Case Study: Rebuilding the Context Pipeline for an AI Legal Research Assistant

Section 1: Industry Context and Business Problem

Industry: Legal Technology

The legal industry generates an extraordinary volume of text. In the United States alone, there are over 6.7 million published court opinions, hundreds of thousands of statutes and regulations across federal and state jurisdictions, and millions of legal memoranda, contracts, and filings produced every year. An attorney preparing a case brief must navigate this ocean of text to find the handful of precedents, statutes, and arguments that are directly relevant to their client's situation.

This is, at its core, an information retrieval and synthesis problem — and it is one where LLMs have shown enormous promise. But the gap between a demo and a production-grade legal research assistant is vast, and that gap is almost entirely about context engineering.

Company Profile: BriefEngine

BriefEngine is a Series B legal technology startup headquartered in Boston, Massachusetts. Founded in 2022 by two former litigators and a machine learning engineer from a major search company, BriefEngine builds an AI-powered legal research assistant designed for mid-size litigation firms (50–200 attorneys).

- **Founded**: 2022
- **Employees**: ~85 (35 engineers, 12 legal domain experts, remainder in sales, ops, and support)
- **Funding**: 38M USD raised across Seed, Series A, and Series B rounds. Most recent round led by a top-tier enterprise SaaS fund at a 180M USD valuation
- **Product**: An AI assistant that helps litigation attorneys prepare case briefs by retrieving relevant case law, synthesizing legal arguments, and generating properly cited draft briefs
- **Customers**: 47 mid-size law firms across 14 U.S. states, representing approximately 3,200 active attorney users
- **Revenue**: 6.2M USD ARR, growing 18% quarter-over-quarter
- **Key Product Lines**:
- **BriefEngine Research** — AI-powered case law search and retrieval
- **BriefEngine Draft** — automated first-draft generation for case briefs, motions, and memoranda

- **BriefEngine Cite** — citation verification and Shepardization (checking if cited cases are still good law)

Business Challenge

BriefEngine's product works — but not well enough. The original system was built in early 2023 using a straightforward RAG pipeline: chunk case law into passages, embed with a general-purpose embedding model, retrieve top-K results, and pass them to an LLM for synthesis. This approach was sufficient to win early customers and raise a Series B.

But as usage scaled and attorneys began relying on the tool for high-stakes litigation, four critical failure patterns emerged:

1. **Bad citations propagate through briefs.** The AI occasionally retrieves and cites cases that have been overruled, superseded, or distinguished by later courts. When this happens, the erroneous citation does not just appear once — it enters the context as a "fact" and the model references it in subsequent paragraphs, building arguments on a foundation that does not exist. In one incident, a BriefEngine-generated brief cited *Henderson v. Pacific Mutual* (1987) three times as controlling authority — but that case had been explicitly overruled in 2019. The attorney caught it during review, but only after spending two hours tracing the error.

2. **Critical precedents get buried in long retrievals.** When the retrieval pipeline returns 40–60 document chunks to fill the context window, the most important cases sometimes land in the middle of the context. Due to the "Lost in the Middle" effect, the LLM attends weakly to these mid-context documents and instead over-weights cases that happen to appear at the beginning or end of the retrieval results — regardless of their actual relevance.

3. **Irrelevant jurisdictions pollute the analysis.** A California state court motion should cite California case law and relevant federal precedent. But the retrieval system, which uses semantic similarity over a national corpus, frequently pulls in cases from New York, Texas, or other jurisdictions that are semantically similar but legally irrelevant. Attorneys report spending significant time deleting inapplicable citations.

4. **Fragmented instructions cause contradictory arguments.** Attorneys provide strategy notes, jurisdictional preferences, and case-specific constraints through multiple interface elements (a strategy field, a notes panel, jurisdiction dropdowns). These inputs are assembled into the prompt through separate template sections. The model sometimes receives contradictory signals — for example, the strategy says "argue for strict liability" while a retrieved case summary discusses negligence standards — and produces briefs that argue both sides.

Why It Matters

The consequences are both financial and reputational:

- **Attorney time waste**: Attorneys currently spend 3–4 hours reviewing and correcting each AI-generated brief. The product's value proposition is reducing this to under 30 minutes. At an average billing rate of 450 USD/hour, each brief review costs the firm 1,350–1,800 USD in unbilled attorney time.
- **Customer churn risk**: Two of BriefEngine's largest customers (representing 840K USD in combined ARR) have escalated complaints to the executive team and set a 90-day deadline for improvement before evaluating competitors.
- **Malpractice exposure**: Attorneys have a professional obligation to verify the accuracy of their filings. An AI tool that introduces bad citations creates malpractice risk — and if a filing based on BriefEngine output results in sanctions, the reputational damage to the company could be existential.
- **Competitive pressure**: Three well-funded competitors (two backed by major legal publishers) are entering the market with context-aware legal AI products. BriefEngine's 18-month head start is eroding.

Constraints

The engineering team must operate within these real-world constraints:

- **Compute budget**: 45K USD/month for cloud GPU inference (currently running Anthropic Claude API and self-hosted embedding models on AWS)
- **Latency**: Attorneys expect a draft brief within 90 seconds. The current pipeline averages 120 seconds, and any architectural change must not increase this
- **Data compliance**: Law firm data is governed by attorney-client privilege. All client case files must remain within BriefEngine's SOC 2-certified infrastructure. No client data may be sent to third-party APIs without explicit per-document consent
- **Corpus size**: The case law corpus contains 6.7 million documents (CaseLaw Access Project), plus approximately 200K firm-specific documents across all customers. The vector index currently holds 42 million chunks
- **Team**: 8 ML engineers, 3 infrastructure engineers, 2 legal domain experts embedded in the ML team. No dedicated ML research team — the engineers must implement proven techniques, not invent new ones
- **Model access**: BriefEngine uses Claude 3.5 Sonnet via API for generation (128K context window) and a fine-tuned E5-large model for embeddings

---

Section 2: Technical Problem Formulation

Problem Type: Retrieval-Augmented Generation with Structured Context Assembly

This is fundamentally a **conditional text generation** problem with a complex retrieval front-end. The model must generate a legal brief that is (a) responsive to the attorney's specific case and strategy, (b) grounded in relevant and valid legal authority, and (c) properly structured with accurate citations.

Why not a simpler framing?

- **Pure retrieval** (just finding relevant cases) is insufficient — attorneys need synthesis, not search results. They need the AI to construct arguments.
- **Pure generation** (just prompting the LLM) fails because the model's training data is frozen and does not contain recent case law, firm-specific documents, or case-specific facts. It will hallucinate citations.
- **Classification or extraction** does not capture the generative, argumentative nature of brief writing.

