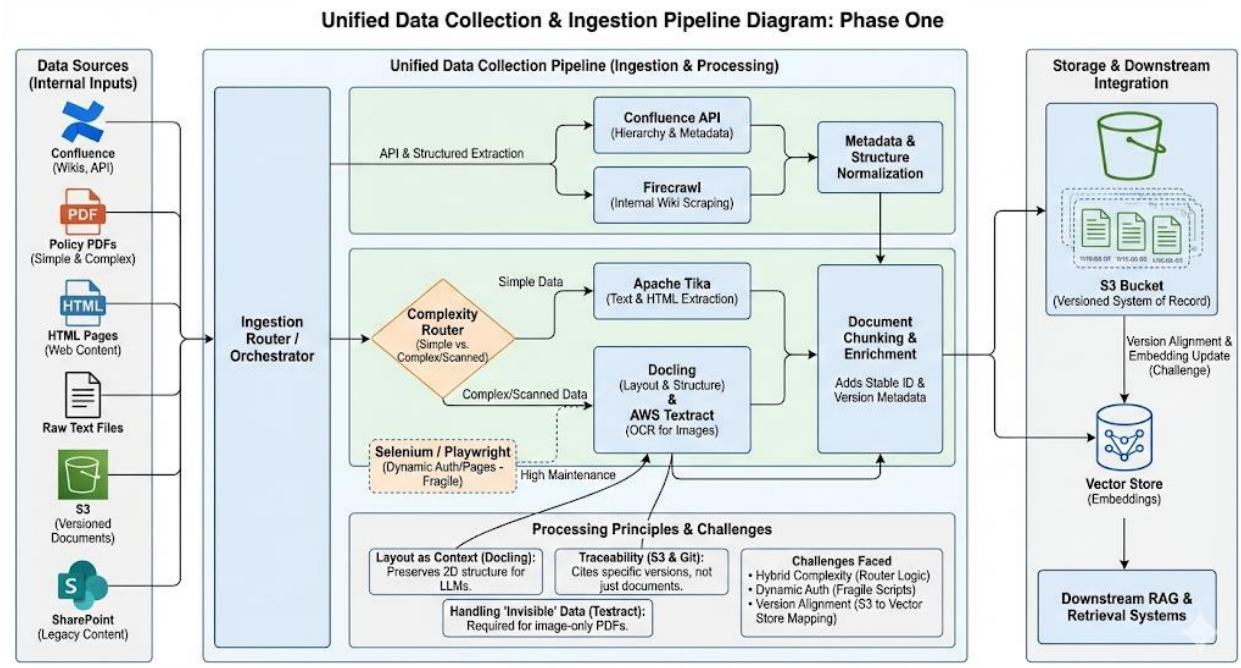
Design and Implementation:

Data Collection

In Phase One, we built a unified data collection pipeline to ingest knowledge from multiple internal sources, including Confluence based internal wikis, policy PDFs, HTML pages, raw text files, and versioned documents stored in S3.

Each document was ingested with a stable identifier and associated version metadata. This allowed us to track document changes over time and ensured that downstream retrieval and generation always referenced the correct policy version.

The ingestion pipeline was designed to support both scheduled batch runs and targeted re ingestion when policy updates were released.



Why We Did That

- **Layout as Context:** In technical manuals, a number's meaning is defined by its row and column headers. Standard extraction flattens tables into nonsense. **Docling** was necessary to preserve this "2D" context into a linear format the LLM can understand.

- **Traceability:** In enterprise environments, "the policy changed yesterday" is a common failure mode. By tracking **S3 versions** and **Git commits**, our RAG system can cite not just the document, but the *version* of the document used.
- **Handling "Invisible" Data: AWS Textract** was strictly required for scanned PDFs (images) where no text layer existed, filling the gap that neither Tika nor standard Python libraries could handle.

What We Tried (Evolution of Approach)

- **Tika vs. Docling:** We initially used **Apache Tika** for everything.
    - *Result:* It was fast and reliable for simple text but failed significantly on financial reports and multi-column research papers. It would merge text across columns, destroying the reading order.
    - *Pivot:* We introduced **Docling** for these complex PDFs. We kept Tika for simpler files because Docling is computationally heavier (slower).
- **Raw HTML Scraping:** We tried scraping Confluence pages directly.
    - *Result:* We got the text but lost the hierarchy (Parent Page > Child Page).
    - *Pivot:* Switched to **Confluence API** to ingest the *tree structure*, allowing us to pass "Context: This is a sub-page of 'Engineering Guidelines'" to the LLM.

Challenges Faced

- **The "Hybrid" Complexity:** Managing two parsers (Tika and Docling) added logic overhead. We had to write a "router" script to detect if a PDF was "simple" (send to Tika) or "complex/scanned" (send to Docling/Textract) to balance cost and speed.
- **Dynamic Auth:** Scripting **Selenium** to handle SSO (Single Sign-On) and MFA redirects for SharePoint was fragile and required constant maintenance compared to API tokens.
- **Version alignment:** Mapping S3 file versions to vector store embeddings was difficult; when a file in S3 is updated, we had to ensure the *old* embeddings were deleted or version-tagged to prevent "ghost" answers.

Tools and Technologies Used

Here is the **Tools Selection** section of your study guide, finalized with the narrative style you requested.

During Phase One, we evaluated multiple tools for data ingestion and content extraction. The goal was to balance reliability, accuracy, maintainability, and alignment with existing infrastructure.

Firecrawl

We explored several custom scraping approaches for internal web content, but they required frequent maintenance as page structures changed. Firecrawl was already part of the internal stack and proved

more reliable for scraping internal wiki-style pages. We adopted Firecrawl for web-based documentation due to its stability and lower operational overhead.

Confluence API

We initially tested scraping Confluence pages using raw HTML extraction. This approach lost important structural information and metadata. The Confluence API provided consistent access to page hierarchy, update timestamps, and ownership data, so it became the primary method for wiki ingestion.

Apache Tika

We evaluated lightweight PDF and text extractors, but they struggled with complex policy documents containing tables and mixed formatting. Apache Tika provided more consistent extraction across PDFs, HTML, and raw text files, making it the default extraction tool.

Docling

While Tika handled raw text extraction well, it did not preserve document structure. We explored Docling to better capture logical sections, headings, and document layout. Docling was adopted alongside Tika for long and complex policy documents where structure mattered for downstream chunking.

AWS Textract

For scanned PDFs and forms, traditional extractors performed poorly. We evaluated multiple OCR options and selected AWS Textract due to its stronger performance on structured documents and tables. Textract was used selectively due to cost and latency considerations.

S3 for Versioned Storage

We explored Git-based storage for document versioning, but S3 provided better integration with existing data pipelines and simpler operational management. S3 was selected as the system of record for versioned documents, enabling controlled updates and rollbacks.

SharePoint Connector

Some legacy content was stored exclusively in SharePoint. We evaluated manual exports but found them error-prone and difficult to keep in sync. A SharePoint connector was added to ensure completeness of ingestion.

Selenium and Playwright

We tested Selenium and Playwright for scraping dynamic pages. While powerful, they introduced higher maintenance and performance costs. These tools were used sparingly and only for sources that could not be ingested through APIs or static scraping.

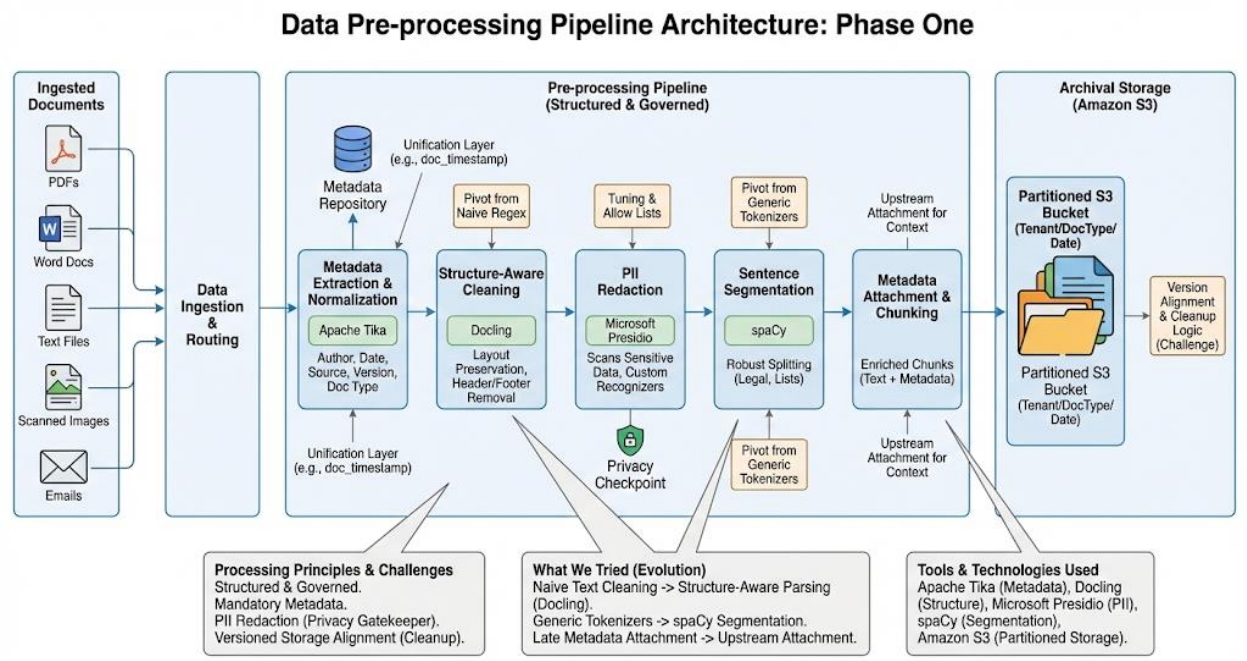
Data Pre-processing

In the Pre-processing Phase,

We implemented a structured data preprocessing pipeline to prepare ingested documents for reliable chunking, indexing, and retrieval. The pipeline focused on cleaning raw inputs, extracting consistent metadata, preserving document structure, and enforcing data governance requirements before any embedding was generated.

We implemented a mandatory "Metadata Extraction" step for every document, pulling critical attributes such as Author, Date, Source, Version, and Document Type. These attributes were not just stored but actively attached to the text chunks to enable "Hybrid Search" (e.g., "Show me policy changes from 2024").

To ensure compliance and security, we integrated a PII (Personally Identifiable Information) redaction layer that scanned and masked sensitive data prior to storage. All processed data was then archived in S3, heavily partitioned by tenant and document type to optimize retrieval performance.



What We Tried (Evolution of Approach)

Naive Text Cleaning. We initially relied on simple regex based cleaning to remove headers and footers.

Result. This approach broke frequently across document types and failed to generalize beyond a small sample set.

Pivot. We moved to structure aware parsing using Docling so that irrelevant sections could be excluded based on document layout rather than string patterns.

Sentence Splitting with Generic Tokenizers. We tested basic sentence tokenizers bundled with common NLP libraries.

Result. They performed poorly on legal and policy text, frequently splitting enumerated clauses incorrectly.

Pivot. We adopted spaCy for sentence segmentation due to its stronger handling of abbreviations, lists, and domain specific punctuation.

Late Metadata Attachment. In early iterations, metadata was added after chunking.

Result. Chunks lost document level context, making filtering and debugging difficult.

Pivot. Metadata extraction was moved upstream so that every chunk inherited document level attributes by default.

Challenges Faced

- PII Detection Accuracy. Presidio occasionally flagged false positives in technical text, such as identifiers or configuration values. We had to tune recognizers and add allow lists to avoid over redaction that reduced answer quality.
- **Metadata Normalization:** Different file types store dates differently (e.g., Creation-Date vs. Last-Modified). We had to write a unification layer to map these disparate fields into a single, queryable schema (e.g., doc_timestamp).
- Versioned Storage Alignment. When documents were updated in S3, ensuring that old chunks and embeddings were removed or correctly version tagged required explicit cleanup logic. Without this, stale content could still surface during retrieval.

Tools and Technologies Used

Microsoft Presidio We evaluated manual regex libraries but found them brittle. Presidio was selected for its context-aware PII detection and its ability to be customized with specific "recognizers" for internal data formats. It serves as the primary gatekeeper for data privacy.

Docling Standard OCR tools flattened documents into nonsense strings. Docling was chosen for its ability to recognize and preserve the hierarchical structure of documents (headers, lists, tables), which is essential for context-aware chunking.

spaCy We needed a robust way to split text without breaking sentences. spaCy provided industrial-strength sentence segmentation and tokenization, ensuring that our chunks were linguistically coherent rather than arbitrarily cut.

Apache Tika While Docling handled structure, Tika remained our go-to for deep metadata extraction across hundreds of file formats. It reliably pulled "invisible" properties like Author, Keywords, and Modification Dates that were critical for our filtering logic.

Amazon S3 (Partitioned) We moved away from flat storage to a partitioned S3 bucket structure (Tenant/DocType/Date). This reduced listing costs and allowed us to lifecycle older, irrelevant data to cheaper storage classes automatically.

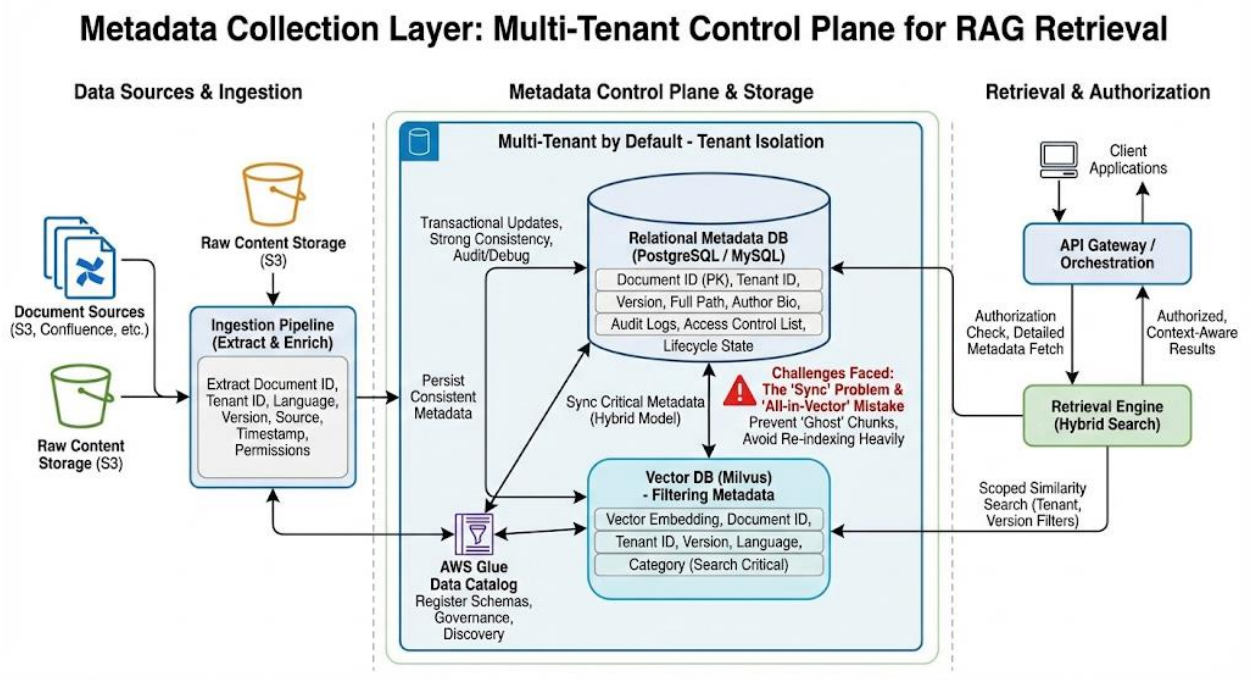
Metadata Collection

In Phase One, we built a dedicated metadata collection layer to track document identity, ownership, access control, and lifecycle state independently from raw content storage. This layer acted as the control plane for retrieval, filtering, and authorization decisions across the RAG system.

