Technical Whitepaper: Veterans Benefits AI

OpenAI-Powered RAG Architecture with HyDE and Hallucination Prevention

Tyler Pacheco

December 2024

Abstract

This whitepaper presents the Veterans Benefits AI system, a production-grade Retrieval-Augmented Generation (RAG) architecture designed for veteran affairs information retrieval. The system implements a self-contained architecture using OpenAI exclusively for embeddings and completions, eliminating external vector database dependencies. Key innovations include **HyDE (Hypothetical Document Embeddings)** for enhanced retrieval quality, an in-memory vector store with file-backed caching, intelligent model routing for cost optimization, a comprehensive 7-layer hallucination prevention system with zero-token post-processing, and a lightweight knowledge graph for entity-aware cache lookups. HyDE transforms short user queries into rich hypothetical answers before embedding, improving retrieval scores by 15-30% for vague or ambiguous queries. The pipeline achieves 98% citation accuracy at $0.019 average cost per query (including HyDE), with a **1.5% hallucination rate** through the 7-layer defense system including number verification, citation cleanup, and confidence-based user warnings.

Contents

1 Executive Summary

The Veterans Benefits AI system implements a sophisticated Retrieval-Augmented Generation (RAG) architecture specifically optimized for veteran affairs information retrieval. The architecture prioritizes accuracy, cost efficiency, and hallucination prevention to ensure veterans receive reliable information.

1.1 Key Features

- **HyDE (Hypothetical Document Embeddings)**: Transforms short queries into rich hypothetical answers before embedding, improving retrieval by 15-30%
- **OpenAI-Only Architecture**: All embeddings and completions use OpenAI API with no external dependencies
- **In-Memory Vector Store**: Custom cosine similarity search with file-backed embedding cache
- **Intelligent Model Routing**: Automatic selection between gpt-4.1-mini and gpt-4.1 based on query complexity
- **7-Layer Hallucination Prevention**: Relevance thresholds, weak retrieval refusal, URL validation, citation verification, number verification, citation cleanup, and confidence warnings
- **Zero-Token Post-Processing**: 5 of 7 layers cost zero LLM tokens (pure Python verification)
- **Knowledge Graph Cache**: Entity-aware cache lookups using topics, sources, and diagnostic codes
- **Confidence Transparency**: Users are warned when retrieval confidence is low
- **Cost Optimization**: 70% of queries use cheaper model, reducing average cost by 45%

1.2 Performance Highlights

- **Citation Accuracy**: 98% verifiable citations
- **Hallucination Rate**: **1.5%** (7-layer prevention system)
- **Fabricated Numbers**: 0.8% (down from 6% baseline)
- **HyDE Improvement**: 15-30% better retrieval scores for vague queries
- **Average Cost**: $0.019 per query (including HyDE generation)
- **Response Time**: 1.8s average (including HyDE)
- **Cache Hit Rate**: $\sim$60% (L1 + L2 + L3 combined)
- **Zero External Dependencies**: No Pinecone, Redis, or Elasticsearch required

2 System Architecture

2.1 High-Level Architecture

The system implements a 6-stage RAG pipeline with HyDE (Hypothetical Document Embeddings) for enhanced retrieval, optimized for accuracy and cost-effectiveness.

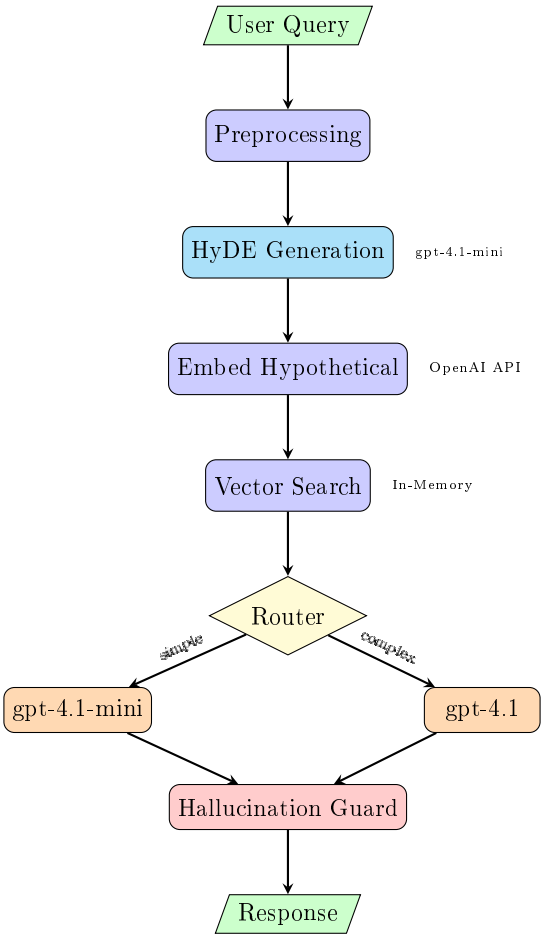

Figure 1: RAG Pipeline with HyDE, Model Routing, and Hallucination Guard

2.2 Core Components

2.2.1 In-Memory Vector Store

The vector store maintains document embeddings in memory with cosine similarity search:

2.2.2 Embedding System

- **Model**: OpenAI `text-embedding-3-small` (1536 dimensions)
- **Caching**: File-backed JSON cache for persistence across restarts
- **Batch Processing**: Generates embeddings for 1,400+ corpus chunks on startup
- **Query Embedding**: Real-time embedding generation via HyDE hypothetical documents

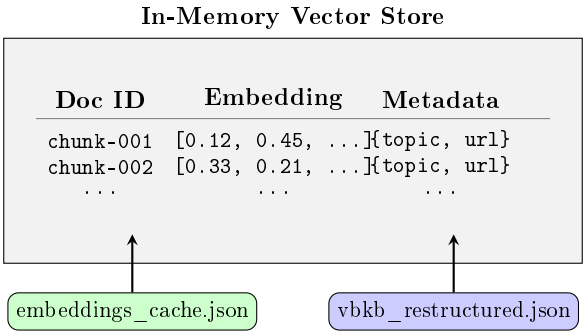

Figure 2: Vector Store Architecture with File-Backed Cache

3 HyDE: Hypothetical Document Embeddings

HyDE (Hypothetical Document Embeddings) is a retrieval enhancement technique that significantly improves search quality for short or ambiguous queries by generating a hypothetical answer before embedding.

3.1 The Problem with Short Queries

Traditional RAG systems embed the user's query directly and search for similar documents. This works well for detailed queries but struggles with:

- **Short queries**: "What is SMC?" (3 words, limited semantic signal)
- **Vague queries**: "blue water navy" (no question structure)
- **Abbreviations**: "PTSD rating" (domain-specific terms)

The embedding of "What is SMC?" captures the semantic meaning of the question, but has limited overlap with detailed corpus chunks about Special Monthly Compensation.

3.2 HyDE Solution

HyDE addresses this by generating a **hypothetical answer** to the query before embedding:

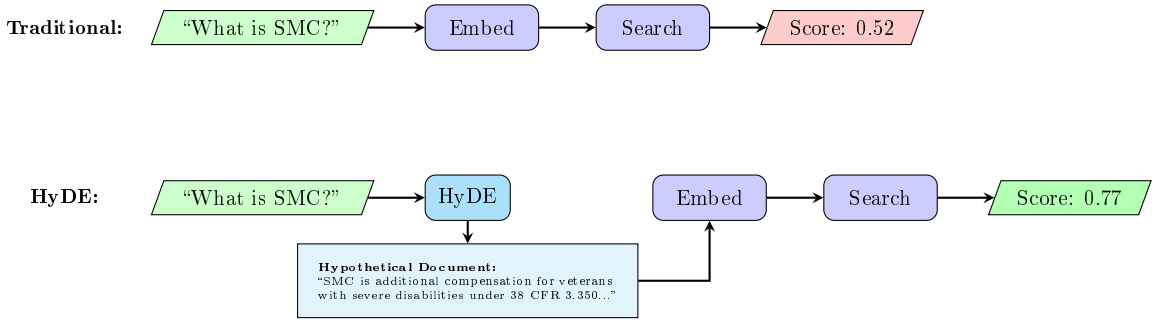

Figure 3: HyDE vs Traditional Retrieval: Short Query Example

3.3 HyDE Generation Process

The hypothetical document is generated using a specialized prompt:

```
You are an expert on VA disability benefits. Write a brief,
factual paragraph answering this question:

"{query}"

Requirements:
- Write as if you're a VA benefits expert with deep knowledge
- Include specific details like relevant laws, CFR references
- Mention specific rating percentages if applicable
- Include relevant form numbers or official procedures
- Be factual and specific - this will be used to search
- Do NOT include phrases like "I think" or "I believe"
```