RAG is the right framing because it combines the LLM's ability to reason and write with an external knowledge base's ability to provide current, specific, and verifiable legal authority. But the quality of the RAG system depends entirely on how context is assembled — which is exactly what context engineering addresses.

Input Specification

The system receives five categories of input:

1. **Case facts** (free text, typically 500–2,000 tokens): A description of the client's situation, key events, and relevant dates. This is provided by the attorney.
2. **Legal question** (free text, 50–200 tokens): The specific legal question to be addressed in the brief (e.g., "Whether the defendant's conduct constitutes intentional infliction of emotional distress under California law").
3. **Strategy notes** (structured + free text, 100–500 tokens): The attorney's preferred argumentative approach, including which legal theories to prioritize, which arguments to avoid, and the desired tone.
4. **Jurisdictional constraints** (structured metadata): The relevant jurisdiction (state, federal circuit, or both), court level, and any specific courts whose precedent should be weighted.
5. **Case law corpus** (external knowledge base): 6.7 million court opinions indexed as 42 million vector embeddings, plus firm-specific document collections.

Each input carries distinct information that the model needs: - Case facts provide the **ground truth** that arguments must be consistent with - The legal question defines the **scope** of the brief - Strategy notes encode the attorney's **intent** — critical for avoiding context clash - Jurisdictional constraints act as **hard filters** on retrieval — a California brief must not cite Alabama law as binding authority - The corpus provides the **evidence base** from which citations are drawn

Output Specification

The model produces a **structured legal brief draft** consisting of:

- **Header**: Case caption, court, and filing metadata
- **Statement of facts**: A narrative derived from the attorney's case facts, framed to support the chosen strategy
- **Legal argument sections** (typically 3–5): Each section presents a legal argument supported by cited authority. Each citation must include the case name, reporter citation, year, and a parenthetical explanation
- **Conclusion**: Summary of arguments and requested relief

The output is constrained by: - **Citation validity**: Every cited case must exist in the corpus and must not have been overruled or superseded - **Jurisdictional relevance**: Binding authority must come from the specified jurisdiction. Persuasive authority from other jurisdictions must be explicitly identified as such - **Internal consistency**: Arguments must not contradict each other or the attorney's stated strategy - **Length**: Typically 3,000–6,000 tokens (5–15 pages when formatted)

Why this output representation? Legal briefs have a rigid, well-defined structure that attorneys expect. Deviating from this structure (e.g., generating a free-form essay) would require more attorney editing time, not less. The structured output also enables automated post-processing: citation verification, jurisdiction checking, and formatting.

Mathematical Foundation

The core mathematical machinery underlying this system combines vector similarity search (for retrieval) with conditional language modeling (for generation). Let us build up from first principles.

Embedding and Similarity

The retrieval system converts text into dense vector representations. An embedding model f_θ maps a text string x to a vector $\mathbf{v} \in \mathbb{R}^d$:

$$f_\theta(x) = \mathbf{v} \in \mathbb{R}^d$$

The key property we want: texts that are semantically similar should have vectors that are close together. We measure closeness using **cosine similarity**:

$$\text{sim}(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{||\mathbf{q}|| \; ||\mathbf{d}||}$$

This metric ranges from -1 (opposite meaning) to 1 (identical meaning), and it is invariant to vector magnitude — only the direction matters. This is important because we want to measure semantic alignment, not text length.

For a query embedding $\mathbf{q}$ and a corpus of N document embeddings $\{\mathbf{d}_1, \ldots, \mathbf{d}_N\}$, top-K retrieval selects:

$$\mathcal{R}_K(\mathbf{q}) = \underset{S \subseteq \{1,\ldots,N\}, |S|=K}{\arg\max} \sum_{i \in S} \text{sim}(\mathbf{q}, \mathbf{d}_i)$$

In practice, exact search over 42 million vectors is too slow. BriefEngine uses **approximate nearest neighbor (ANN)** search via HNSW (Hierarchical Navigable Small World) graphs, which trade a small amount of recall for orders-of-magnitude speedup. The recall@K metric measures how often the true top-K results appear in the approximate results.

Reranking

Initial retrieval using cosine similarity is fast but coarse. A **cross-encoder reranker** provides a more accurate relevance score by attending jointly to the query and document:

$$s_{\text{rerank}}(q, d) = \sigma(g_\phi([q; d]))$$

where $[q; d]$ is the concatenation of query and document tokens, g_ϕ is a transformer cross-encoder, and σ is the sigmoid function. Unlike bi-encoder similarity (which encodes query and document independently), the cross-encoder can model fine-grained interactions between query and document tokens. This is more expensive (it must be run per-document), which is why we first narrow the candidates with the fast bi-encoder.

Token Budget Optimization

The context window has a fixed capacity $T_{\max}$. The total token usage must satisfy:

$$T_{\text{system}} + T_{\text{history}} + T_{\text{RAG}} + T_{\text{user}} + T_{\text{output}} \leq T_{\max}$$

This is a **constrained allocation problem**. We want to maximize the information value of the context while staying within budget. The key insight from the article is that this allocation must be dynamic — different queries need different distributions of context.

For BriefEngine specifically, we can formalize the allocation as:

$$\max_{\mathbf{t}} \; V(\mathbf{t}) \quad \text{subject to} \quad \sum_i t_i \leq T_{\max}, \; t_i \geq t_i^{\min} \; \forall i$$

where $\mathbf{t} = (t_{\text{system}}, t_{\text{facts}}, t_{\text{strategy}}, t_{\text{RAG}}, t_{\text{output}})$ is the token allocation vector, $V(\mathbf{t})$ is the information value function (approximated empirically by brief quality scores), and $t_i^{\min}$ are minimum allocations for each component.

Context Quality Metric

We can formalize the quality of a context assembly as the balance between **relevance** and **coherence**:

$$Q(\mathcal{C}) = \alpha \cdot \text{Relevance}(\mathcal{C}, q) + (1 - \alpha) \cdot \text{Coherence}(\mathcal{C})$$

where: - $\text{Relevance}(\mathcal{C}, q) = \frac{1}{|\mathcal{C}|} \sum_{d \in \mathcal{C}} \text{sim}(\mathbf{q}, \mathbf{d})$ measures how relevant the retrieved documents are to the query - $\text{Coherence}(\mathcal{C}) = 1 - \frac{1}{|\mathcal{C}|^2} \sum_{d_i, d_j \in \mathcal{C}} \mathbb{1}[\text{contradicts}(d_i, d_j)]$ penalizes contradictory documents in context - α is a tradeoff parameter (empirically, $\alpha = 0.7$ works well for legal applications)

This metric directly connects to the failure modes: low relevance corresponds to context confusion, low coherence corresponds to context clash, and optimizing both simultaneously is the goal of context engineering.