For every ingested document, we extracted and persisted a consistent set of fields, including document ID, tenant ID, language, version, source system, last modified timestamp, and access permissions. These fields were treated as first class data rather than auxiliary attributes, and they were required for every downstream operation.

The system was designed as multi tenant by default. Tenant isolation was enforced at both the storage and collection level, ensuring that documents, embeddings, and metadata could never be queried across tenant boundaries. This applied equally to raw files in object storage, vector collections, and relational metadata tables.

Relational metadata was stored in a structured database using PostgreSQL or MySQL. This enabled strong consistency, transactional updates, and expressive queries for auditing, debugging, and administrative workflows. Vector level metadata required for retrieval time filtering was synchronized into the vector database to support fast, scoped queries.



Why We Did That

Retrieval Without Metadata Fails Quietly. Early prototypes showed that high quality embeddings alone are insufficient. Without strong metadata filters, retrieval returns technically relevant but contextually invalid results, such as content from the wrong tenant or an outdated policy version.

Challenges Faced

The "All-in-Vector" Mistake: We initially stuffed *all* metadata (audit logs, full file paths, author bios) directly into the Vector DB metadata fields.

- **Result:** The vector index size ballooned, and updating simple attributes (like a filename change) required re-indexing the vectors, which was computationally expensive.
- **Pivot:** We adopted a hybrid model. We kept the "Search Critical" metadata (Tenant ID, Year, Category) in **Milvus** for filtering, but moved the heavy, descriptive metadata to **PostgreSQL/MySQL**.

The "Sync" Problem: Keeping the Metadata DB (PostgreSQL) and the Vector Store (Milvus) in perfect sync was painful. If a file was deleted in S3, we had to ensure it was transactionally removed from *both* the SQL table and the vector index to prevent "ghost" chunks from appearing in search results.

Tools and Technologies Used

Relational Metadata Database: PostgreSQL or MySQL was used to store authoritative document metadata. This database handled tenant mapping, version tracking, permissions, and lifecycle state with transactional guarantees.

AWS Glue Data Catalog: AWS Glue Data Catalog was used to register processed datasets and metadata schemas. It provided centralized visibility into available document collections and supported governance and discovery workflows across teams.

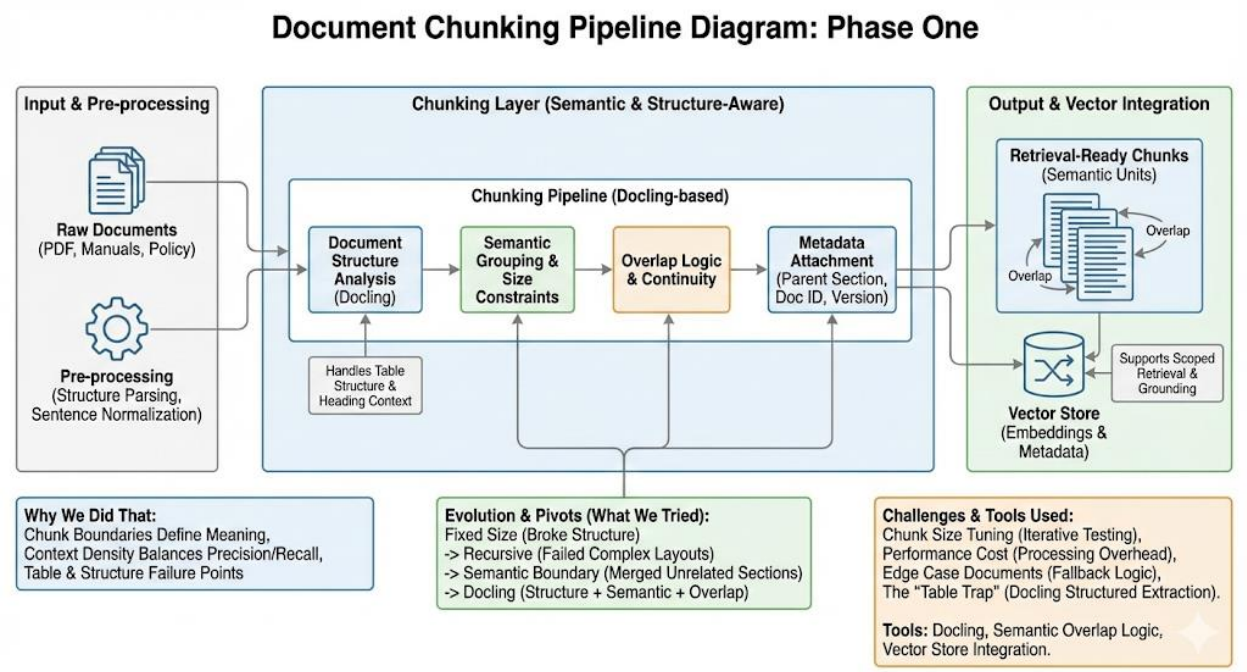
Milvus Metadata: Milvus was used to store retrieval time metadata alongside vector embeddings. Only fields required for filtering, such as tenant ID, document ID, version, and language, were replicated into Milvus to support fast and scoped similarity search without overloading the vector store with relational concerns.

Chunking

In Phase One, we implemented a dedicated chunking layer to convert pre processed documents into retrieval ready units while preserving semantic meaning and structural context. Chunking was treated as a core design decision rather than a mechanical step, since it directly impacts recall quality, grounding, and generation accuracy.

The pipeline evaluated multiple chunking strategies across policy documents, technical manuals, and research style content. Each document was segmented after structure parsing and sentence normalization, with chunk boundaries determined by both semantic coherence and size constraints suitable for embedding models.

The final chunk objects retained references to their parent sections, document identifiers, and version metadata. Overlap was introduced deliberately to avoid hard semantic breaks between adjacent chunks, especially in cases where definitions or constraints span multiple paragraphs.



Why We Did That

Chunk Boundaries Define Meaning. In policy and compliance text, a sentence often derives its meaning from the surrounding section or table header. Naive chunking produces fragments that are syntactically valid but semantically incomplete, leading to misleading retrieval results.

Retrieval Precision Depends on Context Density. Smaller chunks improve recall but degrade answer quality when context is lost. Larger chunks preserve meaning but reduce retrieval precision. The strategy had to balance both forces without relying on post hoc prompt fixes.

Tables and Structured Sections Are Failure Points. Financial and operational documents rely heavily on tables and nested sections. A chunking approach that ignores structure breaks row header relationships and conditional clauses, increasing hallucination risk during generation.

What We Tried (Evolution of Approach)

Fixed Size Chunking. We initially split documents into uniform token based chunks with overlap.

Result. This approach was simple and fast but frequently cut across section boundaries. Definitions, constraints, and table rows were separated from their governing headers, producing low quality retrieval hits.

Pivot. We moved away from size only chunking and introduced structure awareness.

Recursive Chunking. We tested recursive splitting using paragraphs first and sentences as a fallback when size limits were exceeded.

Result. This improved boundary alignment but still failed on complex layouts where paragraphs did not represent true semantic units, especially in documents converted from PDFs.

Pivot. We introduced semantic signals to guide chunk boundaries rather than relying solely on formatting.

Semantic Boundary Chunking. We evaluated sentence and paragraph grouping based on semantic similarity.

Result. This produced more coherent chunks but struggled when applied without document structure. Similarity based grouping occasionally merged content from different sections that shared vocabulary but differed in intent.

Pivot. We combined semantic grouping with explicit document structure to constrain grouping within logical sections.

Docling

We adopted Docling based chunking that combines document structure awareness with semantic grouping and controlled overlap. Chunk boundaries respect section hierarchy, table structure, and heading context, while overlap ensures continuity across adjacent chunks.

This approach consistently preserved intent across clauses, maintained table semantics, and reduced retrieval of orphaned sentences. In downstream evaluation, it led to fewer hallucinated assumptions during generation because retrieved chunks carried sufficient governing context.

Challenges Faced

Chunk Size Tuning. Determining the optimal size and overlap required iterative testing across document types. Policies, manuals, and research papers each exhibited different sensitivity to chunk granularity.

Performance Cost. Structure aware and semantic chunking introduced additional processing overhead compared to fixed size approaches. We mitigated this by applying the strategy selectively to documents where structure mattered most.

Edge Case Documents. Some legacy documents lacked consistent headings or formatting. For these cases, fallback logic was required to prevent over fragmentation or excessively large chunks.

The "Table Trap" in PDFs Our standard extractors failed miserably on financial reports containing embedded tables.

Result: A question like "What was the copay?" returned nonsense because the parser read the table row-by-row as a single flat string, destroying the column relationships.

Pivot:

- Use Structured Table Extraction in Docling
- Don't treat tables as plain text - instead, extract them as structured objects where:
- Row/column relationships are maintained
- Cells have coordinates (row, column indices)
- Headers are distinguished from data cells

Tools and Technologies Used

Docling: Docling was used to drive the final chunking strategy by providing access to document structure, including headings, sections, and tables. It enabled chunk boundaries that aligned with logical units rather than arbitrary token counts.

Semantic Overlap Logic: Custom overlap logic was applied to adjacent chunks to preserve continuity across boundaries. This reduced retrieval gaps and improved answer grounding without significantly increasing index size.

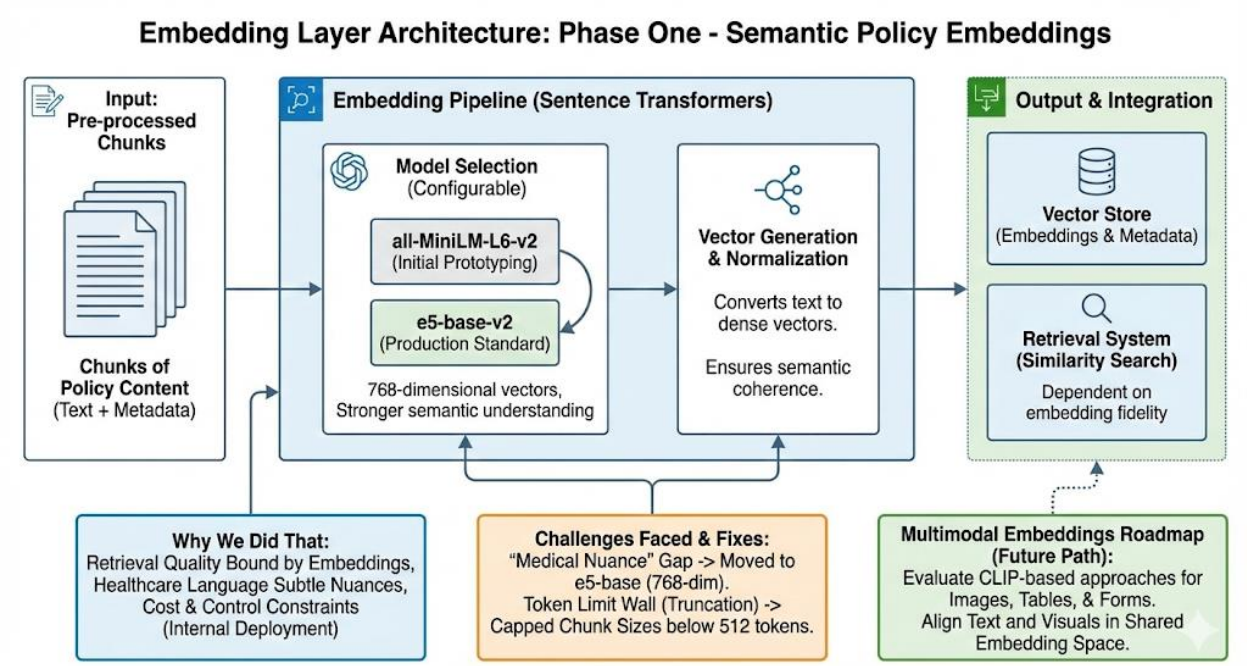
Vector Store Integration: Chunk metadata was tightly integrated with the vector store to ensure retrieval respected document structure, versioning, and tenant isolation during similarity search.

Embeddings

In Phase One, we designed the embedding layer to convert chunks of policy content into dense vector representations that could reliably capture semantic meaning across legal, clinical, and operational language. The embedding strategy was treated as a first order system component, since retrieval quality was directly dependent on embedding fidelity rather than model size alone.

We initially standardized on Sentence Transformers due to their strong performance on semantic similarity tasks, full internal deployability, and predictable cost profile. Embeddings were generated only after preprocessing, metadata attachment, and chunking were complete, ensuring that each vector represented a semantically coherent and context rich unit of information.

The embedding pipeline was built to be modular, allowing models to be swapped without reworking ingestion or retrieval logic. This proved important as we iterated on model choice based on evaluation results.



Why We Did That

Retrieval Quality Is Bound by Embeddings. Early experiments confirmed that even a strong reranker or prompt could not compensate for weak embeddings. If semantically related policy clauses were not close in vector space, retrieval consistently failed regardless of downstream tuning.

Healthcare Language Is Subtle. Policy documents often differ by small but meaningful phrases such as exclusions, limits, or conditional coverage rules. Embeddings needed to capture intent, not just surface level keyword similarity.

Cost and Control Constraints. Embeddings are generated at indexing time and re generated whenever documents change. This made per token pricing and external dependency risk a major concern at scale, pushing us toward internally hosted models.

What We Tried (Evolution of Approach)

all MiniLM L6 v2

We began with all MiniLM L6 v2 from Sentence Transformers.

- Result. The model was fast, lightweight, and inexpensive to run. With 384 dimensional vectors, it enabled quick indexing and low memory usage. It performed well for simple semantic matching and was suitable for early prototyping.

- Limitations. On longer policy clauses and nuanced coverage questions, semantic recall degraded. Closely related policy sections were sometimes not retrieved together, especially when wording differed significantly.
- Pivot. We evaluated higher capacity embedding models to improve semantic capture.

e5 base v2

We migrated to e5 base v2 after benchmarking retrieval accuracy on internal policy queries.

- Result. With 768 dimensional embeddings, the model demonstrated stronger semantic understanding, especially for question style queries and paraphrased policy language. Recall improved consistently on edge cases involving exclusions, footnotes, and conditional rules.
- Tradeoff. Index size and embedding generation time increased. This was accepted due to the measurable gain in retrieval accuracy and reduced hallucination risk downstream.

MPNet Based Models

We evaluated MPNet based Sentence Transformer variants as part of the comparison set.

- Result. MPNet performed competitively on semantic similarity but offered marginal gains compared to e5 base v2 while incurring similar compute costs. Given the smaller improvement delta, we standardized on e5 base v2 for production.

Managed Embedding APIs

We evaluated external embedding providers, including OpenAI and Cohere.

- Result. While quality was strong, these options introduced recurring cost at scale, external dependency risk, and additional compliance review overhead. Given frequent document updates and re indexing requirements, the cost model was not favorable.
- Decision. We selected open source Sentence Transformer models to retain full control over data, cost, and deployment while achieving acceptable retrieval performance.