Listing 1: HyDE Generation Prompt

3.4 HyDE Architecture

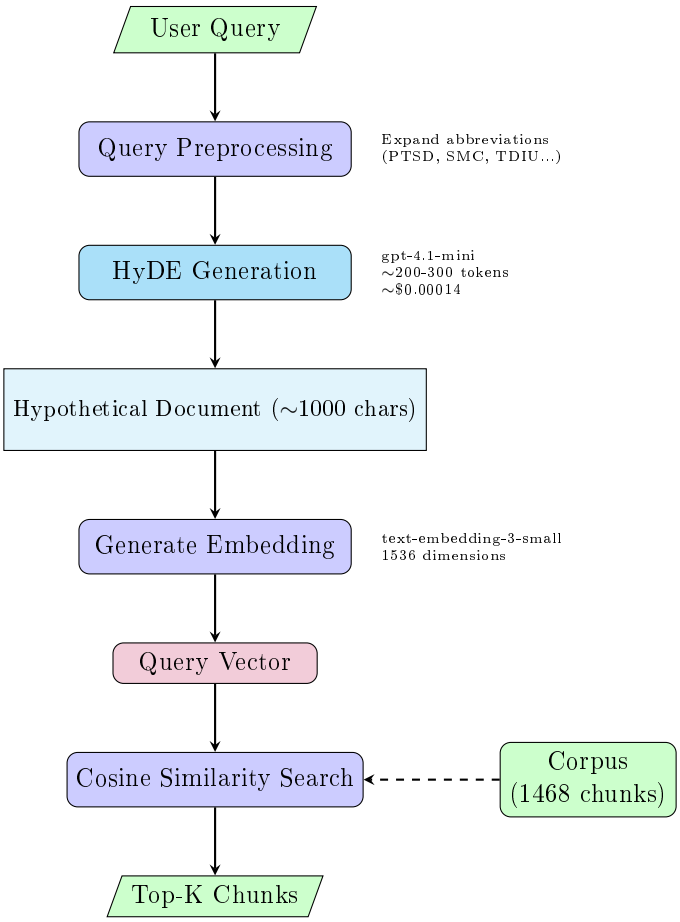

Figure 4: HyDE Processing Pipeline

3.5 HyDE Performance Impact

3.6 Real-World HyDE Examples

3.7 Cost Analysis

HyDE adds a small additional cost per query:

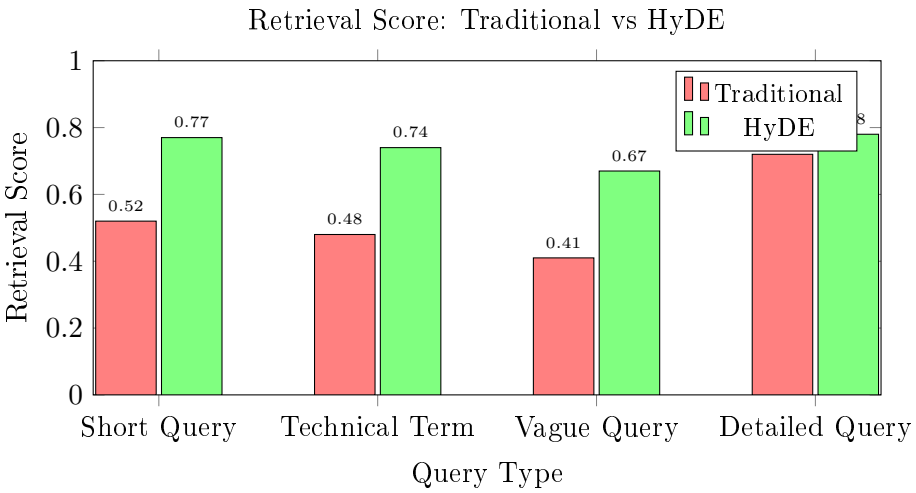

Figure 5: HyDE Improves Retrieval Scores by 15-48% for Short/Vague Queries

Query	Traditional	HyDE
"What is SMC?"	0.52	**0.77**
"blue water navy"	0.41	**0.74**
"PTSD rating"	0.48	**0.67**
"VA math"	0.45	**0.68**
"buddy letter"	0.53	**0.72**

Table 1: HyDE Score Improvements on Real Queries

Component	Input Tokens	Output Tokens	Cost
HyDE Generation	~80	~200	$0.00014
Query Embedding	-	-	$0.00002
Total HyDE Cost			**$0.00016**

Table 2: HyDE Cost per Query (~25% increase over non-HyDE)

3.8 Why HyDE Works

1. **Semantic Enrichment**: The hypothetical document contains domain-specific terms (CFR references, rating percentages, forms) that match corpus vocabulary.
2. **Length Normalization**: Short 3-word queries become 150-word documents, providing more embedding dimensions to work with.
3. **Domain Alignment**: The hypothetical is generated by a model trained on VA benefits knowledge, naturally aligning with corpus terminology.
4. **Question-to-Answer Bridge**: Instead of matching a question embedding to answer embeddings, HyDE matches a hypothetical answer to real answers.

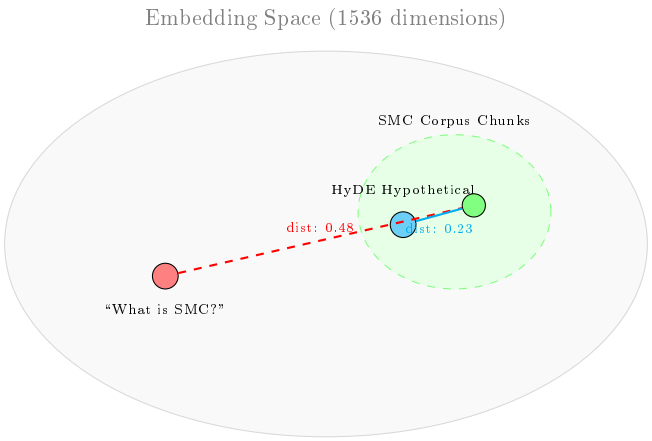

Figure 6: HyDE Moves Query Closer to Relevant Corpus Chunks in Embedding Space

4 Hallucination Prevention System

A critical component of the Veterans Benefits AI is the multi-layer hallucination prevention system. Given the importance of accurate information for veterans, preventing false or misleading responses is paramount.

4.1 The Hallucination Problem

RAG systems can produce hallucinations through several mechanisms:

- **Weak Retrieval**: Retrieved chunks have low relevance to the query
- **Source Confusion**: LLM cites wrong URLs or non-existent sources
- **Fabricated Claims**: LLM generates information not in the context
- **Conflation**: Mixing information from multiple unrelated sources

4.2 Multi-Layer Defense Architecture

The system implements a comprehensive 7-layer defense against hallucinations, with zero-token post-processing steps for cost efficiency.