Loss Function

The generation model is a pre-trained LLM (Claude 3.5 Sonnet) used via API, so we do not train the generation model directly. However, several components are trained or fine-tuned:

1. Embedding Model Fine-tuning Loss

The E5-large embedding model is fine-tuned on legal text pairs using **InfoNCE (contrastive) loss**:

$$\mathcal{L}_{\text{embed}} = -\log \frac{\exp(\text{sim}(\mathbf{q}, \mathbf{d}^+)/\tau)}{\exp(\text{sim}(\mathbf{q}, \mathbf{d}^+)/\tau) + \sum_{j=1}^{N^-} \exp(\text{sim}(\mathbf{q}, \mathbf{d}_j^-)/\tau)}$$

where $\mathbf{d}^+$ is a relevant document, $\mathbf{d}_j^-$ are irrelevant documents, and τ is a temperature parameter.

- **What it optimizes**: Pulling relevant legal passages closer to the query in embedding space while pushing irrelevant ones apart
- **What happens if removed**: The embedding model treats all text generically, failing to distinguish legal relevance (e.g., a case about "battery" the tort vs. "battery" the device)
- **Temperature τ effect**: Lower τ sharpens the distribution, making the model more discriminative but harder to train. Typical range: 0.05–0.1

2. Reranker Loss

The cross-encoder reranker is trained with **binary cross-entropy**:

$$\mathcal{L}_{\text{rerank}} = -[y \log s(q, d) + (1 - y) \log(1 - s(q, d))]$$

where $y \in \{0, 1\}$ is the relevance label and $s(q, d)$ is the predicted relevance score.

- **What it optimizes**: Accurately predicting whether a document is relevant to the query after seeing both together

- **What happens if removed**: The system relies solely on bi-encoder similarity, which misses nuanced relevance signals (e.g., a case that discusses the same legal principle but in a different factual context)

3. Context Quality Loss (End-to-End)

To optimize the full pipeline, BriefEngine uses a **reward-based objective** derived from attorney feedback:

$$\mathcal{L}_{\text{context}} = -\mathbb{E}_{q \sim \mathcal{Q}} \left[R(q, \mathcal{C}(q)) \right]$$

where $R(q, \mathcal{C}(q))$ is a reward signal based on attorney ratings of brief quality (1–5 scale), and $\mathcal{C}(q)$ is the assembled context for query q. This loss is optimized over the retrieval and context assembly parameters (chunk size, top-K, reranking threshold, token budget allocation) using Bayesian optimization.

- **What it optimizes**: The overall quality of the context assembly pipeline as judged by end users
- **What happens if removed**: Each component is optimized independently, potentially leading to locally optimal but globally suboptimal context assemblies

Evaluation Metrics

Primary Metrics:

Metric	Definition	Target
Citation Accuracy	% of cited cases that are valid (exist, not overruled, correct jurisdiction)	> 95%
Relevance@K	% of top-K retrieved documents rated relevant by legal experts	> 80% at K=10
Brief Quality Score	Attorney rating on 1–5 scale (argument quality, citation quality, structure)	> 4.0 average

Secondary Metrics:

Metric	Definition	Target
End-to-end Latency	Time from query submission to complete brief generation	< 90 seconds
Context Utilization	% of context window tokens that are relevant to the query	> 70%
Jurisdictional Precision	% of cited cases from the correct jurisdiction	> 90%
Review Time	Attorney time spent reviewing and correcting the brief	< 30 minutes

Baseline: Keyword Search + Template Brief

Without context engineering, the simplest approach is: 1. BM25 keyword search over the case law corpus (no embeddings, no semantic understanding) 2. Return top-20 results ranked by

term frequency 3. Concatenate all 20 documents into the context window 4. Use a generic prompt: "Write a legal brief based on the following cases"

This baseline achieves: - Citation Accuracy: ~62% (many retrieved cases are topically related but legally irrelevant) - Relevance@10: ~45% (keyword matching misses semantically relevant cases with different terminology) - Brief Quality Score: 2.8/5.0 (arguments are generic, citations often inapplicable) - Review Time: 3.5 hours average

The baseline fails for three fundamental reasons: (1) keyword search cannot capture semantic legal relevance, (2) dumping all results into context without prioritization triggers the distraction failure mode, and (3) the generic prompt provides no strategic direction, triggering the confusion failure mode.

Why Context Engineering

Context engineering is the right approach for this problem because the core failures are all context failures, not model capability failures:

1. **Citation poisoning** (Failure Mode 1) is solved by the **Write** strategy — maintain a persistent, verified citation validity index outside the context window, and check every retrieved case against it before inclusion
2. **Lost-in-the-middle precedents** (Failure Mode 2) is solved by the **Compress** strategy — rerank aggressively and keep only the highest-quality citations, placing them at the beginning and end of the context
3. **Jurisdictional confusion** (Failure Mode 3) is solved by the **Select** strategy — apply hard jurisdictional filters before retrieval, so irrelevant jurisdictions never enter the context
4. **Instruction fragmentation** (Failure Mode 4) is solved by the **Isolate** strategy — use a unified context assembly step that merges all attorney inputs into a single coherent instruction block, and delegate deep research to sub-agents with isolated contexts

The fundamental property of context engineering that makes it right for this problem: the LLM is already capable of writing excellent legal briefs given the right information. The bottleneck is not model intelligence — it is information assembly. This matches the article's core thesis: the model is the student, the context is the open book, and our job is to design that book.

Technical Constraints

- **Model**: Claude 3.5 Sonnet (128K context window), accessed via API
- **Embedding model**: Fine-tuned E5-large (1024-dimensional embeddings, ~335M parameters)
- **Reranker**: Fine-tuned cross-encoder based on DeBERTa-v3-large (~435M parameters)
- **Vector index**: 42 million chunks in HNSW index (Weaviate), average chunk size 512 tokens
- **Inference latency budget**: < 90 seconds end-to-end (retrieval < 2s, reranking < 3s, generation < 85s)

- **Training compute**: 4x A100 GPUs for embedding and reranker fine-tuning (not for LLM training)
- **Context token budget**: 128K total, 35K reserved for output, 93K available for context assembly

---

Section 3: Implementation Notebook Structure

This section defines the structure of the Google Colab notebook that students will implement. The notebook builds a complete context engineering pipeline for the legal research use case, progressing from data exploration through model evaluation.