Multimodal Embeddings Roadmap

Although Phase One focused exclusively on text, the embedding architecture was designed with multimodal expansion in mind. Policy documents frequently contain tables, scanned forms, and images that carry semantic meaning not fully represented in text alone.

We evaluated CLIP based approaches, including OpenAI CLIP variants, as a future path for embedding images and structured visual content. The long term goal is to align text, tables, and images into a shared embedding space, enabling grounded answers that reference both textual clauses and visual artifacts such as benefit tables or claim forms.

This roadmap ensures that future iterations of the system can support richer grounding without rearchitecting the retrieval pipeline.

Challenges Faced

1. The "Medical Nuance" Gap

The Problem: Off-the-shelf models trained on general internet data (Reddit, Wikipedia) struggled with clinical specificity. For example, a query about *"renal failure"* would match well with *"kidney disease,"* which is good. However, the model often failed to distinguish between *"Skilled Nursing Facility"* (covered) and *"Custodial Care"* (not covered) because, linguistically, they look very similar (both involve caring for patients in a facility).

The Fix: This was the primary driver for moving from the 384-dimensional **MiniLM** to the 768-dimensional **e5-base**. The larger vector space provided enough "room" to capture the subtle but contractually critical differences between these terms.

2. The Token Limit Wall (Truncation)

The Problem: Many embedding models have a strict limit (e.g., 512 tokens). If a policy section was 700 tokens long, the model would simply **truncate** (delete) the last 200 tokens before creating the embedding.

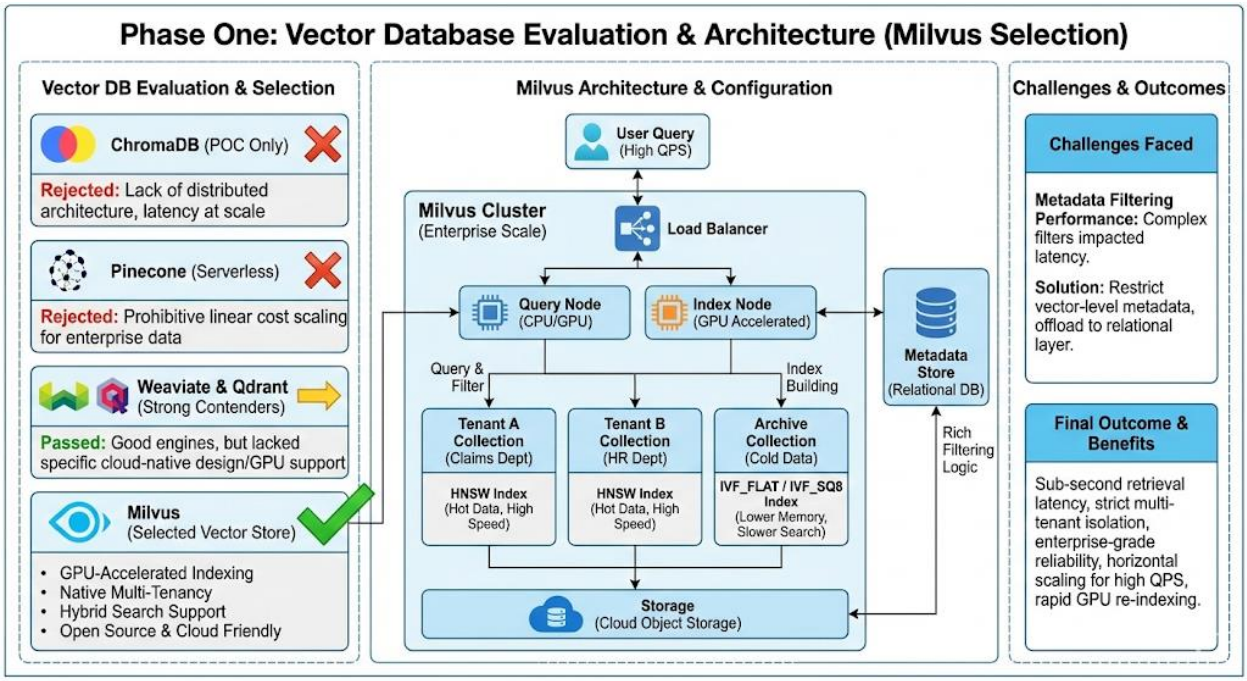
The Risk: In insurance policies, the "Exclusions" (what is *not* covered) are often listed at the very bottom. Truncation meant our search engine was "blind" to the most important part of the document, potentially leading the AI to say a service was covered when it wasn't.

The Fix: We ensured our **Chunking Strategy** (previous section) strictly capped chunk sizes *below* the embedding model's limit (512 tokens), ensuring 100% of the text was encoded.

Vector Database

In Phase One, we evaluated multiple vector databases to support semantic retrieval at enterprise scale. The vector store was treated as a core system dependency since it directly impacts latency, recall, multi tenant isolation, and long term operability. The evaluation focused on scalability under high query volume, filtering capability, operational control, and alignment with internal deployment constraints.

Objective: Select and implement a vector store capable of handling millions of policy chunks with sub-second retrieval latency, strict tenant isolation, and enterprise-grade reliability.



Why We Did That

- **The Scale Problem:** Our document corpus isn't static. It grows daily. We needed a system that wouldn't just work for the 50MB POC but for terabytes of data in production.
- **Hybrid Necessity:** As discovered in the Embedding phase, semantic search isn't enough. We need to combine vector similarity with keyword matching (BM25) for specific medical codes. The database had to support this natively.

Evaluation Phase (The "Why Not" List)

We evaluated four major players against our specific enterprise requirements before settling on the final architecture.

- **ChromaDB**
  - *Verdict:* **Rejected (POC Only).**
  - *Why:* Excellent for local development and our initial Streamlit demo. However, it lacked the distributed architecture required for high availability and struggled with latency as the dataset grew beyond memory limits.
- **Pinecone**
  - *Verdict:* **Rejected.**
  - *Why:* While it’s the gold standard for "easy" serverless vector search, the cost scales linearly with data. For an enterprise handling massive internal logs and archives, the long-term cost modeling was prohibitive compared to self-managed options.
- **Weaviate & Qdrant**
  - *Verdict:* **Strong Contenders, but passed.**

- *Why:* Both are excellent engines. However, Milvus edged them out on specific architectural needs: its separation of storage and compute (cloud-native design) and superior GPU acceleration support for indexing.

Why Milvus?

- **GPU-Accelerated Indexing:** Milvus can leverage GPUs to build indices orders of magnitude faster than CPU-only solutions. When re-indexing a massive policy library after an embedding model update, this reduces downtime from days to hours.
- **Native Multi-Tenancy:** It supports robust isolation. We didn't have to hack "tenant_id" into metadata filters; we could architecturally separate data.
- **Hybrid Search Support:** Strong native support for weighted reranking, allowing us to balance Dense Vector results with Sparse (Keyword) results.
- Open Source and Cloud Friendly. Milvus aligned with internal requirements for open source software and cloud portability. It could be deployed in a managed cloud configuration or self hosted within the internal VPC without architectural changes.

Deep Dive: How We Configured It

1. Multi-Tenancy Strategy: Separate Collections

Instead of dumping everything into one giant index and filtering by tenant_id (which ruins search speed), we implemented Separate Collections per Tenant.

- *Benefit:* Complete physical isolation of data. A query from the "Claims Dept" scans only their index, ensuring zero chance of leaking "HR" data.
- *Performance:* Smaller, dedicated indices yield faster search (lower latency) than filtering a massive global index.

2. Indexing Strategy: HNSW vs. IVF

We utilized Milvus's flexibility to choose different index types for different data needs:

- **HNSW (Hierarchical Navigable Small World):** Used for our active "Hot" data (current policies). It consumes more memory but offers the fastest possible retrieval speed.
- **IVF_FLAT / IVF_SQ8:** Used for "Cold" archival data. It uses significantly less memory (compressed), accepting slightly slower search speeds for older records that are rarely accessed.

3. Handling High QPS (Queries Per Second)

During load testing, the system faced spikes of hundreds of concurrent agent queries. Milvus scaled horizontally, maintaining stable latency even under high QPS, proving the decision to move away from in-process stores like Chroma was correct.

Challenges

Metadata Filtering Performance. Complex filters combining tenant, version, language, and document type occasionally impacted query latency. We had to restrict vector level metadata to retrieval critical fields and offload richer filtering logic to the relational metadata layer.

Indexing Algorithm

In Phase One, we designed the vector indexing layer to balance retrieval accuracy, latency, and memory efficiency under production scale workloads. Index selection was treated as a tunable system component rather than a static choice, since different document volumes and query patterns required different performance tradeoffs.

The indexing strategy was implemented within Milvus and aligned closely with embedding dimensionality, expected query concurrency, and tenant level isolation. Multiple index types were evaluated and combined to support both high recall semantic search and predictable response time during peak call center traffic.

Why We Did That

Approximate Search Is Mandatory at Scale. Exact nearest neighbor search becomes infeasible as vector counts grow into the millions. Approximate algorithms were required to keep query latency within live call handling constraints.

Recall and Latency Must Be Tunable. No single index configuration performs optimally across all workloads. The system needed the ability to trade recall for speed in a controlled and observable way.

Memory Efficiency Matters. Higher dimensional embeddings significantly increase memory footprint. Indexing strategies had to reduce memory pressure without materially degrading retrieval quality.

What We Implemented

HNSW

- Hierarchical Navigable Small World was used as a primary indexing strategy for collections requiring high recall and low query latency.

- Behavior. HNSW constructs a layered graph that enables fast navigation through the vector space, producing strong nearest neighbor results even under tight latency budgets.
- Use Case. Applied to smaller or high value collections where retrieval accuracy directly impacted answer correctness.
- Tradeoff. Higher memory consumption and longer index build time compared to other approaches.

IVF

- Inverted File Index was introduced to support large scale collections with millions of vectors.
- Behavior. IVF partitions the vector space into clusters and restricts search to a subset of relevant partitions, reducing query cost.
- Use Case. Applied to large policy corpora where predictable performance under high QPS was required.
- Tradeoff. Recall sensitivity depends on clustering quality and the number of partitions searched.

Hybrid Index Strategy

- Milvus allowed us to combine HNSW and IVF characteristics to balance recall and latency.
- Behavior. IVF was used to narrow the candidate search space, while HNSW improved local navigation within selected partitions.
- Result. This hybrid approach consistently delivered better recall than IVF alone while maintaining lower latency and memory usage than pure HNSW at scale.

Product Quantization

- Product Quantization was evaluated to address memory constraints as index size grew.
- Behavior. PQ compresses vectors into smaller representations, reducing memory footprint while enabling approximate distance computation.
- Use Case. Applied selectively to very large collections where memory efficiency outweighed minor recall degradation.
- Tradeoff. Slight loss in precision, requiring careful validation to ensure retrieval quality remained acceptable for policy driven answers.

Challenges Faced

Parameter Sensitivity. Index performance was highly sensitive to configuration parameters such as graph connectivity, cluster count, and probe depth. Extensive tuning was required per collection to avoid silent recall degradation.

Re index Overhead. Index rebuilds after large document updates were computationally expensive and required careful orchestration to prevent service disruption.

Mixed Workloads. Different tenants and document types exhibited different retrieval patterns, making a single indexing profile insufficient across the system.

Query Cleaning and Validation

In Phase One, we implemented a query cleaning and validation layer to normalize agent input before it entered the retrieval pipeline. Call center queries are often noisy, incomplete, or include sensitive identifiers copied directly from internal systems. Without preprocessing, these issues degraded retrieval accuracy and introduced compliance risk.

The query pipeline was designed to be lightweight, deterministic where possible, and fast enough to operate inline with live call handling. Each step was evaluated for latency impact to ensure it did not violate end to end response time requirements.

Why We Did That

Garbage In Produces Confidently Wrong Answers. Even high quality embeddings and indices fail when queries contain misspellings, extraneous identifiers, or irrelevant tokens. Cleaning the query upfront consistently improved retrieval relevance.

Compliance Starts at the Prompt. Agent queries sometimes included Member IDs, claim numbers, or dates of birth. These identifiers must never reach the LLM, making early detection and masking mandatory.

Intent Drives Retrieval Strategy. A vague coverage question and a precise code lookup require different retrieval behavior. Identifying intent early allowed the system to route queries more effectively.

Spell Correction

- We introduced spell correction to handle common typos and shorthand entered during live calls.
- Implementation. SymSpell was used for fast dictionary based correction, with Python TextBlob evaluated for comparison.
- Result. Correcting misspelled medical terms and procedure names improved recall without materially increasing latency.

PII Detection and Masking

- All incoming queries were scanned for sensitive data.

- Implementation. Microsoft Presidio was applied at query time to detect and mask PII such as Member IDs, SSNs, and dates of birth.
- Result. This ensured that no sensitive identifiers were passed to downstream retrieval or generation components.

Language Detection

- We added language detection to validate that queries were supported by the current corpus.
- Implementation. The langdetect library was used to identify query language.
- Result. Non supported languages could be handled gracefully or routed to fallback workflows rather than producing irrelevant results.

Query Intent Classification

- We introduced a lightweight intent classification step to guide retrieval behavior.
- Implementation. A small LLM based classifier was used to distinguish between policy coverage questions, procedural lookups, code based queries, and general informational requests.
- Result. Intent classification enabled more targeted retrieval strategies and reduced false positives during semantic search.

Challenges Faced

Latency Budget. Each preprocessing step added incremental latency. Careful optimization and early exits were required to keep the pipeline within live call constraints.

False Positives in PII Detection. Presidio occasionally flagged numeric codes or internal identifiers as sensitive data. Tuning recognizers and allow lists was required to avoid over masking.

Spell Correction Risk. Over aggressive correction occasionally altered legitimate medical terms or internal acronyms. Guardrails were added to restrict corrections to high confidence cases.

Intent Ambiguity. Some queries combined multiple intents in a single sentence. The classifier had to be tuned to prioritize retrieval safety over aggressive routing.

Query Embedding

In Phase One, we implemented a dedicated query embedding step to convert cleaned and validated agent queries into vector representations compatible with the document embedding space. Query embeddings were generated using the same model and configuration as document embeddings to ensure semantic alignment during similarity search.

This step was intentionally separated from document embedding to allow query specific formatting, validation, and future customization without impacting the indexing pipeline.

What We Implemented

- We standardized on e5 base v2 for both document and query embeddings.
- Behavior. The model demonstrated strong performance on semantic similarity, paraphrase detection, and question answer style retrieval across policy text.
- Benefit. Using a single model simplified versioning, evaluation, and re indexing workflows.

Instruction Based Query Formatting

- We applied instruction based formatting to all queries before embedding.
- Implementation. Each user query was prefixed with the string “query: “ prior to embedding generation.
- Result. This aligned the query representation with the training objective of the E5 model, improving semantic matching against document chunks embedded with the corresponding “passage” format.

Query Optimization Training

In Phase One, we introduced a query optimization training loop to improve retrieval quality based on real agent behavior rather than static benchmarks. Early evaluations showed that generic embedding models performed well on standard semantic similarity tasks but struggled with internal jargon, shorthand, and query patterns unique to customer care workflows.

To address this, we treated retrieval as a learnable system component and invested in feedback driven optimization grounded in production usage.

Why We Did That

Generic Models Miss Domain Signals. Healthcare policy queries often include internal abbreviations, partial phrases, and conversational shortcuts that are not well represented in public training data. Without adaptation, relevant content was frequently under retrieved.