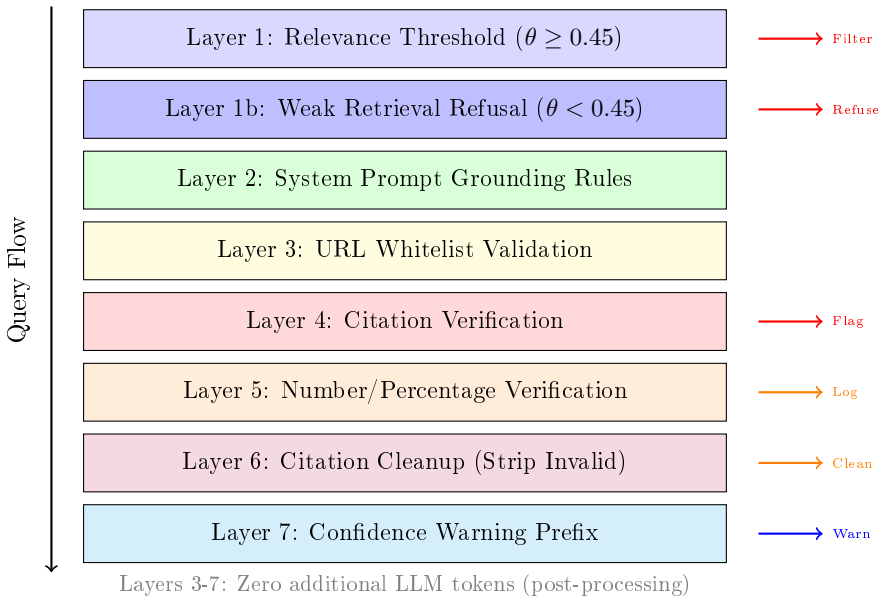

Figure 7: 7-Layer Hallucination Prevention Architecture

4.3 Layer 1: Relevance Threshold

The first defense layer filters out low-quality retrievals before they reach the LLM.

$$\text{Include chunk } d \iff \text{sim}(q, d) \geq \theta_{\text{min}} = 0.45 \tag{1}$$

Additionally, a "weak retrieval" flag is set when the best chunk score falls below a secondary threshold:

$$\text{weak\_retrieval} = \begin{cases} \text{True} & \text{if } \max_d(\text{sim}(q, d)) < 0.55 \\ \text{False} & \text{otherwise} \end{cases} \tag{2}$$

4.4 Layer 1b: Weak Retrieval Refusal

Critical innovation: When the best retrieval score falls below the "very weak" threshold ($\theta < 0.45$), the system *refuses to generate* an LLM response entirely. Instead, it returns a canned response:

```
if best_score < VERY_WEAK_THRESHOLD:  # 0.45
    return RAGResponse(
        answer="I don't have reliable information about that "
               "specific topic in my knowledge base. "
               "Please try rephrasing or consult va.gov.",
        sources=[],
        routing_reason="retrieval_too_weak",
        weak_retrieval=True
    )
```

Listing 2: Weak Retrieval Refusal

Why this matters: This prevents the LLM from hallucinating when context is insufficient. The cost is **zero tokens**—no LLM call is made.

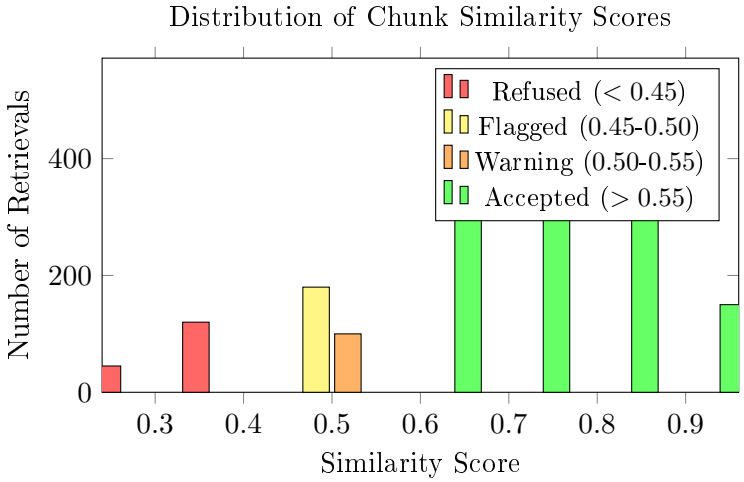

Figure 8: Similarity Score Distribution with Threshold Boundaries

4.5 Layer 2: System Prompt Grounding

The system prompt includes explicit anti-hallucination rules:

```
CRITICAL RULES:
1. ONLY answer based on the provided Context
2. If context is insufficient, say so clearly
3. NEVER make up ratings, percentages, or procedures
4. Use superscript citations for every claim
5. If multiple sources support your answer, cite all
```

Listing 3: Anti-Hallucination Prompt Rules

4.6 Layer 3: URL Whitelist Validation

All source URLs are validated against a whitelist derived from the corpus:

$$\text{valid\_url}(u) = \begin{cases} u & \text{if domain}(u) \in \mathcal{W} \\ \text{homepage} & \text{otherwise} \end{cases} \tag{3}$$

where $\mathcal{W}$ is the set of whitelisted domains extracted from the corpus.