3.1 Data Acquisition Strategy

Dataset: CaseLaw Access Project (Harvard Law School)

The CaseLaw Access Project (CAP) provides free, bulk access to over 6.7 million published U.S. court decisions. For this case study, we will work with a curated subset:

- **California state courts** (Supreme Court and Courts of Appeal): ~180,000 opinions
- **Ninth Circuit federal courts**: ~95,000 opinions
- **Date range**: 2000–2024
- **Format**: JSON with full text, metadata (court, date, citation, jurisdiction), and citation graph

We use this subset because (a) it is large enough to be realistic but small enough to fit on Colab, (b) California litigation is one of BriefEngine's largest markets, and (c) the Ninth Circuit provides corresponding federal precedent.

For embedding and retrieval experiments, we will use a further-sampled working set of 10,000 opinions (chunked into ~50,000 passages) that can be embedded and indexed within Colab's memory and compute constraints.

Data loading pipeline: 1. Download pre-processed CAP subset from a public GCS bucket (provided) 2. Parse JSON into structured records (opinion text, metadata, citation links) 3. Chunk opinions into 512-token passages with 64-token overlap 4. Build metadata index for jurisdiction filtering

TODO (Student): - Implement the chunking function with configurable chunk size and overlap - Implement a metadata extraction function that parses court name, date, and jurisdiction from the raw records - Validate that all chunks retain their parent document's metadata

3.2 Exploratory Data Analysis

Students will explore the dataset to understand its structure and identify potential challenges for retrieval:

- **Distribution of opinion lengths** (tokens per opinion) — are there outliers that will create problematic chunks?
- **Temporal distribution** — how are opinions distributed across years? Are there gaps?
- **Jurisdictional breakdown** — what proportion of opinions come from each court level?
- **Citation network statistics** — how densely connected are the citations? What is the average number of citations per opinion?
- **Vocabulary analysis** — what are the most frequent legal terms? How does vocabulary differ between jurisdictions?

TODO (Student): - Plot the distribution of opinion lengths and identify the 95th percentile length - Create a bar chart showing opinions per court per decade - Compute and visualize the in-degree distribution of the citation graph (how many times is each case cited?) - Identify the top 20 most-cited cases in the corpus — these are likely landmark precedents - Answer: What chunk size would capture 90% of legal arguments without exceeding 1,000 tokens?

3.3 Baseline Approach

Implement a BM25 keyword search baseline to establish a performance floor.

Implementation: - Use the `rank_bm25` library to build a BM25 index over the chunked passages - Implement a query function that returns top-K passages by BM25 score - Assemble a naive context by concatenating all retrieved passages - Send the context + a generic prompt to the LLM and evaluate the output

TODO (Student): - Implement the BM25 indexing and retrieval pipeline - Write a function that assembles a context window from BM25 results (simple concatenation) - Evaluate the baseline on 10 sample legal queries (provided) using the Relevance@K metric - Measure the context utilization ratio: what percentage of retrieved tokens are actually relevant? - Answer: Why does BM25 struggle with legal text? Provide two specific examples from your results.

3.4 Model Design: Context Engineering Pipeline

This is the core implementation section. Students will build a context engineering pipeline that addresses all four failure modes.

Architecture Overview:

The pipeline consists of five stages:

1. **Query Analysis** — Parse the attorney's input into structured components (legal question, jurisdictional constraints, strategy)

2. **Filtered Retrieval** — Apply hard metadata filters (jurisdiction, date range, case validity) before semantic search
3. **Semantic Search + Reranking** — Embed the query, retrieve top-K candidates via cosine similarity, then rerank with a cross-encoder
4. **Context Assembly** — Allocate the token budget across components, compress history, and assemble the final context with structured XML tags
5. **Generation + Citation Verification** — Generate the brief and post-process to verify all citations

Each stage corresponds to one or more context engineering strategies from the article.

Stage 1: Query Analysis

This stage implements the **Select** strategy by parsing the query into components that will drive filtered retrieval.

TODO (Student): - Implement `parse_legal_query(query_text, jurisdiction, strategy_notes)` that returns a structured `LegalQuery` dataclass with fields: `legal_question`, `jurisdiction_filter`, `date_range`, `strategy_keywords`, `excluded_topics` - The function should extract the core legal question, identify jurisdictional constraints, and parse strategy notes into actionable filters - Include input validation: the function should raise `ValueError` if the jurisdiction is not in the supported list

```
from dataclasses import dataclass
from typing import List, Optional, Tuple

@dataclass
class LegalQuery:
    legal_question: str
    jurisdiction_filter: List[str]
    date_range: Tuple[int, int]
    strategy_keywords: List[str]
    excluded_topics: List[str]

def parse_legal_query(
    query_text: str,
    jurisdiction: str,
    strategy_notes: str = "",
    date_range: Optional[Tuple[int, int]] = None
) -> LegalQuery:
    """Parse raw attorney input into a structured legal query.

    Args:
        query_text: The legal question or research request
        jurisdiction: Target jurisdiction (e.g., "CA", "9th-circuit", "federal")
        strategy_notes: Attorney's strategy preferences (optional)
        date_range: Tuple of (start_year, end_year) or None for all years

    Returns:
        LegalQuery with parsed and validated fields

    Hints:
        1. Define a mapping of jurisdiction codes to court name patterns
        2. Extract strategy keywords by splitting on commas and stripping whitespace
        3. Set default date_range to (2000, 2024) if not provided
        4. Validate jurisdiction against the supported list
    """
    # TODO: Implement this function
    pass
```

Stage 2: Filtered Retrieval

This stage applies hard filters to eliminate irrelevant documents before they can enter the context window, directly preventing the **context confusion** failure mode.

TODO (Student): - Implement `filter_corpus(chunks, metadata, query: LegalQuery)` that returns only chunks matching the jurisdiction, date range, and validity constraints - The function must handle hierarchical jurisdiction logic: if the query targets California state courts, include both California Supreme Court and California Courts of Appeal, plus relevant Ninth Circuit federal cases - Measure the reduction ratio: what percentage of the corpus is eliminated by filtering?

```
def filter_corpus(
    chunks: List[str],
    metadata: List[dict],
    query: LegalQuery
) -> Tuple[List[str], List[dict]]:
    """Filter corpus chunks by jurisdiction, date, and validity.

    Args:
        chunks: List of text chunks
        metadata: List of metadata dicts with keys: 'court', 'date', 'jurisdiction',
                  'is_overruled', 'citation'
        query: Parsed LegalQuery with filter criteria

    Returns:
        Tuple of (filtered_chunks, filtered_metadata)

    Hints:
        1. Build a set of acceptable court patterns from query.jurisdiction_filter
        2. Filter by date_range using the 'date' field (year extraction)
        3. CRITICAL: Exclude any chunk where is_overruled is True
           (this prevents context poisoning)
        4. Return both the filtered chunks and their corresponding metadata
    """
    # TODO: Implement this function
    pass
```

Stage 3: Semantic Search + Reranking

This stage implements the core **Select** strategy — retrieving the most relevant documents using embedding similarity, then refining with a cross-encoder reranker.