Retrieval Errors Cascade Downstream. Even small retrieval failures caused the generation layer to produce confident but incorrect answers. Improving retrieval precision had a disproportionate impact on overall system accuracy and safety.

How We Did It

Retrieval Log Collection

We instrumented the retrieval layer to capture query text, embedded vectors, retrieved chunks, ranking scores, and final agent interactions. These logs were stored with strict privacy controls and excluded any masked PII.

Relevance Labeling

From the logs, we constructed query result pairs labeled as relevant or not relevant. Labeling was performed using a combination of human review from domain experts and automated heuristics based on agent follow up behavior.

Query Rewrite Chains

- We implemented query rewrite chains using LangChain to normalize and expand ambiguous queries before embedding.
- Behavior. The rewrite step reformulated terse or incomplete agent input into clearer semantic queries while preserving original intent.
- Benefit. This reduced variability in query phrasing and improved consistency in retrieval outcomes.

Feedback and Evaluation Loops

- We integrated Deepeval to continuously evaluate retrieval quality and surface regressions.
- Behavior. Deepeval scores were tracked over time to measure improvements and detect degradation after model updates or corpus changes.
- Benefit. This enabled controlled iteration rather than subjective tuning and made retrieval quality measurable and auditable.

Challenges Faced

Labeling Cost and Consistency. Human labeling required domain expertise and was time intensive. Ensuring consistent relevance judgments across reviewers required clear guidelines and periodic calibration.

Multi Query Generation

In Phase One, we implemented multi query generation to improve recall for complex or ambiguous agent questions. Single query semantic search often failed to retrieve all relevant policy sections, especially when coverage rules were distributed across multiple documents or phrased differently than the agent's input.

Multi query generation expanded a single user question into several semantically related queries, increasing the likelihood of retrieving all governing context required for a grounded answer.

Why We Did That

Single Queries Are Fragile. Small wording differences between an agent's question and policy language frequently caused relevant sections to be missed, even with strong embeddings.

Policy Logic Is Distributed. Coverage rules, exclusions, and exceptions are often defined across multiple sections or documents. Relying on a single query biased retrieval toward only one aspect of the policy.

Recall Matters More Than Precision Upfront. Early retrieval stages should favor recall, since downstream ranking and grounding logic can filter noise more safely than hallucinating due to missing context.

How We Implemented It

LLM Based Query Expansion

- We used a lightweight LLM to generate multiple reformulations of the original query.
- Implementation. Claude Haiku was selected due to its low latency and cost profile. For each user query, the model generated three to five semantically related queries that preserved intent while varying phrasing, terminology, and focus.
- Result. Expanded queries captured synonyms, alternate policy language, and implicit constraints that were often absent from the original agent input.

Retrieval Orchestration

- Each generated query was embedded and executed independently against the vector store.
- Implementation. LangChain's MultiQueryRetriever was used to orchestrate parallel retrieval and aggregate results across queries.
- Result. The combined candidate set consistently improved recall without significantly increasing end to end latency.

Prompt Engineering

- Custom prompts were designed to constrain the LLM to produce policy relevant rewrites rather than creative paraphrases.
- Guardrails. The prompt emphasized semantic equivalence, domain terminology, and avoidance of speculative expansions.

Challenges Faced

Latency Control. Generating multiple queries added an additional LLM call. Careful model selection and batching were required to keep total response time within SLA.

Query Drift Risk. Poorly constrained prompts occasionally produced queries that shifted intent or introduced unrelated concepts. Prompt tuning and strict validation were required to prevent retrieval noise.

Candidate Explosion. Multi query retrieval increased the number of retrieved chunks. Downstream ranking and deduplication logic had to be strengthened to maintain precision.

Cost Management. While individual calls were inexpensive, cumulative cost could grow under high traffic. Multi query generation was applied selectively based on query complexity and confidence thresholds.

Retrieval Strategy: Semantic, Keyword, and Graph Based

In Phase One, we implemented a hybrid retrieval strategy that combined semantic similarity, keyword matching, and graph based reasoning. No single retrieval method was sufficient to handle the diversity of call center queries, which ranged from conversational coverage questions to precise code based lookups and entity specific follow ups.

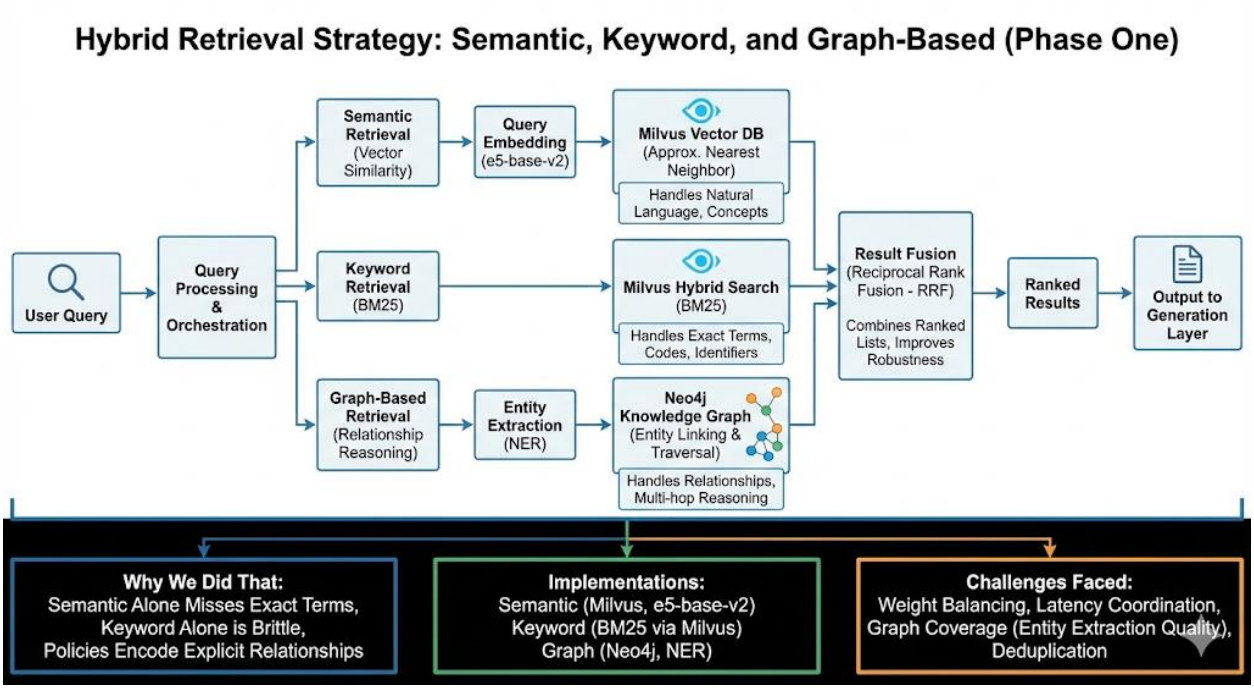
The retrieval layer was designed to maximize recall while maintaining strict relevance constraints. Each retrieval modality contributed a different signal, and final results were combined in a controlled and explainable manner.

Why We Did That

Semantic Search Alone Is Insufficient. Vector similarity excels at conceptual matching but can miss exact terms such as procedure codes, benefit names, or internal identifiers that agents rely on.

Keyword Search Alone Is Brittle. BM25 performs well for exact matches but fails when agent phrasing differs from policy language or when synonyms are used.

Policies Encode Relationships. Many answers depend on relationships between entities such as procedures, plans, exclusions, and providers. These relationships are not always explicit in text proximity alone.



Semantic Retrieval

Semantic retrieval was implemented using vector similarity search over embedded document chunks.

- Implementation. Milvus was used to execute approximate nearest neighbor search against e5 base v2 embeddings.
- Behavior. This path handled natural language questions and paraphrased queries effectively, retrieving conceptually related policy sections even when wording differed.

Keyword Retrieval

Keyword based retrieval was added to capture exact term matches.

- Implementation. BM25 was enabled through Milvus hybrid search capabilities.
- Behavior. This path excelled at precise lookups such as CPT codes, plan names, and regulatory identifiers that must match exactly.

Graph Based Retrieval

We introduced a knowledge graph to model explicit relationships across policy entities.

- Implementation. Named entities were extracted using NER and linked into a Neo4j graph. Entities included procedures, benefits, plans, exclusions, and providers.
- Behavior. Graph traversal allowed retrieval of related policy sections based on entity relationships rather than text similarity alone, supporting multi hop reasoning such as exclusions tied to specific plans.

Result Fusion

Results from all three retrieval paths were merged using Reciprocal Rank Fusion.

- Implementation. Each retrieval method produced a ranked list of candidate chunks. RRF combined these lists by assigning higher weight to results that appeared across multiple modalities.
- Benefit. This approach improved robustness and reduced dependence on any single retrieval signal while remaining simple and explainable.

Challenges Faced

Weight Balancing. Tuning the relative influence of semantic, keyword, and graph based results required iterative testing. Over weighting any single modality degraded performance on certain query types.

Latency Coordination. Running multiple retrieval paths in parallel introduced orchestration complexity. Careful timeout management and early stopping were required to maintain response time guarantees.

Graph Coverage. Knowledge graph quality depended on accurate entity extraction. Missed or incorrectly linked entities reduced the effectiveness of graph based retrieval.

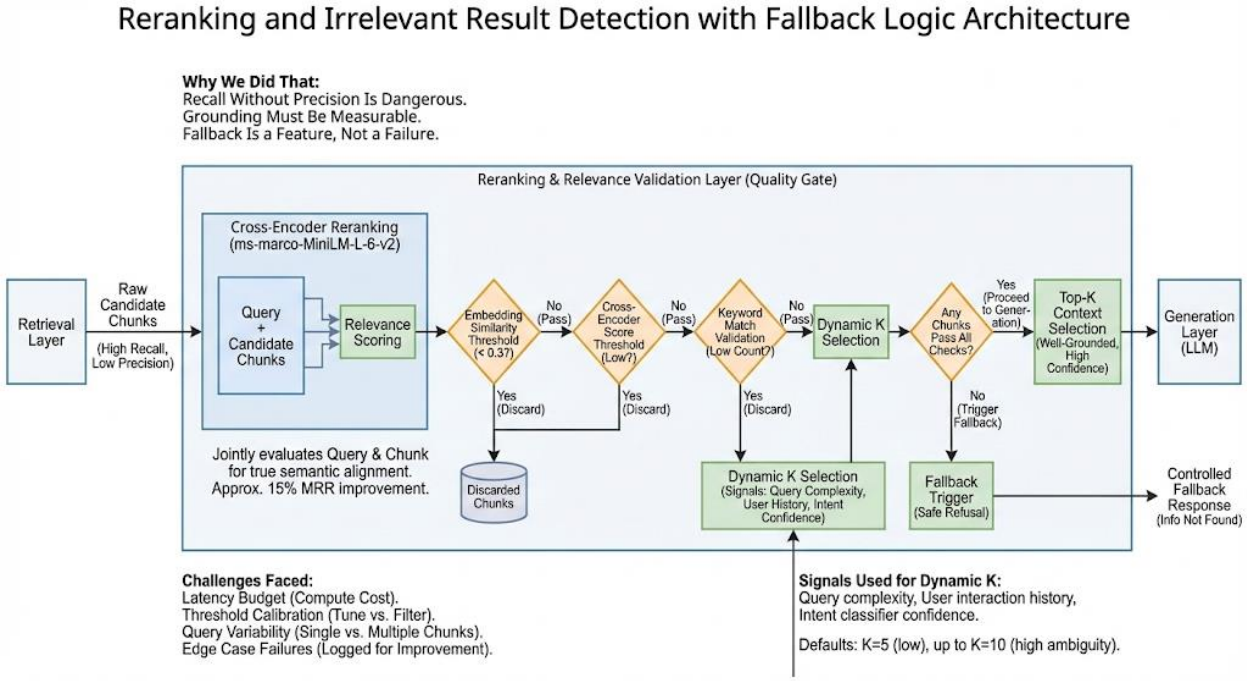
Deduplication. Combining results from multiple retrieval methods increased duplicate candidates. Additional normalization and ranking logic was required to present a clean and relevant result set.

Reranking and Irrelevant Result Detection with Fallback Logic

In Phase One, we implemented a dedicated reranking and relevance validation layer to ensure that only well grounded, high confidence context reached the generation stage.

Early experiments showed that raw retrieval results, even when recall was high, frequently included loosely related or misleading chunks that increased hallucination risk.

This layer acted as a quality gate between retrieval and generation, enforcing minimum relevance standards and triggering safe fallback behavior when grounding was insufficient.



Why We Did That

Recall Without Precision Is Dangerous. High recall retrieval improves coverage but increases the risk of passing irrelevant context to the LLM. Without reranking, the generation model confidently synthesized incorrect answers from weak evidence.

Grounding Must Be Measurable. The system needed objective criteria to decide when retrieved context was strong enough to support an answer and when it was not.

Fallback Is a Feature, Not a Failure. A safe refusal is preferable to a confident but incorrect response in a healthcare setting.

Re ranking

We selected cross encoder ms marco MiniLM L 6 v2 for reranking.

- Why. The model offered a strong balance between precision and latency, making it suitable for inline reranking during live calls.

- Behavior. The cross encoder evaluated the query and each candidate chunk jointly, producing a relevance score that reflected true semantic alignment rather than embedding distance alone.
- Impact. Offline evaluation showed an approximate fifteen percent improvement in Mean Reciprocal Rank after introducing reranking.

Top K Determination

Rather than using a fixed number of retrieved chunks, we implemented dynamic K selection to adapt to query characteristics.

Embedding Similarity Threshold

We applied a minimum cosine similarity threshold between the query embedding and retrieved chunk embeddings.

- Implementation. Chunks with cosine similarity below 0.3 were discarded.
- Purpose. This removed weak semantic matches that passed approximate search but did not carry meaningful relevance.

Cross Encoder Score Threshold

After reranking, we enforced a minimum relevance score from the cross encoder.

- Purpose. Chunks that failed to meet this threshold were excluded from the final context set, even if they ranked highly in vector search.

Keyword Match Validation

We introduced a lightweight keyword match count check.

- Purpose. For code based or entity heavy queries, this ensured that retrieved chunks contained a minimum number of exact term matches, preventing purely semantic but contextually incorrect results.

Fallback Trigger

If no chunks passed all relevance checks, the system returned a controlled fallback response indicating that the information was not found in the policy corpus, rather than attempting generation.

Signals Used

- Query complexity. Measured by length and named entity count extracted during query processing.
- User interaction history. Vague or repeated follow up queries were allocated more context to increase recall.
- Intent classifier confidence. Lower confidence scores triggered a larger candidate set to reduce the risk of missing relevant information.

Defaults

- The default K was set to five chunks.
- Under low confidence or high ambiguity conditions, K could increase up to ten.

Challenges Faced

Latency Budget. Cross encoder reranking introduced additional compute cost. Candidate set size must be tightly controlled to keep response time within SLA.

Threshold Calibration. Poorly chosen thresholds either filtered too aggressively or allowed weak context through. Extensive offline tuning was required to find stable operating points.