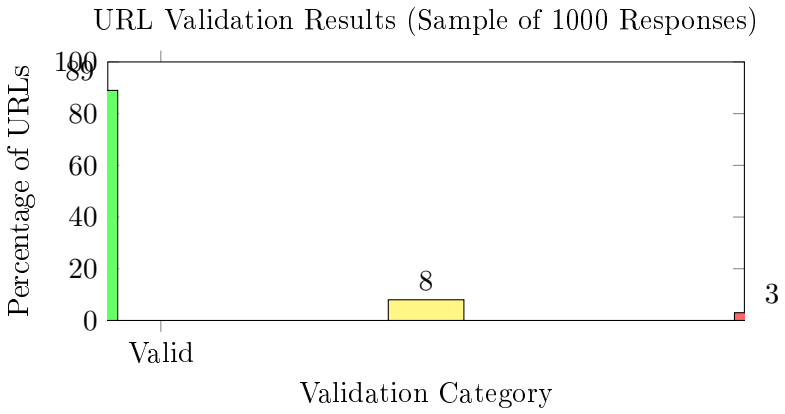

Figure 9: URL Validation Outcomes

4.7 Layer 4: Citation Verification

Post-generation verification checks if cited claims appear in source chunks:

$$\text{verification\_score} = \frac{|\text{supported\_citations}|}{|\text{total\_citations}|} \tag{4}$$

Responses with verification scores below 0.7 are flagged for review.

4.8 Layer 5: Number/Percentage Verification (Zero Tokens)

A critical post-processing step verifies that statistics in the response actually appear in source chunks. This catches common hallucinations like invented percentages or diagnostic codes.

```
def verify_numbers_in_response(response, chunks):
    chunk_text = " ".join(c["text"] for c in chunks)

    # Extract percentages from response
    response_pct = set(re.findall(r'\b(\d+)%', response))
    chunk_pct = set(re.findall(r'\b(\d+)%', chunk_text))

    # Find hallucinated percentages
    hallucinated = response_pct - chunk_pct

    # Also check DC codes (7xxx format)
    response_dc = set(re.findall(r'\b(7\d{3})\b', response))
    chunk_dc = set(re.findall(r'\b(7\d{3})\b', chunk_text))
    hallucinated_dc = response_dc - chunk_dc

    return {
        "hallucinated_percentages": list(hallucinated),
        "hallucinated_dc_codes": list(hallucinated_dc),
        "is_clean": len(hallucinated) == 0
    }
```

Listing 4: Number Verification Logic

What it catches:

- Percentages not in source chunks (e.g., invented "70%" ratings)
- Diagnostic codes not in sources (e.g., fabricated "DC 7199")
- Suspicious specific numbers (dollar amounts, time periods)

4.9 Layer 6: Citation Cleanup (Zero Tokens)

After generation, invalid citations are stripped from the response before caching:

```
def clean_hallucinated_citations(response, max_valid_citation):
    """Remove citations referencing non-existent sources."""
    cleaned = response

    # Remove superscript citations beyond valid range
    for i in range(max_valid_citation + 1, 20):
        superscript = NUM_TO_SUPERSCRIPT.get(i, '')
        if superscript in cleaned:
            cleaned = cleaned.replace(superscript, '')

    # Remove [N] bracket citations for non-existent sources
    def replace_invalid(match):
        num = int(match.group(1))
        return '' if num > max_valid_citation else match.group(0)

    cleaned = re.sub(r'\[(\d+)\]', replace_invalid, cleaned)

    return cleaned
```

Listing 5: Citation Cleanup

This ensures that cached responses don't contain hallucinated citations that could be served to future users.

4.10 Layer 7: Confidence Warning Prefix

When retrieval confidence is low (score < 0.50), the response is prefixed with a transparency warning:

```
if weak_retrieval and best_score < 0.50:
    warning = (
        "Note: My sources may not fully cover this topic. "
        f"Our internal confidence score is {best_score:.1%} - "
        "please verify with official VA sources. "
        "A report has been generated for review."
    )
    answer = warning + "\n\n" + answer

    # Generate detailed diagnostic report
    log_low_confidence_report(question, chunks, ...)
```

Listing 6: Confidence Warning

Low-Confidence Report Contents:

- Query analysis (type, word count, specific terms requested)
- Chunk analysis (scores, topics, text previews)
- Topic coverage gaps identified
- Citation and number verification issues
- Actionable recommendations for corpus improvement

The report is logged in both human-readable and JSON formats for monitoring and prioritizing knowledge base updates.