TODO (Student): - Implement `embed_and_retrieve(query_text, filtered_chunks, embedding_model, top_k=20)` using a sentence-transformers model - Implement `rerank(query_text, candidates, reranker_model, top_n=10)` using a cross-encoder - Compare retrieval quality with and without reranking on the sample queries - Experiment with different values of top_k (10, 20, 50) and top_n (5, 10, 15) and report the Relevance@K tradeoff

```
def embed_and_retrieve(
    query_text: str,
    chunks: List[str],
    model,  # SentenceTransformer model
    top_k: int = 20
) -> List[Tuple[str, float]]:
    """Retrieve top-K chunks by cosine similarity.

    Args:
        query_text: The legal query text
        chunks: Pre-filtered corpus chunks
        model: A SentenceTransformer model for encoding
        top_k: Number of candidates to retrieve
```

```
    Returns:
        List of (chunk_text, similarity_score) tuples, sorted by score descending

    Hints:
        1. Encode the query and all chunks using model.encode()
        2. Normalize embeddings to unit length
        3. Compute cosine similarity via dot product (since normalized)
        4. Use np.argsort to get top-K indices
        5. Return chunks with their scores
    """
    # TODO: Implement this function
    pass


def rerank(
    query_text: str,
    candidates: List[Tuple[str, float]],
    reranker,  # CrossEncoder model
    top_n: int = 10
) -> List[Tuple[str, float]]:
    """Rerank candidates using a cross-encoder for fine-grained relevance.

    Args:
        query_text: The legal query text
        candidates: List of (chunk_text, initial_score) from embed_and_retrieve
        reranker: A CrossEncoder model
        top_n: Number of final results to return

    Returns:
        List of (chunk_text, reranker_score) tuples, sorted by reranker score descending

    Hints:
        1. Create input pairs: [(query_text, chunk_text) for each candidate]
        2. Score all pairs using reranker.predict()
        3. Sort by reranker score and return top_n
        4. Note: reranker scores are NOT cosine similarities -- they are
           logits that can be any real number
    """
    # TODO: Implement this function
    pass
```

Stage 4: Context Assembly

This is where all four strategies converge. Students will implement a `ContextEngine` that allocates token budgets, compresses history, and assembles structured context.

TODO (Student): - Implement the `ContextEngine` class with methods for token estimation, budget allocation, history compression, and final assembly - The context must use XML-style tags to clearly delineate each section (as shown in the article's code examples) - Implement dynamic budget allocation: if there are few retrieved documents, reallocate unused RAG budget to conversation history - Test that the assembled context never exceeds the 93K available token budget

```
class ContextEngine:
    """Assembles optimal context for legal brief generation.

    Implements all four context engineering strategies:
    - Write: Loads persisted case validity index and attorney preferences
    - Select: Uses filtered retrieval and reranking results
    - Compress: Truncates history to fit token budget, keeping most recent
    - Isolate: Structures context with clear XML boundaries
    """

    def __init__(self, max_tokens: int = 128000, reserved_output: int = 35000):
        self.max_tokens = max_tokens
        self.reserved = reserved_output
        self.available = max_tokens - reserved_output
```

```
    def estimate_tokens(self, text: str) -> int:
        """Estimate token count. ~4 chars per token for English legal text.

        Hints:
            1. Simple heuristic: len(text) // 4
            2. For more accuracy, use tiktoken if available
            3. Legal text tends to be slightly more token-dense due to
               specialized vocabulary -- consider a factor of 3.8
        """
        # TODO: Implement this method
        pass

    def allocate_budget(
        self,
        system_prompt: str,
        case_facts: str,
        strategy_notes: str,
        num_retrieved_docs: int,
        avg_doc_length: int
    ) -> dict:
        """Dynamically allocate token budget across context components.

        Returns a dict with keys: 'system', 'facts', 'strategy', 'rag',
        'history', 'buffer' and integer token values for each.

        Hints:
            1. System prompt, facts, and strategy are fixed-size allocations
            2. RAG budget = min(num_retrieved_docs * avg_doc_length, 60% of remaining)
            3. History gets 25% of remaining after RAG
            4. Buffer of 5% for safety margin
            5. If RAG allocation is under-utilized, redistribute to history
        """
        # TODO: Implement this method
        pass

    def compress_history(self, history: List[str], budget: int) -> List[str]:
        """Keep most recent history messages within token budget.

        Hints:
            1. Iterate from most recent to oldest
            2. Accumulate token counts
            3. Stop when budget is exceeded
            4. Return messages in chronological order (reverse back)
        """
        # TODO: Implement this method
        pass

    def assemble(
        self,
        system_prompt: str,
        case_facts: str,
        strategy_notes: str,
        retrieved_docs: List[Tuple[str, float]],
        history: List[str],
        memory: List[str]
    ) -> str:
        """Assemble the complete context with XML-tagged structure.

        The output should look like:
        <system>...</system>
        <memory>...</memory>
        <case_facts>...</case_facts>
        <strategy>...</strategy>
        <history>...</history>
        <retrieved_authority>
          [0.94] Case text here...
          [0.91] Another case...
        </retrieved_authority>

        Hints:
            1. Call allocate_budget to get token limits per section
            2. Compress history to fit its budget
            3. Truncate retrieved docs to fit RAG budget (in order of relevance)
            4. Include relevance scores in retrieved docs for transparency
```

```
            5. Place highest-relevance docs at the BEGINNING and END of the
               retrieved section (mitigates "Lost in the Middle")
            6. Verify total tokens <= self.available before returning
        """
        # TODO: Implement this method
        pass
```

Stage 5: Citation Verification

This stage implements the **Write** strategy — maintaining a persistent validity index and checking every citation in the generated output.