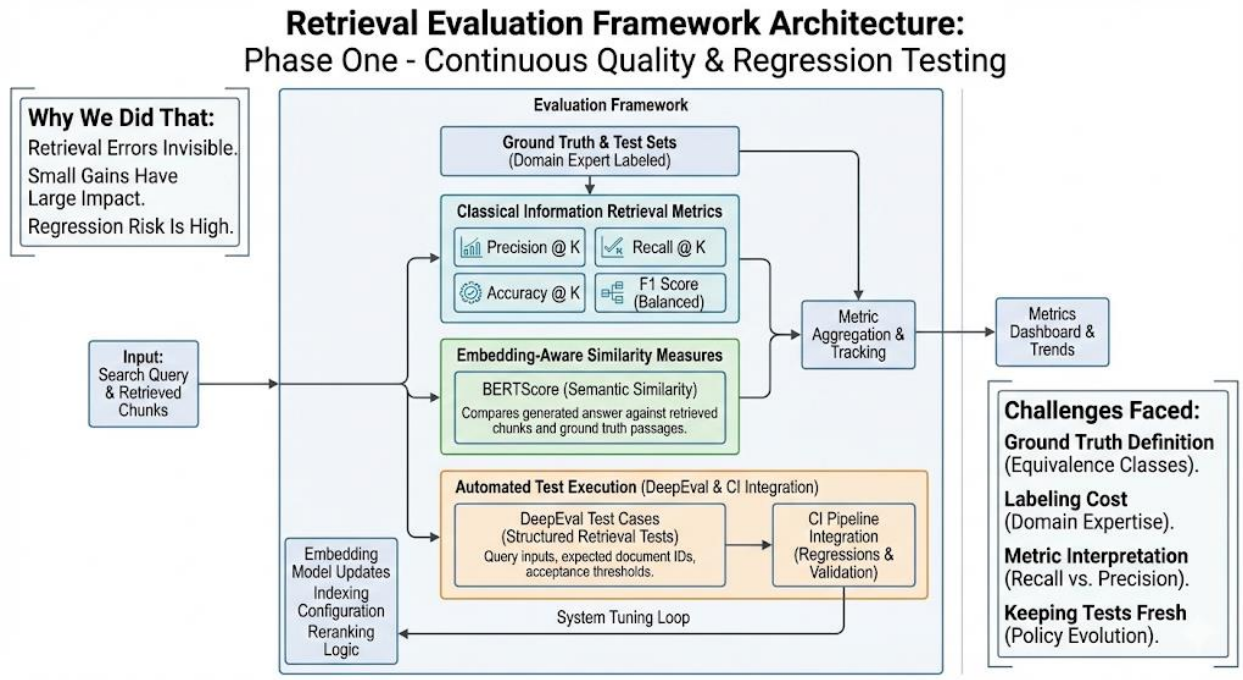
Query Variability. Some valid answers relied on a single highly relevant chunk. Others required multiple moderately relevant chunks. The gating logic had to handle both cases without bias.

Edge Case Failures. Rare queries produced no results above threshold despite the information existing in the corpus. These cases were logged and fed back into retrieval and embedding improvement workflows.

Evaluation of Retrieval

In Phase One, we implemented a formal retrieval evaluation framework to measure how effectively the system surfaced the correct policy context before generation. Retrieval evaluation was treated as equally important as generation evaluation, since downstream answer quality is bounded by the quality of retrieved evidence.

The evaluation strategy combined classical information retrieval metrics with embedding aware similarity measures and automated test execution integrated into the development lifecycle.



Why We Did That

Retrieval Errors Are Often Invisible. A generated answer can appear fluent and confident even when the correct policy section was never retrieved. Without explicit retrieval metrics, these failures go undetected.

Small Retrieval Gains Have Large Impact. Minor improvements in recall or ranking often produced outsized reductions in hallucinations and fallback rates.

Regression Risk Is High. Changes to chunking, embeddings, indexing, or reranking can silently degrade retrieval quality unless continuously measured.

Precision at K

Precision at K measured the proportion of retrieved chunks within the top K results that were truly relevant.

- Purpose. This metric quantified how much irrelevant context was entering the reranking and generation stages. High precision reduced noise and improved grounding.

Recall at K

Recall at K measured whether at least one relevant chunk appeared within the top K retrieved results.

- Purpose. This was the most critical metric for safety. If the correct policy content was not retrieved, generation could not succeed regardless of prompt quality.

Accuracy at K

Accuracy at K measured whether the top ranked result contained the correct answer context.

- Purpose. This metric reflected how well the ranking logic surfaced the most useful chunk first, which directly impacted answer clarity and citation quality.

F1 Score

The F1 score balanced precision and recall into a single measure.

- Purpose. This provided a stable comparison signal when tuning retrieval parameters such as chunk size, overlap, or index configuration.

BERTScore

We used BERTScore to evaluate semantic similarity between retrieved chunks and ground truth policy passages.

- Purpose. This captured relevance beyond exact text overlap, which was especially important for paraphrased queries and policy language variations.
- Benefit. Embedding based evaluation surfaced near misses where retrieval was semantically close but not sufficient for correct grounding.

DeepEval

We used DeepEval to define structured retrieval test cases and execute them automatically.

- Usage. Test cases included query inputs, expected relevant document identifiers, and acceptance thresholds for retrieval metrics.

- Integration. Retrieval evaluations were integrated into the CI pipeline so that changes to embeddings, indexing, or retrieval logic were validated before deployment.
- Benefit. This prevented silent regressions and made retrieval quality a first class release criterion rather than a post deployment concern.

Additional Tooling

We supplemented these evaluations with offline scripts and internal dashboards to track metric trends over time. Retrieval metrics were correlated with generation hallucination rates and fallback frequency to validate system level impact.

Challenges Faced

Ground Truth Definition. Many policy questions mapped to multiple acceptable passages. Evaluation logic had to allow for equivalence classes rather than single correct chunks.

Labeling Cost. Creating and maintaining high quality retrieval test sets required domain expertise and regular updates as policies evolved.

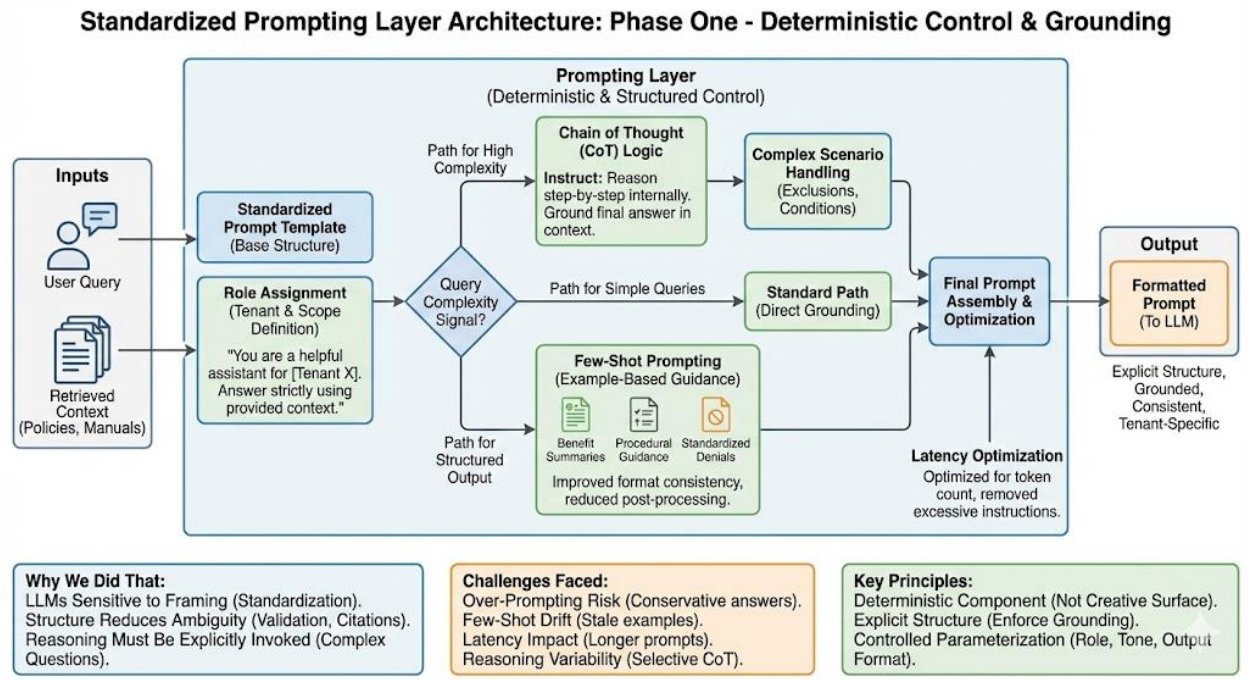
Metric Interpretation. High recall with low precision was not always preferable. Thresholds had to be interpreted in the context of reranking and fallback behavior.

Keeping Tests Fresh. As new documents were ingested, test cases had to be updated to reflect current policy structure rather than outdated references.

Prompting

In Phase One, we designed the prompting layer to tightly control how the LLM interpreted retrieved context and generated responses. Prompting was treated as a deterministic system component rather than a creative surface, with explicit structure to enforce grounding, consistency, and tenant specific behavior.

The prompt template was standardized across tenants while allowing controlled parameterization for role, tone, and output format. This ensured predictable behavior across use cases while supporting enterprise level customization.



Why We Did That

LLMs Are Sensitive to Framing. Small variations in prompt wording produced materially different behavior in early experiments. Without standardization, answer quality and safety varied across requests.

Structure Reduces Ambiguity. Free form responses made it difficult to validate outputs, enforce citations, or integrate responses into downstream workflows.

Reasoning Must Be Explicitly Invoked. Complex policy questions often require multi step reasoning across retrieved clauses. Without guidance, the model either skipped steps or produced shallow summaries.

Role Assignment

Each prompt explicitly defined the assistant's role and scope.

- Implementation. The system prompt specified that the model was a helpful assistant for a specific tenant and must answer strictly using the provided policy context.
- Effect. This reduced cross tenant leakage, discouraged the use of external knowledge, and aligned responses with organizational tone and responsibility.

Few Shot Prompting

Few shot examples were added for queries requiring structured or repeatable outputs.

- Use Cases. Benefit eligibility summaries, step by step procedural guidance, and standardized denial explanations.
- Effect. Few shot prompting improved format consistency and reduced post processing complexity in the UI layer.

Chain of Thought for Complex Queries

For multi clause or conditional policy questions, we enabled guided reasoning.

- Implementation. The prompt instructed the model to reason step by step internally before producing a final answer grounded in retrieved text.
- Effect. This improved accuracy on complex scenarios involving exclusions, prerequisites, or layered coverage rules. It also reduced logical errors caused by skipping intermediate conditions.

Challenges Faced

Over Prompting Risk. Excessive instructions occasionally constrained the model too tightly, leading to overly conservative or incomplete answers. Iterative tuning was required to balance safety and usefulness.

Few Shot Drift. Example based prompting required regular review as policies and response standards evolved. Stale examples could bias answers incorrectly.

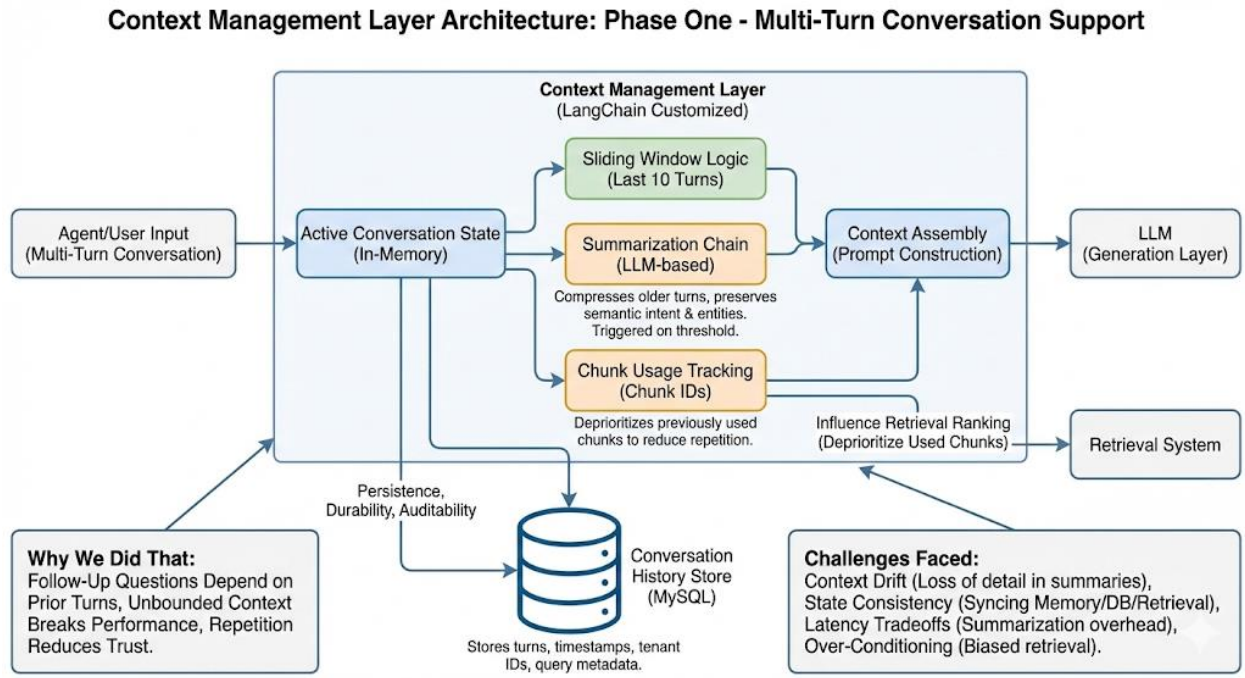
Latency Impact. Longer prompts increased token count and response time. Prompts were optimized to include only instructions that materially improved behavior.

Reasoning Variability. Some queries did not benefit from explicit reasoning steps. Logic was added to enable chain of thought selectively based on query intent and complexity signals.

Context Management

In Phase One, we implemented a dedicated context management layer to support multi turn conversations without degrading retrieval quality or latency. Call center interactions often involve follow up questions, clarifications, and incremental refinement of intent. Without explicit context handling, the system either lost continuity or repeatedly retrieved the same information.

The context layer was designed to preserve conversational relevance while enforcing strict limits on prompt size, memory growth, and repetition.



Why We Did That

Follow Up Questions Depend on Prior Turns. Agents frequently ask short follow ups such as what about referrals or does this apply to dependents. These queries are unintelligible without conversation history.

Unbounded Context Breaks Performance. Passing full conversation history to the LLM quickly increases token usage, latency, and cost, while often adding little incremental signal.

Repetition Reduces Trust. Re surfacing the same policy chunks across turns made the assistant feel unaware of prior responses and frustrated agents.

Conversation History Storage

We persisted conversation history in a relational store to support durability and auditability.

- Implementation. MySQL was used to store conversation turns, timestamps, tenant identifiers, and query metadata.
- Windowing Strategy. A sliding window of the last ten turns was maintained for active conversations. Older turns were excluded from the live prompt unless explicitly summarized.

Summarization for Long Conversations

For extended interactions, we introduced summarization to compress historical context.

- Implementation. A lightweight LLM based summarization chain condensed older turns into a concise semantic summary.
- Behavior. The summary preserved user intent, key entities, and prior conclusions while removing redundant phrasing and conversational noise.
- Benefit. This allowed long conversations to remain coherent without exceeding prompt size limits.

Chunk Usage Tracking

We tracked which document chunks had already been used in prior turns.

- Implementation. Chunk identifiers were stored alongside conversation state.
- Behavior. During retrieval, previously used chunks were deprioritized unless the query intent explicitly required revisiting them.
- Benefit. This reduced repetition and encouraged retrieval of complementary or missing context in follow up turns.