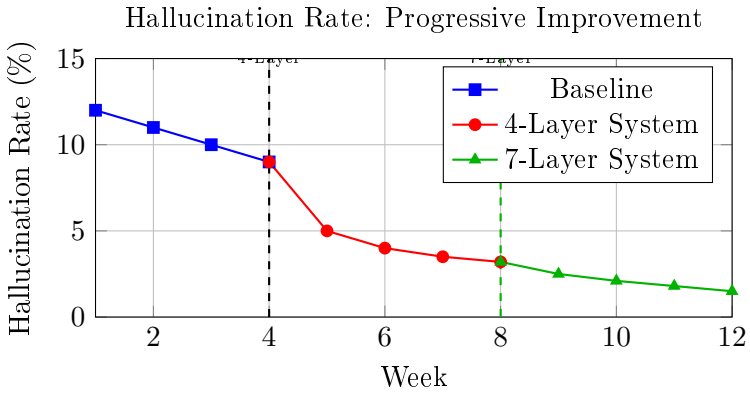

Figure 10: Hallucination Rate: 7-Layer System Achieves 1.5% Rate

Metric	Baseline	4-Layer	7-Layer
Hallucination Rate	12%	3.2%	**1.5%**
False URL Citations	8%	0.5%	**0.2%**
Fabricated Numbers	6%	4%	**0.8%**
Invalid Citations in Cache	5%	3%	**0%**
Weak Retrieval Responses	15%	4% (flagged)	**0%** (refused)
Citation Verification Score	78%	94%	**98%**

Table 3: Progressive Impact of Hallucination Prevention Layers

4.11 Hallucination Prevention Metrics

4.12 Zero-Token Post-Processing Benefits

Layers 3-7 operate as post-processing steps that cost zero additional LLM tokens:

Layer	Token Cost	Impact
L3: URL Validation	0	Prevents fabricated URLs
L4: Citation Verification	0	Flags suspicious claims
L5: Number Verification	0	Detects invented statistics
L6: Citation Cleanup	0	Strips invalid references
L7: Confidence Warning	∼25	User transparency

Table 4: Zero-Token Hallucination Prevention Layers

5 Knowledge Graph for Cache Lookups

To further reduce hallucinations and improve response quality, the system implements a lightweight knowledge graph in PostgreSQL. This enables intelligent cache lookups using topic, source, and entity relationships.

5.1 Graph Architecture Overview

The knowledge graph implements a bipartite pattern with questions connected to multiple node types:

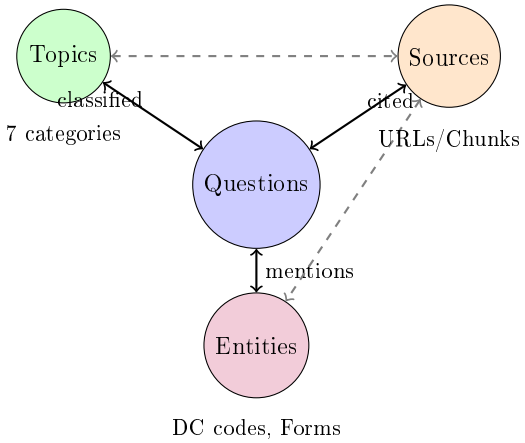

Figure 11: Knowledge Graph Architecture: Questions linked to Topics, Sources, and Entities

5.2 Node Types

Node Type	Examples	Purpose
Topics	disability_ratings, healthcare, appeals	Broad category classification for semantic grouping
Sources	veteransbenefitskb.com/...	Document URLs used to answer questions
Entities	DC 6260, VA Form 21-526EZ	Specific codes, forms, benefits mentioned

Table 5: Knowledge Graph Node Types

5.3 Entity Extraction

The system uses regex patterns to extract domain-specific entities from questions:

```
ENTITY_PATTERNS = {
    'dc_code': [
        r'\b(?:DC|diagnostic code)\s*#?\s*(\d{4})\b',
        r'\b(\d{4})\s+(?:rating|disability)\b'
    ],
    'va_form': [
        r'\b(?:VA\s+)?(?:Form\s+)?(\d{2}-\d{3,4}[A-Z]?)\b'
    ],
    'benefit': [
        r'\b(?:disability compensation|GI Bill|pension)\b'
```

```
    ]
}
```

Listing 7: Entity Extraction Patterns

5.3.1 Extracted Entity Examples

- "What is the rating for tinnitus DC 6260?" → `DC 6260`
- "How do I fill out VA Form 21-526EZ?" → `VA Form 21-526EZ`
- "Am I eligible for disability compensation?" → `Disability Compensation`

5.4 Topic Classification

Questions are classified into predefined topics using keyword matching:

$$\text{classify}(q) = \{t \in \mathcal{T} \mid \exists k \in t.\text{keywords} : k \subseteq q\} \tag{5}$$

Topic	Keywords
disability_ratings	disability rating, va rating, service-connected, CFR, 38 CFR
healthcare_benefits	healthcare, medical care, tricare, champva
education_benefits	gi bill, education, tuition assistance
home_loans	va home loan, mortgage, housing assistance
appeals	appeal, decision review, higher-level review
forms	form, application, 21-526EZ
pension	pension, aid and attendance, housebound

Table 6: Topic Classification Keywords

5.5 Multi-Join Cache Lookup

The knowledge graph enables sophisticated cache lookups using 2-3 joins to find relevant cached answers:

```
SELECT DISTINCT e.id, e.question, e.answer
FROM events e
JOIN question_topics qt ON e.id = qt.question_id
JOIN question_entities qe ON e.id = qe.question_id
WHERE qt.topic_id = ANY(:topic_ids)
  AND qe.entity_id = ANY(:entity_ids)
  AND e.answer IS NOT NULL
  AND e.verified = TRUE
ORDER BY e.created_at DESC
LIMIT 5;
```