TODO (Student): - Implement `build_validity_index(metadata)` that creates a lookup table mapping case citations to their validity status (valid, overruled, superseded, distinguished) - Implement `verify_citations(generated_text, validity_index)` that extracts all case citations from the generated brief and checks each against the index - Return a report listing each citation, its validity status, and whether it should be kept or flagged

```
def build_validity_index(metadata: List[dict]) -> dict:
    """Build a citation validity lookup from corpus metadata.

    Args:
        metadata: List of dicts with keys including 'citation',
                  'is_overruled', 'overruled_by', 'date'

    Returns:
        Dict mapping citation string -> {'status': str, 'overruled_by': str or None, 'date': int}

    Hints:
        1. Iterate through all metadata records
        2. For each citation, store its validity status
        3. If overruled, store the overruling case citation and date
        4. Handle duplicate citations (same case in multiple chunks)
           by keeping the most recent metadata
    """
    # TODO: Implement this function
    pass


def verify_citations(
    generated_text: str,
    validity_index: dict
) -> List[dict]:
    """Extract and verify all case citations in generated text.

    Args:
        generated_text: The LLM-generated legal brief text
        validity_index: The citation validity lookup

    Returns:
        List of dicts with keys: 'citation', 'status', 'found_in_index',
        'recommendation' ('keep', 'flag', 'remove')

    Hints:
        1. Use regex to extract case citations (pattern:
           "Case Name, Reporter Vol. Page (Year)")
        2. Look up each citation in the validity index
        3. If not found, flag as 'unverified'
        4. If found and overruled, mark as 'remove'
        5. If found and valid, mark as 'keep'
    """
    # TODO: Implement this function
    pass
```

3.5 Training Strategy

In this pipeline, we do not train the generation LLM. Instead, we fine-tune two critical components: the embedding model and the reranker.

Embedding Model Fine-tuning:

- **Model**: `intfloat/e5-large-v2` (335M parameters)
- **Optimizer**: AdamW with weight decay 0.01
- **Learning rate**: 2e-5 with linear warmup (10% of steps) followed by cosine decay
- **Batch size**: 32 (with in-batch negatives for contrastive learning)
- **Loss**: InfoNCE with temperature $\tau = 0.07$
- **Training data**: Legal query-passage pairs derived from the citation graph (if case A cites passage from case B in its reasoning, that is a positive pair)
- **Epochs**: 3 (legal domain adaptation typically converges quickly from a strong general-purpose checkpoint)

Why AdamW over SGD? Fine-tuning a pre-trained model requires careful, adaptive updates. SGD with momentum can work but requires much more learning rate tuning. AdamW's per-parameter adaptive learning rates handle the heterogeneous gradient magnitudes across transformer layers. The weight decay prevents the fine-tuned model from drifting too far from the pre-trained initialization.

Why cosine schedule? The linear warmup prevents large, destabilizing updates in the first few steps when the optimizer's moment estimates are still noisy. Cosine decay gradually reduces the learning rate, allowing the model to settle into a good minimum rather than oscillating.

Reranker Fine-tuning:

- **Model**: `cross-encoder/ms-marco-MiniLM-L-12-v2` (33M parameters)
- **Optimizer**: AdamW with weight decay 0.01
- **Learning rate**: 3e-5 with linear warmup and cosine decay
- **Batch size**: 16
- **Loss**: Binary cross-entropy
- **Training data**: Query-passage pairs labeled by legal experts (relevant / not relevant)
- **Hard negatives**: For each positive pair, include 3 hard negatives — passages that are semantically similar but legally irrelevant (e.g., same topic but wrong jurisdiction)

TODO (Student): - Implement the fine-tuning loop for the embedding model using the InfoNCE loss - Implement hard negative mining: given a positive pair (query, relevant_passage), find the top-3 most similar but irrelevant passages as hard negatives - Train for 3 epochs and plot the training loss curve - Compare retrieval Relevance@10 before and after fine-tuning

3.6 Evaluation

Quantitative Evaluation:

Evaluate the complete pipeline (filtered retrieval + reranking + context assembly + generation) on a held-out test set of 50 legal queries with expert-annotated relevance judgments.

TODO (Student): - Compute Relevance@K for K = 5, 10, 20 and plot the results - Compute Citation Accuracy on generated briefs (% of citations that are valid and from the correct jurisdiction) - Compare against the BM25 baseline across all metrics - Create a summary table showing baseline vs. pipeline performance for each metric - Plot a precision-recall curve for the retrieval component

3.7 Error Analysis

Systematic Error Categorization:

After running the pipeline on the test queries, categorize errors into the four failure modes from the article:

1. **Poisoning errors**: Cases where an invalid citation appeared in the output despite the validity check
2. **Distraction errors**: Cases where a relevant precedent was retrieved but the model ignored it in the generated brief
3. **Confusion errors**: Cases where irrelevant documents made it through filtering and degraded the output
4. **Clash errors**: Cases where the generated brief contradicted the attorney's stated strategy

TODO (Student): - Run the pipeline on all 50 test queries and manually categorize the errors - For each error type, compute: (a) frequency, (b) severity (1–3 scale), (c) root cause - Identify the top 3 failure modes by frequency and propose a specific fix for each - Answer: Which failure mode is hardest to detect automatically? Why?

3.8 Scalability and Deployment Considerations

Latency Profiling:

Profile the end-to-end pipeline to identify bottlenecks:

- Query parsing: expected < 10ms
- Metadata filtering: expected < 100ms
- Embedding + ANN search: expected < 500ms
- Reranking (20 candidates): expected < 2s
- Context assembly: expected < 50ms
- LLM generation: expected < 85s

TODO (Student): - Instrument each pipeline stage with timing measurements - Run the pipeline on 10 queries and compute mean and P95 latency per stage - Identify the top 2 latency bottlenecks - Propose one optimization for each bottleneck (e.g., batch embedding, caching, model distillation) - Write a simple inference benchmarking script that measures tokens-per-second for the full pipeline

3.9 Ethical and Regulatory Analysis

Legal AI Ethics:

Legal AI raises specific ethical concerns that differ from general AI applications:

- **Unauthorized practice of law**: AI-generated briefs must be reviewed by a licensed attorney. The system must never present itself as providing legal advice.
- **Bias in case law**: Historical case law reflects historical biases. If the retrieval system disproportionately surfaces cases from eras with discriminatory rulings, the AI may perpetuate those biases.
- **Access to justice**: AI legal tools could democratize legal research for smaller firms and pro bono work. But they could also widen the gap if only large firms can afford them.
- **Confidentiality**: Attorney-client privilege must be maintained. The system must not leak information between clients.
- **Explainability**: Attorneys need to understand why the AI chose specific citations. Black-box retrieval is unacceptable.