Memory Framework

We leveraged LangChain memory abstractions as the baseline framework.

- Customization. Default memory behavior was extended with custom summarization chains, chunk tracking logic, and tenant aware isolation.
- Result. This provided flexibility without locking the system into rigid memory semantics.

Challenges Faced

Context Drift. Summaries occasionally lost subtle constraints or exclusions mentioned earlier. Summarization prompts had to be tuned to preserve policy critical details.

State Consistency. Keeping MySQL conversation state, in memory context, and retrieval behavior synchronized required careful transaction handling.

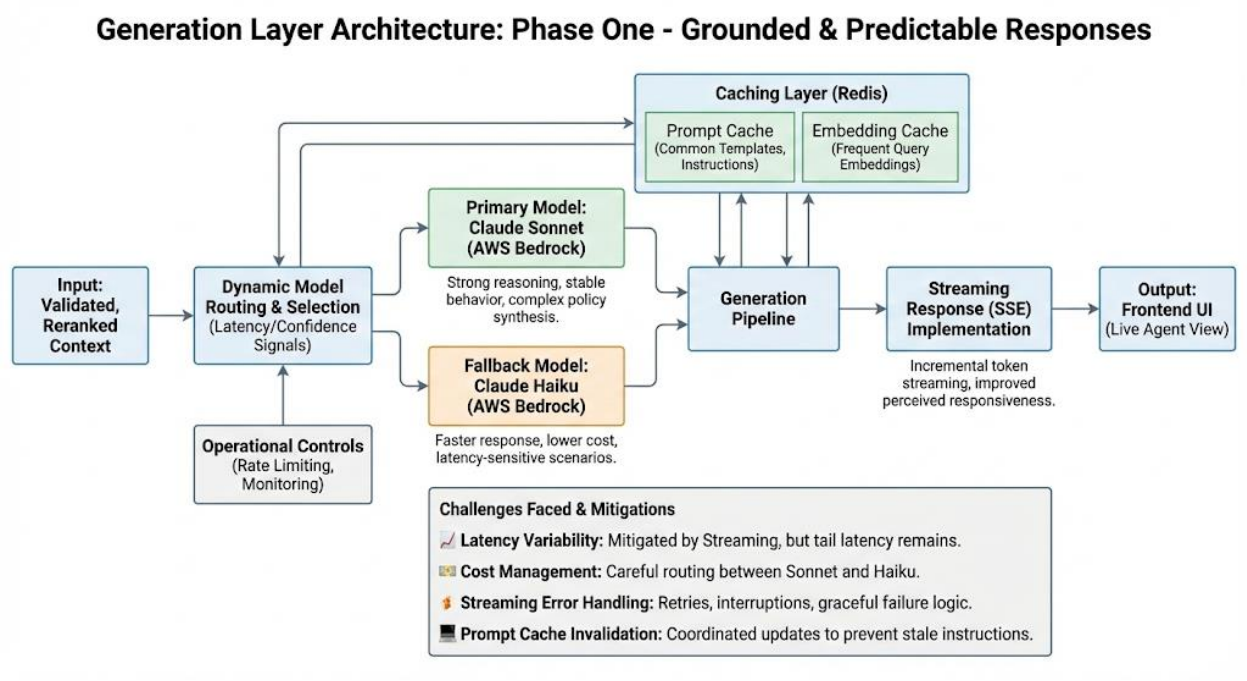
Latency Tradeoffs. Summarization introduced additional LLM calls. Logic was added to trigger summarization only when conversation length exceeded defined thresholds.

Over Conditioning. In some cases, too much prior context biased retrieval toward earlier topics even when the user intent had shifted. Reset and decay logic was required to allow clean topic transitions.

Generation

In Phase One, we designed the generation layer to produce grounded, policy aligned responses with predictable latency suitable for live call center usage. Model selection, invocation patterns, and response delivery were optimized to balance answer quality, cost, and responsiveness.

The generation layer was treated as the final step in a controlled pipeline. It consumed only validated, reranked context and enforced strict prompt constraints to prevent the use of external knowledge.



Primary Model

Claude Sonnet was selected as the primary generation model through AWS Bedrock.

Rationale. Sonnet provided strong reasoning ability, stable behavior under structured prompting, and high quality language generation for complex policy questions. It consistently performed well when synthesizing multi clause answers grounded in retrieved text.

Fallback Model

Claude Haiku was configured as a fallback model for latency sensitive scenarios.

Rationale. Haiku offered faster response times and lower cost, making it suitable when prompt complexity was low or when Sonnet was unavailable.

Prompt Caching

To reduce repeated computation and improve latency, we introduced prompt level caching.

- Implementation. Common system prompts, role definitions, and frequently used instruction blocks were cached in Redis.
- Behavior. Cached prompt components were reused across requests, reducing token processing overhead and improving time to first token.

Embedding Cache

Frequently repeated query embeddings were also cached in Redis.

- Benefit. This reduced redundant embedding computation for common or repeated agent questions, especially during high traffic periods.

Streaming Responses

We implemented streaming output to improve perceived responsiveness.

- Implementation. Server sent events were used to stream tokens incrementally to the frontend as they were generated.
- Benefit. Agents saw partial answers almost immediately, reducing perceived wait time and improving usability during live calls.

Operational Controls

Generation requests were rate limited and monitored to prevent overload. Model selection logic dynamically chose between Sonnet and Haiku based on confidence signals, prompt length, and latency constraints.

Challenges Faced

Latency Variability. Even with caching, generation latency varied depending on prompt size and model behavior. Streaming mitigated perception but did not eliminate tail latency.

Cost Management. Sonnet provided higher quality but at higher cost. Careful routing logic was required to use Haiku where appropriate without degrading answer quality.

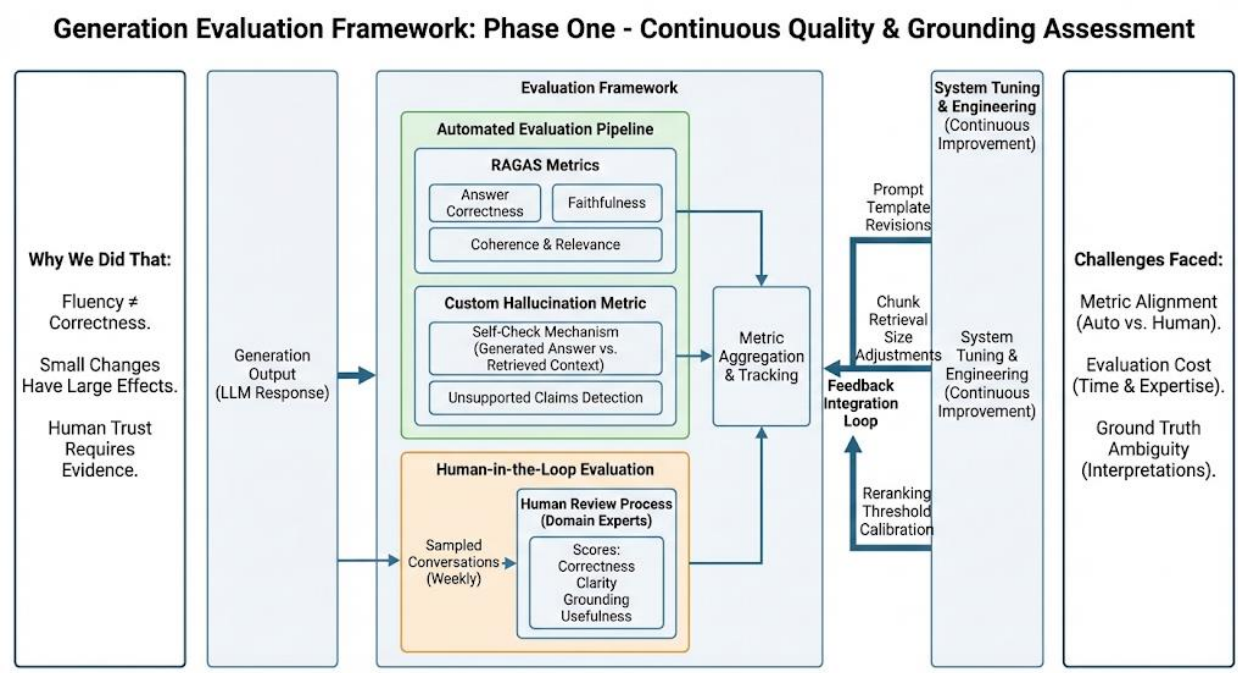
Streaming Error Handling. Managing partial outputs required additional logic to handle retries, interruptions, and graceful failure when a stream was terminated early.

Prompt Cache Invalidation. Changes to prompt templates required coordinated cache invalidation to prevent serving stale instructions.

Evaluation of Generation

In Phase One, we implemented a structured evaluation framework to measure generation quality beyond subjective feedback. The goal was to ensure that responses were correct, grounded in retrieved context, and consistent across time as the system evolved.

Generation evaluation was treated as a continuous process rather than a one time validation step. Metrics were tracked across deployments and used to guide prompt, retrieval, and reranking adjustments.



Why We Did That

Fluency Does Not Equal Correctness. LLMs can produce confident, well written answers that are factually incorrect or poorly grounded. Explicit evaluation was required to detect these failures.

Small Changes Have Large Effects. Minor updates to prompts, chunking, or reranking thresholds could materially impact answer quality. Without metrics, regressions would go unnoticed.

Human Trust Requires Evidence. In a regulated environment, leadership and agents needed measurable proof that the system was improving over time.

Automated Evaluation

RAGAS Metrics

We used RAGAS to evaluate core RAG specific quality dimensions.

- Answer Correctness. Measured whether the generated answer accurately reflected the retrieved policy content.
- Faithfulness. Evaluated whether claims in the answer were supported by the provided context rather than model prior knowledge.
- Coherence and Relevance. Assessed logical flow and alignment with the user question.
- These metrics provided a standardized baseline for tracking generation quality across releases.

Custom Hallucination Metric

We implemented a custom hallucination score to detect unsupported claims.

- Implementation. The generated answer was automatically compared against retrieved document chunks using a self check mechanism. Statements not supported by the retrieved context were penalized.
- Purpose. This metric surfaced subtle hallucinations that passed general relevance checks but introduced unsupported details.

Human in the Loop Evaluation

We complemented automated metrics with regular human review.

- Process. Each week, a sampled set of conversations was reviewed and rated by domain knowledgeable reviewers.
- Criteria. Reviewers scored answers on correctness, clarity, grounding, and usefulness for live call handling.

Feedback Integration

Evaluation results were fed back into system tuning.

- Usage. Metrics informed prompt template revisions, chunk retrieval size adjustments, and reranking threshold calibration.
- Benefit. This closed the loop between evaluation and engineering, enabling deliberate and measurable improvement rather than ad hoc tuning.

Challenges Faced

Metric Alignment. Automated scores did not always align perfectly with human judgment. Calibration and combined interpretation were required.

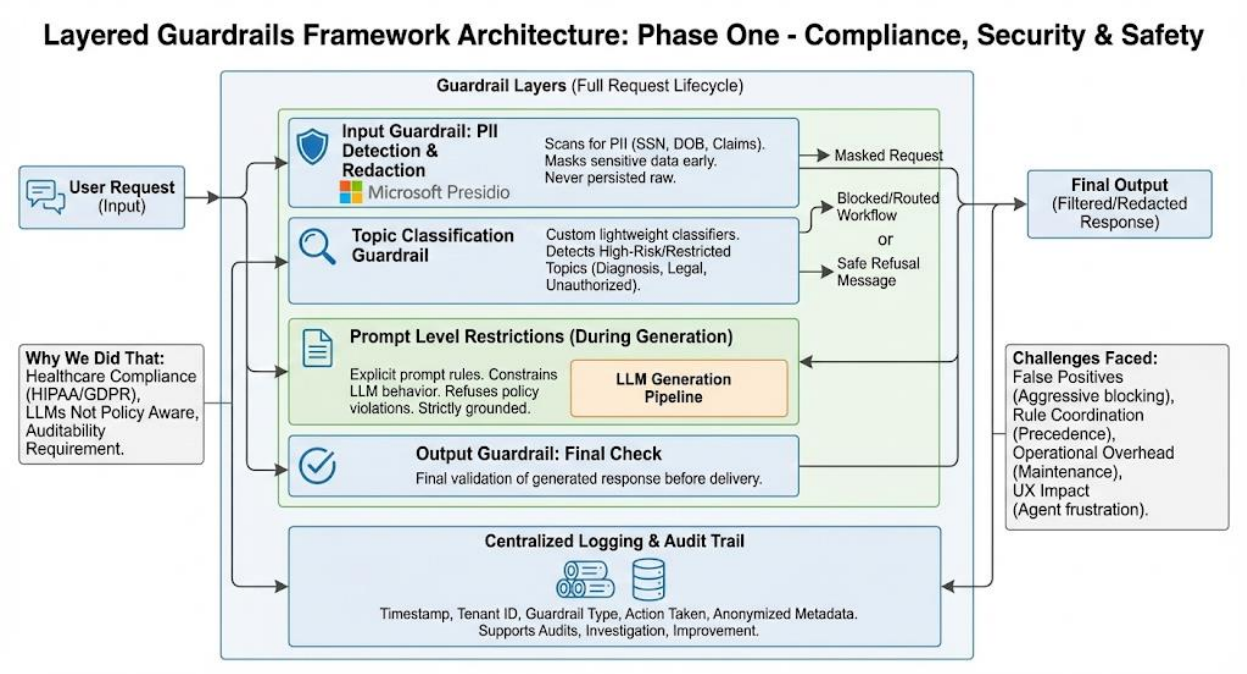
Evaluation Cost. Human review was time intensive and required domain expertise. Sampling strategies had to balance coverage and effort.

Ground Truth Ambiguity. Some policy questions had multiple acceptable interpretations. Evaluation guidelines had to be carefully defined to avoid false negatives.

Guardrails

In Phase One, we implemented a layered guardrail framework to ensure regulatory compliance, protect sensitive data, and prevent unsafe or inappropriate system behavior. Guardrails were enforced across the entire request lifecycle, from user input to generation output, rather than relying on a single control point.

The design assumed that failures would occur and focused on early detection, safe blocking, and full auditability rather than silent correction.



Why We Did That

Healthcare Compliance Is Non Negotiable. The system operates in an environment governed by HIPAA and GDPR, where improper handling of personal or sensitive data can lead to severe legal and operational consequences.

LLMs Are Not Policy Aware by Default. Without explicit controls, models may respond to requests that violate internal policy, compliance rules, or acceptable use standards.

Auditability Is a Requirement. When a response is blocked or altered, there must be a clear and traceable explanation for why the decision was made.

PII Detection and Redaction

Microsoft Presidio was used as the primary mechanism for identifying and masking sensitive data.

- Scope. Queries were scanned for member identifiers, social security numbers, dates of birth, claim numbers, and other regulated attributes.
- Behavior. Detected PII was masked before any downstream processing and never persisted in raw form.
- Benefit. This ensured that sensitive data never reached the LLM or was stored in logs.

Compliance Sensitive Topic Classification

We introduced custom lightweight classifiers to detect restricted or high risk query categories.

- Examples. Requests involving diagnosis, treatment advice, legal interpretation, or internal operational data beyond agent authorization.
- Behavior. Queries flagged by classifiers were either blocked, routed to approved workflows, or returned with a safe refusal message.

Prompt Level Restrictions

Explicit prompt rules were used to constrain model behavior.