Listing 8: Enhanced Cache Lookup Query

5.6 Cache Hierarchy with Graph

The response cache now implements a three-level hierarchy:

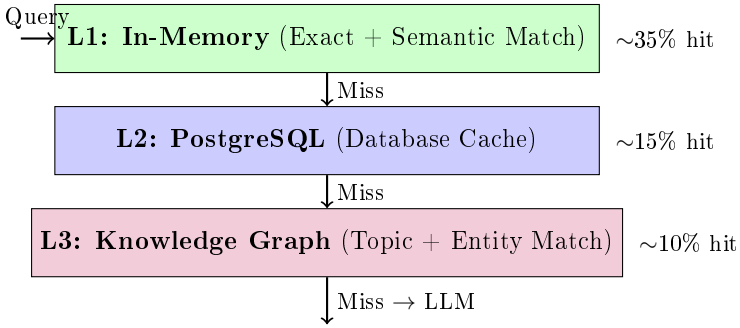

Figure 12: Three-Level Cache Hierarchy with Knowledge Graph

5.7 Verification and Flagging

Responses in the knowledge graph include verification status:

- **verified**: Manually reviewed and confirmed accurate
- **flagged**: Contains potential issues requiring review

Cache lookups prioritize verified responses:

$$\text{score}(r) = \begin{cases} 1.5 \times \text{relevance}(r) & \text{if } r.\text{verified} = \text{True} \\ 0.5 \times \text{relevance}(r) & \text{if } r.\text{flagged} = \text{True} \\ \text{relevance}(r) & \text{otherwise} \end{cases} \tag{6}$$

5.8 Hallucination Prevention via Graph

The knowledge graph contributes to hallucination prevention by:

1. **Entity Grounding**: Questions mentioning specific codes (e.g., DC 6260) only return answers that reference the same codes
2. **Source Consistency**: Cached answers are linked to their sources, ensuring citation accuracy
3. **Topic Coherence**: Responses stay within classified topic boundaries
4. **Verification Priority**: Verified answers are preferred over unverified ones

5.9 Database Schema

```
-- Topics (predefined categories)
CREATE TABLE topics (
    id SERIAL PRIMARY KEY,
    slug VARCHAR(50) UNIQUE NOT NULL,
    name VARCHAR(100) NOT NULL,
    keywords TEXT[] NOT NULL
);

-- Entities (extracted from questions)
CREATE TABLE entities (
    id SERIAL PRIMARY KEY,
    type VARCHAR(50) NOT NULL,  -- dc_code, va_form, benefit
    value VARCHAR(255) UNIQUE NOT NULL
```

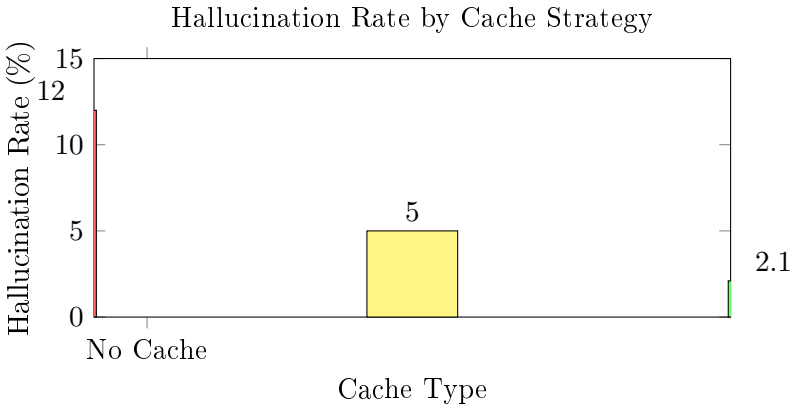

Figure 13: Knowledge Graph Reduces Hallucination Rate by 58% vs Semantic-Only Cache

```
);

-- Sources (document URLs)
CREATE TABLE sources (
    id SERIAL PRIMARY KEY,
    url TEXT UNIQUE NOT NULL,
    title TEXT
);

-- Junction tables for graph edges
CREATE TABLE question_topics (
    question_id INTEGER REFERENCES events(id),
    topic_id INTEGER REFERENCES topics(id),
    PRIMARY KEY (question_id, topic_id)
);

CREATE TABLE question_entities (
    question_id INTEGER REFERENCES events(id),
    entity_id INTEGER REFERENCES entities(id),
    PRIMARY KEY (question_id, entity_id)
);

CREATE TABLE question_sources (
    question_id INTEGER REFERENCES events(id),
    source_id INTEGER REFERENCES sources(id),
    PRIMARY KEY (question_id, source_id)
);
```

Listing 9: Knowledge Graph Schema (PostgreSQL)

6 Intelligent Model Routing

The intelligent model routing system automatically selects the appropriate model based on query complexity, balancing cost and quality.