TODO (Student): - Analyze the corpus for temporal bias: are older (potentially outdated) cases over-represented in retrievals? Run a retrieval experiment and plot the distribution of retrieved case dates. - Propose a fairness metric specific to legal AI (hint: consider whether the system retrieves cases equitably across different demographic contexts) - Write a 500-word ethical impact assessment addressing: (a) who benefits from this system, (b) who could be harmed, (c) what safeguards should be in place, and (d) whether this system should be deployed without attorney review

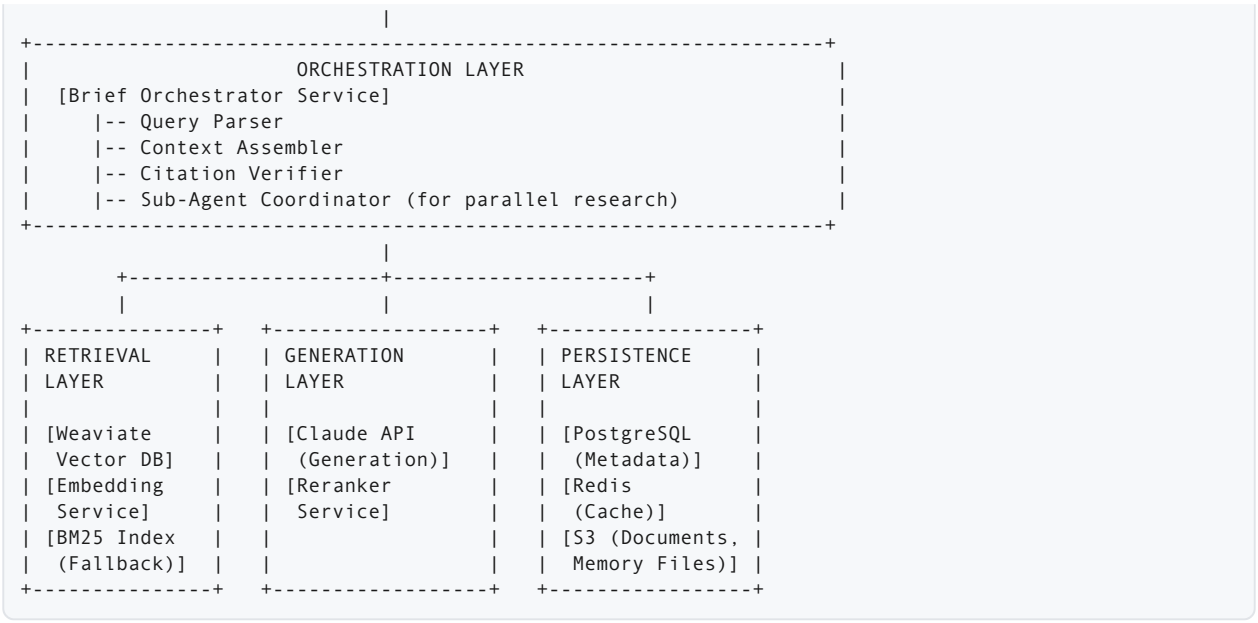
---

Section 4: Production and System Design Extension

This section extends the implementation into a production system design. It is intended for advanced students who want to understand how the context engineering pipeline would operate at scale in BriefEngine's production environment.

Architecture Diagram

The production system follows a layered architecture:



The orchestration layer is the heart of the context engineering pipeline. The Brief Orchestrator Service coordinates all stages: it parses the query, calls the retrieval layer, assembles context, calls the generation layer, verifies citations, and returns the final brief. The Sub-Agent Coordinator implements the **Isolate** strategy by dispatching parallel research queries to independent sub-agents, each with their own context window.

API Design

REST API Endpoints:

```
POST /api/v1/briefs
  Request:
    {
      "case_facts": "string (500-2000 tokens)",
      "legal_question": "string (50-200 tokens)",
      "jurisdiction": "string (e.g., 'CA', '9th-circuit')",
      "strategy_notes": "string (optional)",
      "date_range": {"start": 2000, "end": 2024},
      "output_format": "markdown" | "docx" | "pdf"
    }
  Response (streaming, SSE):
    event: progress
    data: {"stage": "retrieval", "status": "complete", "docs_found": 47}

    event: progress
    data: {"stage": "reranking", "status": "complete", "docs_selected": 10}

    event: chunk
    data: {"text": "## Statement of Facts\n\n..."}

    event: citations
    data: {"verified": 12, "flagged": 1, "details": [...]}

    event: done
    data: {"brief_id": "uuid", "latency_ms": 78400}

GET /api/v1/briefs/{brief_id}
  Returns the complete brief with metadata and citation verification report.

POST /api/v1/briefs/{brief_id}/feedback
  Request:
    {
      "quality_score": 4,
      "citation_issues": ["Henderson v. Pacific Mutual incorrectly cited"],
      "strategy_alignment": 5,
```

```
      "comments": "Good structure, needed minor jurisdiction corrections"
    }
  Feedback is stored for end-to-end context quality optimization.

GET /api/v1/retrieval/preview
  Request (query params): legal_question, jurisdiction, top_k
  Returns retrieved documents before generation -- allows attorneys to
  inspect context before committing to a full brief generation.
```

Serving Infrastructure

- **Model serving**: Claude 3.5 Sonnet via Anthropic API (no self-hosting for the generation model). The embedding model and reranker are served via Triton Inference Server on GPU instances.
- **Embedding service**: 2x g5.xlarge instances (NVIDIA A10G) behind an ALB, serving the fine-tuned E5-large model. Batch inference with dynamic batching (max batch size 64, max latency 100ms).
- **Reranker service**: 1x g5.xlarge instance serving the cross-encoder. Processes up to 20 query-document pairs per request.
- **Vector database**: Weaviate cluster (3 nodes, 64GB RAM each) with HNSW index. Configured for recall@20 > 0.95 with ef=128.
- **Scaling strategy**: Horizontal scaling of embedding and reranker services based on request queue depth. The vector database scales by adding read replicas. The orchestrator is stateless and scales horizontally behind a load balancer.
- **Auto-scaling triggers**: Scale up when P95 latency exceeds 100s or queue depth exceeds 50 requests. Scale down when utilization drops below 30% for 15 minutes.

Latency Budget

End-to-end target: < 90 seconds (P95).

Stage	Target (P50)	Target (P95)	Notes
Query parsing	5ms	15ms	CPU-only, regex + validation
Metadata filtering	50ms	150ms	PostgreSQL query with indexes
Embedding (query)	20ms	50ms	Single vector, GPU
ANN search	100ms	300ms	Weaviate HNSW, ef=128
Reranking (20 docs)	800ms	2,000ms	Cross-encoder, GPU batch
Context assembly	30ms	80ms	CPU, token counting + formatting
LLM generation	45,000ms	75,000ms	Claude API, streaming
Citation verification	200ms	500ms	Index lookup + regex
Total	**46,205ms**	**78,095ms**	**Within 90s budget**

The LLM generation stage dominates latency. Strategies to reduce it: (a) request shorter outputs when the brief scope is narrow, (b) use prompt caching for the system prompt and frequently-

used case law chunks, (c) stream output to the attorney so they can begin reading before generation completes.