- Implementation. The system prompt clearly instructed the model to refuse certain query types and to answer strictly from provided context.
- Effect. This reduced the likelihood of policy violations even when upstream classifiers failed.

Logging and Audit Trail

All guardrail triggers were logged centrally.

- Captured Data. Timestamp, tenant identifier, guardrail type triggered, action taken, and anonymized query metadata.
- Purpose. These logs supported compliance audits, incident investigation, and continuous improvement of detection logic.

Challenges Faced

False Positives. Early versions of PII detection and classifiers were overly aggressive, blocking legitimate agent queries. Tuning and allow lists were required to balance safety and usability.

Rule Coordination. Multiple guardrails could trigger on a single request. Clear precedence rules were needed to ensure consistent outcomes.

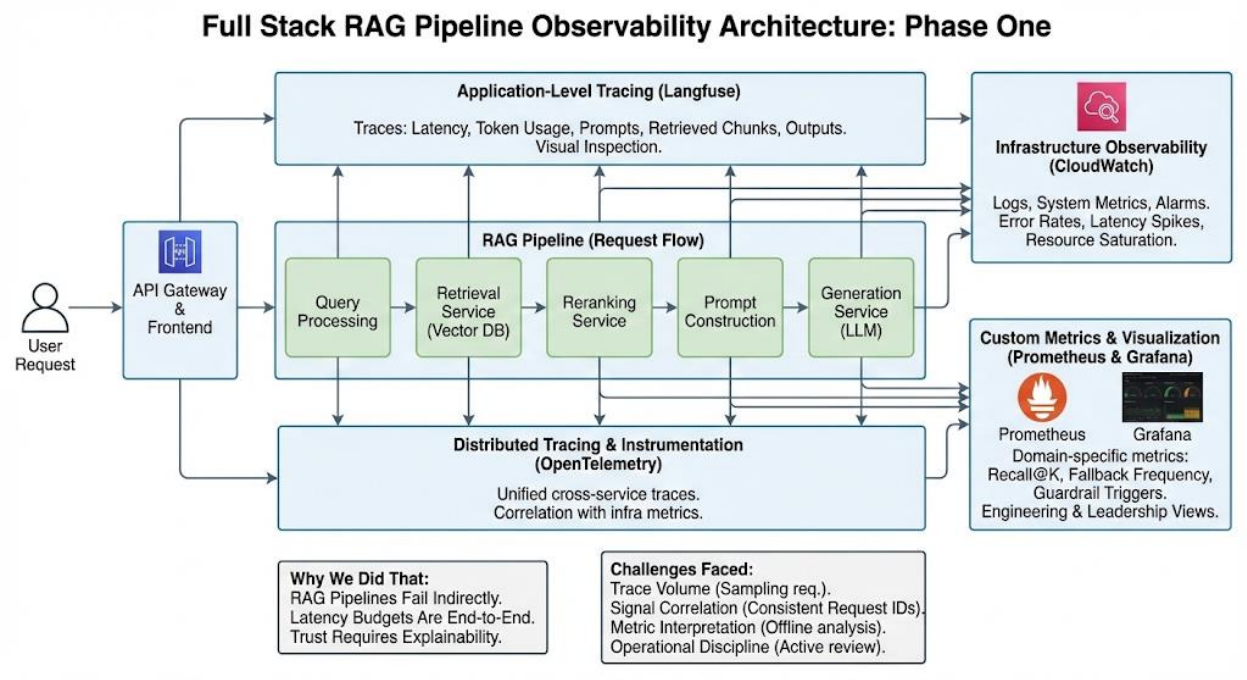
Operational Overhead. Maintaining classifiers, prompt rules, and audit logic introduced ongoing maintenance cost. Processes were established to review logs and update rules based on real usage patterns.

User Experience Impact. Hard blocks can frustrate agents during live calls. Careful messaging and fallback guidance were required to maintain trust while enforcing compliance.

Tracing and Observability

In Phase One, we implemented a full observability stack to provide end to end visibility into request flow, system performance, and model behavior. Given the number of moving parts in a RAG pipeline, silent failures or latent degradation were unacceptable. Observability was treated as a first class system requirement rather than a debugging aid.

The tracing and monitoring layer was designed to answer three core questions at all times. What happened, where did it happen, and why did it happen.



Why We Did That

RAG Pipelines Fail Indirectly. Retrieval, reranking, prompting, and generation are tightly coupled. A small issue upstream often manifests as a downstream quality problem that is difficult to diagnose without full trace visibility.

Latency Budgets Are End to End. A fast model does not matter if retrieval or reranking silently adds delay. We needed per stage timing to enforce call center SLAs.

Trust Requires Explainability. When agents or leadership questioned an answer, we needed the ability to reconstruct the full request lifecycle, including retrieved chunks, scores, and model decisions.

Langfuse

Langfuse was used as the primary application level tracing tool for LLM workflows.

- Coverage. Each request was traced across query processing, retrieval, reranking, prompt construction, and generation.
- Signals. Latency per stage, token usage, prompt versions, retrieved chunks, and final outputs.
- Benefit. Langfuse made it possible to visually inspect why a response was slow or incorrect and tie behavior back to specific retrieval or prompt decisions.

CloudWatch

CloudWatch was used for infrastructure level observability.

- Coverage. Application logs, system metrics, and alarms for backend services.
- Usage. Alerts were configured for error rates, latency spikes, and resource saturation.
- Benefit. This provided operational awareness and rapid detection of system level issues independent of model behavior.

Prometheus and Grafana

We introduced Prometheus and Grafana for custom, domain specific metrics.

- Metrics Tracked. Retrieval latency, recall at K, reranking acceptance rates, fallback frequency, and guardrail trigger rates.
- Dashboards. Grafana dashboards were built for both engineering and leadership views, enabling ongoing performance tracking.
- Benefit. These metrics exposed gradual degradation that would not surface as hard failures.

OpenTelemetry

OpenTelemetry was used to unify distributed tracing across microservices.

- Coverage. Requests were traced from the frontend through API gateways, retrieval services, vector database calls, and model invocation.
- Benefit. This enabled correlation of application traces with infrastructure metrics, making cross service bottlenecks visible.

Challenges Faced

Trace Volume. High traffic produced large volumes of trace data. Sampling strategies were required to balance visibility and cost.

Signal Correlation. Mapping Langfuse traces to infrastructure level traces required consistent request identifiers across systems.

Metric Interpretation. Some metrics such as recall at K required offline computation and careful interpretation to avoid misleading conclusions.

Operational Discipline. Observability only provided value when dashboards and alerts were actively reviewed. Processes had to be established to ensure signals led to action rather than passive monitoring.

Drift Measuring – Not Important (Nice to Have)

In Phase One, we implemented a drift measurement framework to detect gradual degradation in retrieval and generation quality as data, usage patterns, and models evolved. Drift was treated as an expected operational reality rather than an exceptional failure. Without explicit monitoring, embedding quality and LLM behavior can decay silently while appearing stable at the surface.

Embedding Drift Detection

We monitored changes in embedding distributions over time.

- Method. Query and document embeddings were projected using PCA and clustered to observe shifts in density and separation.
- Signal. Significant movement or cluster collapse indicated potential degradation in semantic separation.

LLM Output Drift

We tracked changes in generation behavior over time.

- Method. Response length, citation count, refusal rate, and answer structure were monitored and compared across time windows.
- Purpose. This surfaced subtle changes in tone or completeness that were not captured by correctness metrics alone.

Embedding Distribution Drift

We measured statistical properties of embedding vectors, including mean, variance, and nearest neighbor distances.

- Purpose. Distributional shifts often preceded observable retrieval failures and acted as early warning signals.

Query Pattern Drift

We analyzed changes in incoming query characteristics.

- Signals. Query length, entity density, intent class distribution, and language usage.
- Purpose. Shifts in agent behavior or policy focus often required retrieval and prompt retuning.

Answer Quality Drift

Automated metrics and human review scores were tracked longitudinally.

- Purpose. Declining trends triggered deeper investigation even when absolute scores remained above thresholds.

Multi Tenancy – Not Important (Nice to Have)

In Phase One, we designed the system to be multi tenant by default, with strict isolation guarantees across data storage, retrieval, and access control. This was a non negotiable requirement due to state and country specific regulations governing healthcare data usage and residency.

Multi tenancy was treated as a foundational architectural concern rather than an access control afterthought. Every layer of the system enforced tenant boundaries explicitly.

Vector Store Isolation

We enforced tenant level isolation in the vector database.

- Implementation. Each tenant was assigned a separate Milvus collection.
- Behavior. Queries were executed only against the collection associated with the active tenant, eliminating the possibility of cross tenant retrieval by design.
- Benefit. This simplified access control and improved retrieval performance by reducing search scope.

Metadata Based Filtering

In addition to physical separation, metadata filters were applied at query time.

- Implementation. Tenant identifiers were included as mandatory filters in retrieval queries and validated before execution.
- Purpose. This provided defense in depth and protected against misrouted requests or configuration errors.

Multimodal Support – Not Important (Nice to Have)

In Phase One, we designed the retrieval architecture with explicit support for multimodal content, even though initial production usage focused on text. Healthcare policy knowledge is not purely textual.

Benefit tables, scanned forms, diagrams, and annotated images often carry meaning that cannot be reliably captured through text extraction alone.

Rather than treating multimodal retrieval as a future rewrite, we introduced it as an extension of the existing retrieval pipeline with clear separation of concerns and shared fusion logic.

Why We Did That

Policies Are Not Text Only. Coverage rules and operational guidance are frequently embedded in tables, scanned claim forms, and visual layouts where spatial structure matters as much as wording.

Text Extraction Has Hard Limits. Even with strong parsers, some information is lost when images or tables are flattened into text. Native image representations preserve meaning that text cannot.

Future Proofing Without Rework. Designing multimodal support early avoided tight coupling between retrieval logic and text only assumptions. This reduced long term architectural risk.

Image Embeddings

We introduced image embedding support using CLIP style models.

- Implementation. Images extracted from documents were embedded using CLIP and stored in Milvus alongside text vectors, in separate modality specific collections.
- Behavior. Image embeddings captured visual and semantic similarity, enabling retrieval of relevant forms, tables, or diagrams based on conceptual queries rather than filenames or captions.

Text and Image Retrieval

Text and image retrieval paths were kept logically separate but combined at ranking time.

- Implementation. Text queries executed against text vector indexes, while image similarity searches executed against image embedding indexes. Results were fused downstream using the same ranking framework.
- Benefit. This allowed multimodal retrieval without forcing text and image vectors into an artificial shared space prematurely.

Future Roadmap

- Audio and Video Support
- We planned support for audio and video based knowledge sources.
- Approach. Audio content would be transcribed using Whisper, while acoustic embeddings would be generated using CLAP style models.

- Use Cases. Recorded training sessions, policy walkthroughs, and call recordings annotated with operational guidance.

Multi Language Support – Not Important (Nice to Have)

In Phase One, we implemented multi language support to ensure the system could serve agents handling members across diverse linguistic backgrounds. Member inquiries are not limited to English, and policy guidance must remain accurate and grounded regardless of query language.

The multi language design focused on correctness, isolation, and predictable routing rather than attempting automatic translation as a first step.

Why We Did That

Language Mismatch Breaks Retrieval. Embedding an English query against non English content or vice versa produces weak similarity signals and unreliable results. Correct language alignment was required for stable retrieval.

Regulatory and Regional Variation. Different regions may maintain policy documents in different languages. Mixing languages within a single retrieval space increased the risk of returning context from the wrong jurisdiction.

Agent Efficiency. Agents should not be required to translate queries manually or guess which language index to search. The system needed to route queries automatically and transparently.

Embedding Strategy

We introduced a multilingual embedding model to support semantic similarity across languages.

- Model Selection. paraphrase multilingual MiniLM L12 v2 was evaluated and selected for its support of more than fifty languages and its strong performance on cross language semantic tasks.
- Usage. Documents and queries were embedded using the same multilingual model within language specific collections to maintain alignment.
- Benefit. This enabled consistent semantic retrieval within each language while preserving future flexibility for cross language expansion if required.

Language Detection

Incoming queries were automatically analyzed to determine language.

- Implementation. langdetect was used as the primary detector, with fasttext evaluated to validate and improve accuracy on short or ambiguous queries.
- Behavior. Language detection occurred early in the request pipeline and was treated as a routing signal rather than a soft hint.

Routing Logic

Based on detected language, queries were routed to the appropriate language specific index.

- Implementation. Separate vector indexes were maintained per language and tenant. For example, Spanish queries were routed exclusively to the Spanish index.
- Benefit. This eliminated cross language noise and ensured that retrieved context matched both linguistic and regulatory expectations.

Personalization – Not Important (Nice to Have)

In Phase One, we introduced controlled personalization to improve retrieval relevance based on individual agent behavior while preserving tenant isolation and compliance constraints. Personalization was applied as a ranking signal rather than a hard filter to avoid narrowing the retrieval space too aggressively.

The design goal was to adapt to how agents actually search and interact with policy content, without compromising grounding, auditability, or fairness.

Why We Did That

Agents Develop Usage Patterns. Over time, agents tend to handle similar issue types, plans, or regions. Ignoring this signal caused the system to repeatedly surface generic results instead of the most practically useful ones.

Relevance Is Contextual to the User. Two agents asking the same question may benefit from different supporting documents based on prior interactions and expertise.

Personalization Should Not Override Policy Truth. Boosting relevance must never cause exclusion of authoritative content or introduce bias toward outdated material.

User Embeddings

We learned lightweight user embeddings derived from historical interaction data.

- Source Signals. Query history, clicked documents, and explicit ratings where available.

- Behavior. Each agent was represented as a vector capturing recurring semantic interests rather than exact queries.
- Storage. User vectors were stored in a separate Milvus collection, fully isolated from document embeddings.

Personalized Retrieval Boosting

Personalization was applied during ranking rather than initial retrieval.

- Implementation. Retrieved document chunks that were previously viewed, clicked, or positively rated by the user received a relevance boost.
- Constraint. Boosting was bounded and never allowed to surface content that failed baseline relevance or grounding thresholds.
- Benefit. This improved result ordering without reducing recall or introducing blind spots.

Challenges Faced

Cold Start Problem. New agents lacked interaction history. The system defaulted to non personalized retrieval until sufficient signal was collected.