Monitoring

Key Metrics Dashboard:

- **Retrieval quality**: Relevance@10 (sampled daily from expert annotations), context utilization ratio
- **Generation quality**: Citation accuracy (automated check on every brief), attorney quality scores (from feedback endpoint)
- **Latency**: P50, P95, P99 for each pipeline stage. Alert if P95 > 100s.
- **Error rates**: 5xx error rate, citation verification failure rate, context budget overflow rate
- **Token usage**: Average tokens per context assembly, average output tokens, monthly API cost
- **Business metrics**: Briefs generated per day, average attorney review time (from feedback), customer NPS

Alerting Thresholds:

Alert	Threshold	Severity
P95 latency > 100s	5-minute sustained	P1 (page on-call)
Citation accuracy < 90%	Rolling 1-hour window	P1
5xx error rate > 2%	5-minute window	P2
Relevance@10 < 70%	Daily batch evaluation	P2
Context overflow rate > 5%	1-hour window	P3

Model Drift Detection

Legal context engineering faces two types of drift:

1. **Corpus drift**: New case law is published continuously. Cases get overruled. The vector index becomes stale.
2. **Detection**: Weekly comparison of citation validity index against legal update feeds (e.g., Shepard's Citations). Alert if > 1% of indexed cases have changed validity status since last update.
3. **Mitigation**: Nightly incremental index updates. Full reindex monthly.
4. **Query drift**: Attorney usage patterns change over time (new practice areas, new types of cases).
5. **Detection**: Monitor the distribution of query embeddings weekly using Maximum Mean Discrepancy (MMD) between current week and baseline. Alert if MMD > threshold.

6. **Mitigation**: Retrain embedding model quarterly on recent query-passage pairs.
7. **Quality drift**: Gradual degradation of generation quality that is not captured by automated metrics.
8. **Detection**: Weekly sample of 50 generated briefs reviewed by legal domain experts. Track rolling quality score.
9. **Mitigation**: If quality score drops below 3.8/5.0 for two consecutive weeks, trigger a full pipeline review.

Model Versioning

- **Embedding model**: Versioned by date and training data hash (e.g., `e5-legal-v3-20250115-abc123`). Each version stored in S3 with metadata (training data, hyperparameters, evaluation metrics).
- **Reranker model**: Same versioning scheme.
- **Context assembly configuration**: Version-controlled in Git (chunk size, token budgets, prompt templates, retrieval parameters).
- **Rollback strategy**: Each deployment tags the current model version. Rollback replaces the serving model with the previous tagged version. Rollback latency < 5 minutes (just update the model path in Triton).
- **Version compatibility**: The vector index must be rebuilt when the embedding model changes. This takes approximately 4 hours for the full 42M-chunk corpus. During reindexing, the previous index remains live.

A/B Testing

Framework:

- Traffic splitting at the API gateway level. Attorneys are randomly assigned to experiment or control groups (sticky assignment per user to ensure consistent experience).
- **Primary metric**: Attorney quality score (1-5) from the feedback endpoint.
- **Guardrail metrics**: Citation accuracy must not decrease by more than 2 percentage points. P95 latency must not increase by more than 10 seconds.
- **Statistical significance**: Two-tailed t-test with $\alpha = 0.05$ and minimum 200 briefs per group (based on power analysis assuming effect size of 0.3 standard deviations).
- **Minimum experiment duration**: 2 weeks (to account for weekday/weekend usage patterns and case type variation).
- **Kill switch**: If guardrail metrics are violated at any point during the experiment, the experiment group is automatically routed back to the control configuration.

CI/CD for ML

Training Pipeline:

1. **Trigger**: New training data available (weekly attorney feedback batch) or scheduled quarterly retrain
2. **Data validation**: Check training data for label quality (inter-annotator agreement > 0.8), class balance, and format consistency
3. **Training**: Run fine-tuning on dedicated GPU instances (4x A100, spot instances for cost)
4. **Evaluation gate**: New model must meet or exceed current production model on:
5. Relevance@10 on held-out test set (must not decrease by > 1%)
6. Citation accuracy on held-out test set (must not decrease by > 0.5%)
7. Latency benchmark (must not increase P95 by > 10%)
8. **Shadow deployment**: New model serves traffic in shadow mode (results logged but not shown to users) for 48 hours
9. **Canary deployment**: 5% of traffic routed to new model for 72 hours. Monitor guardrail metrics.
10. **Full rollout**: If canary passes, gradual rollout over 24 hours (5% -> 25% -> 50% -> 100%)
11. **Reindex**: If the embedding model changed, trigger vector index rebuild in parallel with rollout

Infrastructure: - Training orchestration: Kubernetes Jobs with GPU node pools - Experiment tracking: MLflow (self-hosted) - Model registry: S3 + DynamoDB metadata store - Pipeline orchestration: Airflow for scheduled retraining, GitHub Actions for CI

Cost Analysis

Monthly Cloud Compute Costs (Estimated):

Component	Instance/Service	Count	Monthly Cost
Embedding service	g5.xlarge (A10G)	2	2,400 USD
Reranker service	g5.xlarge (A10G)	1	1,200 USD
Weaviate cluster	r6g.2xlarge	3	2,700 USD
Orchestrator	c6g.xlarge	3	540 USD
PostgreSQL (RDS)	db.r6g.large	1	400 USD
Redis (ElastiCache)	r6g.large	1	280 USD
S3 storage	~2TB	-	50 USD
Claude API	~15M input + 3M output tokens/day	-	32,000 USD
Training (quarterly, amortized)	4x A100 spot, ~8 hours	-	350 USD
Total			**39,920 USD/month**

The Claude API cost dominates at approximately 80% of total infrastructure spend. Key cost optimization strategies:

1. **Prompt caching**: Cache the system prompt and frequently-retrieved case law chunks. Anthropic's prompt caching feature can reduce input token costs by up to 90% for cached prefixes. Estimated savings: 8,000–12,000 USD/month.
2. **Context efficiency**: Better context engineering means fewer wasted tokens. Improving context utilization from 60% to 80% could reduce input tokens by 25%. Estimated savings: 4,000–6,000 USD/month.
3. **Output length optimization**: Smarter prompts that produce concise briefs (without sacrificing quality) reduce output token costs. Target: 15% reduction in average output tokens.
4. **Tiered model routing**: Use a smaller, cheaper model (Claude Haiku) for citation verification and query parsing, reserving Sonnet for brief generation. Estimated savings: 2,000 USD/month.

With all optimizations, the target monthly cost is approximately 25,000–28,000 USD — within BriefEngine's 45K USD/month compute budget with headroom for growth.