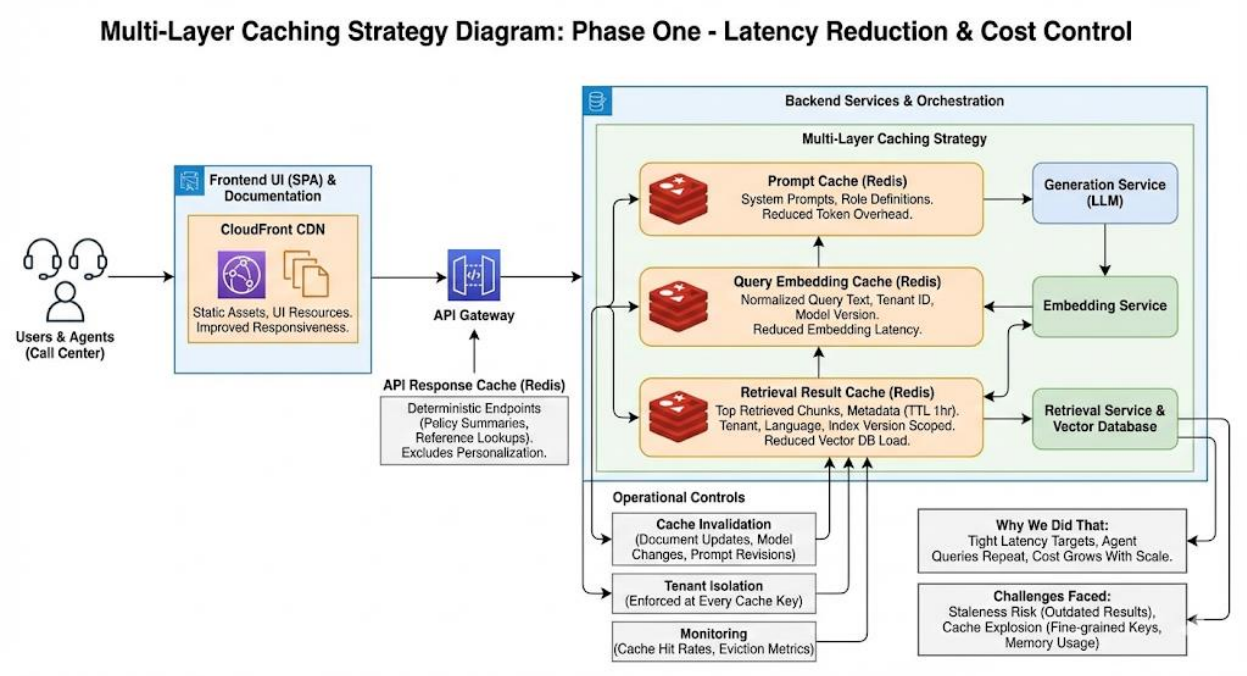
Bias Risk. Over boosting familiar documents risked reinforcing narrow knowledge exposure. Strict caps and decay logic were required.

Privacy Considerations. Interaction data had to be carefully scoped and anonymized to ensure it did not capture sensitive member information.

Caching

In Phase One, we implemented a multi layer caching strategy to reduce latency, control cost, and stabilize performance under peak call center load. Caching was applied selectively across the request lifecycle, focusing on deterministic outputs and high repetition paths rather than speculative optimization.

The caching design emphasized correctness and safety first. Cached artifacts were scoped, versioned, and invalidated carefully to avoid serving stale or cross tenant data.



Why We Did That

Latency Targets Are Tight. Live call handling requires fast responses. Even small delays across embedding, retrieval, or generation stages compound quickly.

Agent Queries Repeat. A significant portion of agent questions cluster around common benefits, procedures, and eligibility rules. Recomputing identical results wasted compute and increased tail latency.

Cost Grows With Scale. Embedding generation, retrieval, and LLM calls all incur cost. Caching provided a direct and measurable way to reduce recurring overhead.

Query Embedding Cache

We cached query embeddings to avoid repeated embedding computation for identical or near identical queries.

- Implementation. Redis was used as an in memory cache keyed by normalized query text, tenant identifier, and embedding model version.
- Benefit. This reduced embedding latency for frequent queries and stabilized response times during traffic spikes.

Retrieval Result Cache

We cached retrieval outputs for high frequency queries.

- Implementation. The top retrieved chunks and associated metadata were cached with a time to live of one hour.
- Scope. Cache keys included tenant, language, retrieval configuration, and index version to prevent stale or mis scoped results.
- Benefit. This reduced load on the vector database and improved consistency for common lookups.

Prompt Cache

We cached static and semi static prompt components.

- Implementation. System prompts, role definitions, and reusable instruction blocks were stored in Redis and assembled dynamically at request time.
- Benefit. This reduced token processing overhead and improved time to first token during generation.

API Response Cache

We introduced response level caching for deterministic endpoints.

- Scope. Non conversational API responses such as policy summaries or reference lookups were cached when safe to do so.
- Constraint. Responses involving personalization or live conversation state were excluded to avoid incorrect reuse.

Content Delivery Network

- Static frontend assets and documentation were served through a CDN.
- Implementation. CloudFront was used to distribute UI assets and static resources.
- Benefit. This reduced load on backend services and improved UI responsiveness for agents across regions.

Operational Controls

- Cache entries were versioned and invalidated on document updates, embedding model changes, or prompt revisions.
- Tenant isolation was enforced at every cache key level to prevent cross tenant leakage.
- Cache hit rates and eviction metrics were monitored continuously to validate effectiveness.

Challenges Faced

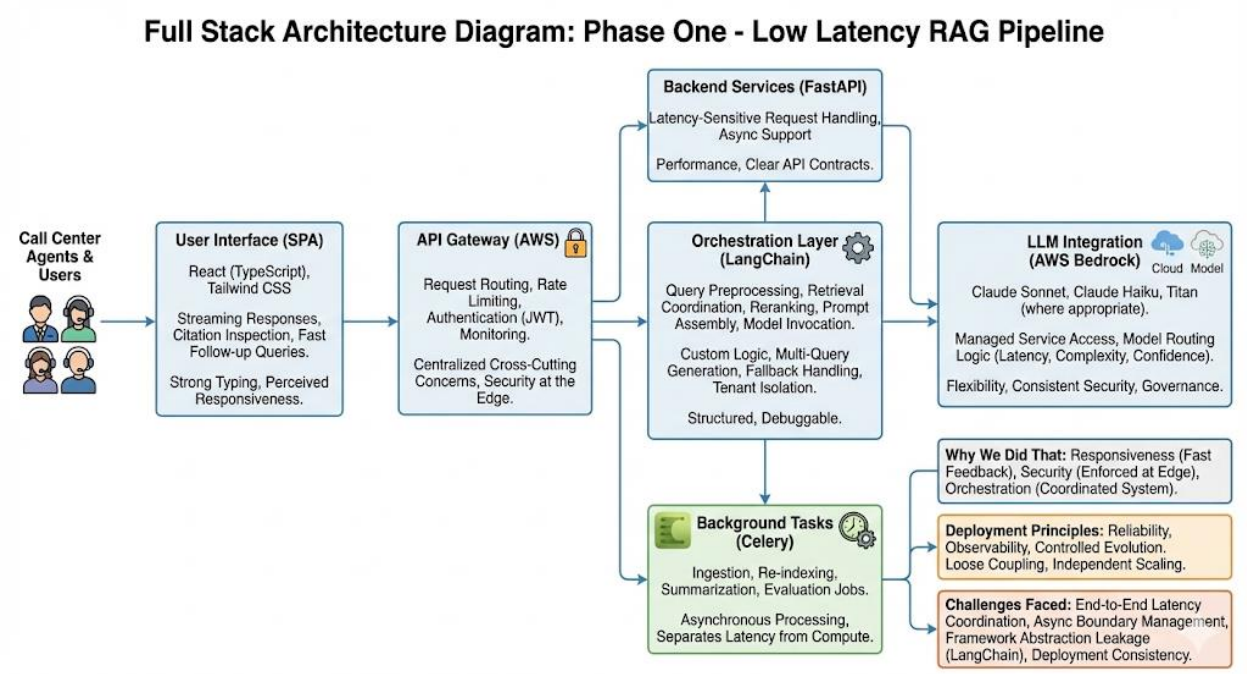
Staleness Risk. Cached retrieval results could become outdated after policy updates. Tight invalidation hooks tied to ingestion and indexing events were required.

Cache Explosion. Fine grained cache keys increased memory usage. Careful normalization and eviction policies were needed to control growth.

Full Stack Architecture

In Phase One, we implemented a full stack architecture that supported low latency interaction, secure access, and scalable orchestration of the RAG pipeline. The system was designed as a set of loosely coupled services with clear boundaries between user experience, orchestration, and model execution.

The full stack design prioritized reliability, observability, and controlled evolution. Each layer could be scaled or modified independently without cascading changes across the system.



Why We Did That

Call Center Usage Demands Responsiveness. Agents require fast feedback during live calls. UI rendering, API handling, and backend orchestration all needed to minimize perceived and actual latency.

Security and Access Control Are Mandatory. Authentication, authorization, and rate limiting could not be embedded ad hoc into application logic. They needed to be enforced consistently at the platform edge.

RAG Pipelines Are Orchestrated Systems. Retrieval, reranking, prompting, and generation must be coordinated explicitly. A clear orchestration layer reduced complexity and improved debuggability.

User Interface

The frontend was built as a modern single page application.

- Technology Stack. React with TypeScript was used for type safety and maintainability. Tailwind was used for consistent styling and rapid iteration.
- Behavior. The UI supported streaming responses, citation inspection, and fast follow up queries.
- Benefit. Strong typing reduced runtime errors, and streaming improved perceived responsiveness during generation.

API Gateway

All client traffic was routed through a centralized gateway.

- Implementation. AWS API Gateway handled request routing, rate limiting, authentication, and monitoring.
- Security. JWT based authentication enforced user and tenant identity before any backend processing occurred.
- Benefit. This centralized cross cutting concerns and reduced duplication across services.

Orchestration Layer

LangChain was used to orchestrate the RAG workflow.

- Responsibilities. Query preprocessing, retrieval coordination, reranking, prompt assembly, and model invocation.
- Customization. Default LangChain components were extended with custom logic for multi query generation, fallback handling, and tenant isolation.
- Benefit. This provided structure without locking the system into rigid abstractions.

Backend Services

The backend was implemented as a set of microservices.

- Framework. FastAPI was used for its performance, async support, and clear API contracts.
- Asynchronous Processing. Celery handled background tasks such as ingestion, re indexing, summarization, and evaluation jobs.
- Benefit. This separated latency sensitive request handling from long running or compute heavy tasks.

LLM Integration

All model access was routed through a managed service.

- Platform. AWS Bedrock was used to access Claude Sonnet and Claude Haiku for generation, along with Titan where appropriate.
- Routing Logic. Model selection was driven by latency sensitivity, prompt complexity, and confidence signals.
- Benefit. This provided flexibility across models while maintaining consistent security and governance controls.

Challenges Faced

End to End Latency Coordination. Optimizing individual components was insufficient. Careful profiling was required to control cumulative latency across UI, gateway, orchestration, and model execution.

Async Boundary Management. Coordinating synchronous user requests with asynchronous backend tasks introduced complexity in error handling and state consistency.

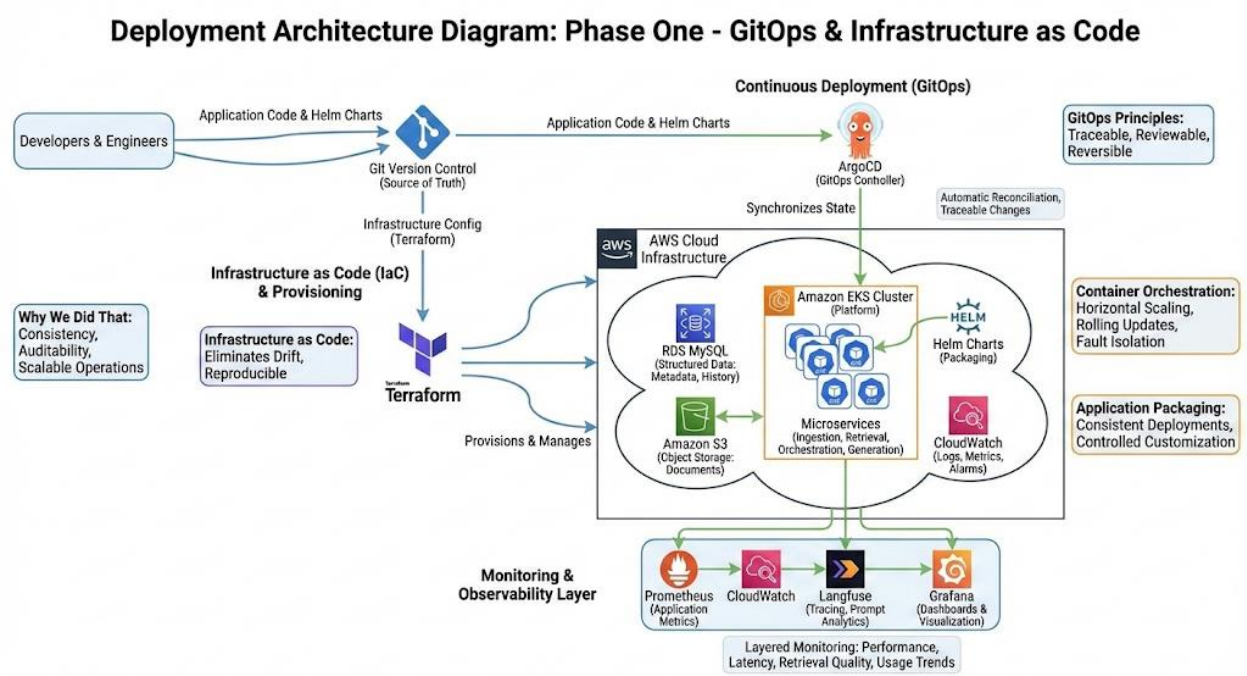
Framework Abstraction Leakage. LangChain abstractions occasionally obscured performance bottlenecks. Custom instrumentation was required to maintain visibility.

Deployment Consistency. Ensuring compatible versions across frontend, backend services, and orchestration logic required disciplined release management.

Deployment

In Phase One, we implemented a production grade deployment strategy focused on repeatability, controlled change management, and operational visibility. Given the compliance and reliability requirements of the system, manual deployments and environment specific configuration drift were not acceptable.

The deployment model followed infrastructure as code and GitOps principles, ensuring that all changes were traceable, reviewable, and reversible.



Why We Did That

Consistency Across Environments. Development, staging, and production environments needed to behave identically to avoid surprises during promotion.

Auditability and Rollback. In regulated systems, every change must be attributable to a commit and reversible without emergency intervention.

Scalable Operations. The platform had to support ongoing model, prompt, and retrieval updates without service interruption.

Container Orchestration

- We deployed backend services on Kubernetes.
- Platform. Amazon EKS was used to run all microservices, including ingestion, retrieval, orchestration, and generation services.
- Benefit. EKS provided horizontal scaling, rolling updates, and fault isolation across services.

Application Packaging

- Helm charts were used to package and deploy services.
- Usage. Helm templates defined service configuration, resource limits, secrets references, and environment specific overrides.
- Benefit. This enabled consistent deployments while allowing controlled customization per environment.

Continuous Deployment

- We adopted a GitOps deployment model.
- Tooling. ArgoCD continuously synchronized the Kubernetes cluster state with version controlled manifests.
- Behavior. Any approved change in Git was automatically reconciled into the cluster.
- Benefit. This reduced manual intervention and ensured cluster state always reflected declared configuration.

Infrastructure as Code

- All supporting infrastructure was provisioned declaratively.
- Tooling. Terraform was used to manage cloud resources such as EKS clusters, networking, IAM roles, databases, and storage.
- Benefit. This eliminated environment drift and made infrastructure changes auditable and reproducible.

Relational Storage

- RDS MySQL was used for structured data storage.
- Usage. Metadata tracking and conversation history were stored in MySQL with strict access controls and backups enabled.

Object Storage

- Amazon S3 served as the system of record for documents.
- Usage. Raw, processed, and versioned documents were stored with lifecycle policies and tenant level access controls.

Monitoring and Observability

- We implemented layered monitoring across infrastructure and application layers.
- CloudWatch provided system logs, basic metrics, and alarms.
- Prometheus collected application and custom metrics.
- Grafana dashboards visualized performance, latency, and retrieval quality trends.
- Langfuse provided request level tracing, prompt inspection, and model usage analytics.

Challenges Faced

Monitoring Noise. Early alert configurations were too sensitive, generating noise. Thresholds and alerting rules had to be tuned to surface actionable signals only.