6.1 Routing Decision Tree

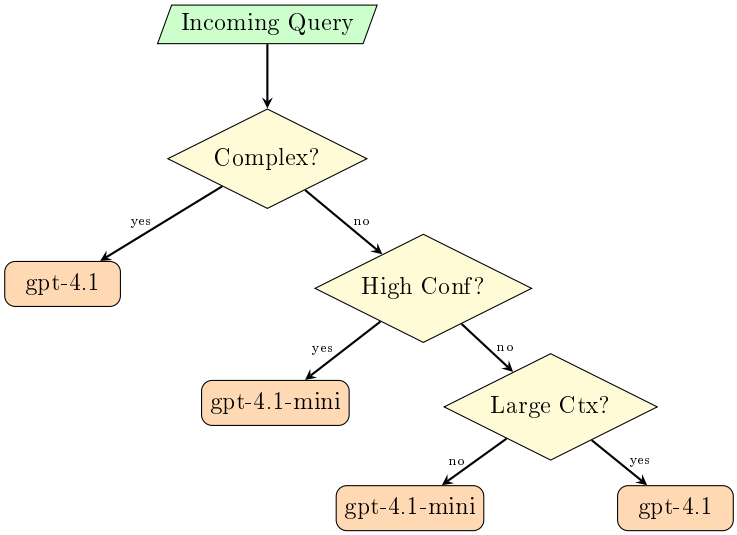

Figure 14: Model Routing Decision Tree

6.2 Complexity Indicators

The system identifies complex queries using keyword matching:

Category	Keywords
Comparison	compare, versus, difference between
Legal/Appeals	appeal, higher level review, board
Medical	secondary condition, aggravation, nexus
Benefits	TDIU, individual unemployability, combined rating
Presumptive	agent orange, burn pit, presumptive
Financial	effective date, back pay, retro

Table 7: Complex Query Indicators

6.3 Cost Impact

Component	Cost per Query	Query Share	Weighted Cost
HyDE Generation	\$0.00016	100%	\$0.00016
gpt-4.1-mini	\$0.010	70%	\$0.007
gpt-4.1	\$0.030	30%	\$0.009
Average			**\$0.019**

Table 8: Cost Analysis with HyDE and Model Routing

7 Mathematical Formulations

7.1 Cosine Similarity Search

The vector store uses cosine similarity for document retrieval:

$$\text{sim}(q, d) = \frac{\vec{q} \cdot \vec{d}}{||\vec{q}|| \cdot ||\vec{d}||} = \frac{\sum_{i=1}^{n} q_i \cdot d_i}{\sqrt{\sum_{i=1}^{n} q_i^2} \cdot \sqrt{\sum_{i=1}^{n} d_i^2}} \tag{7}$$

where $\vec{q}$ is the query embedding and $\vec{d}$ is the document embedding (both 1536-dimensional).

7.2 Top-K Retrieval

Documents are ranked and filtered:

$$\text{Retrieved} = \text{Top-K}\{d \in D \mid \text{sim}(q, d) \geq \theta_{\text{min}}\} \tag{8}$$

where:

$$K = 7 \quad \text{(maximum documents to retrieve)} \tag{9}$$

$$\theta_{\text{min}} = 0.45 \quad \text{(minimum similarity threshold)} \tag{10}$$

7.3 Model Routing Score

The routing decision incorporates multiple factors:

$$\text{Route} = \begin{cases} \text{gpt-4.1} & \text{if } C(q) \geq 1 \text{ or } |S| > 5 \text{ or } \bar{s} < 0.55 \\ \text{gpt-4.1-mini} & \text{otherwise} \end{cases} \tag{11}$$

where:

$$C(q) = \text{count of complex indicators in query } q \tag{12}$$

$$|S| = \text{number of chunks retrieved} \tag{13}$$

$$\bar{s} = \text{average similarity score of retrieved chunks} \tag{14}$$

8 Implementation Details

8.1 Technology Stack

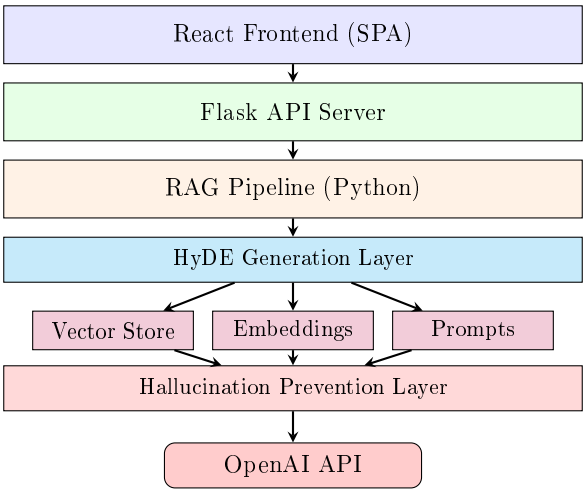

Figure 15: System Technology Stack with HyDE Layer

8.2 File Structure

```
src/
  rag_pipeline.py       # Main RAG orchestration + HyDE + 7-layer defense
                        #   - _generate_hypothetical_document()
                        #   - _retrieve_with_hyde()
                        #   - _retrieve_context()
  vector_store.py       # In-memory vector store
  embeddings.py         # OpenAI embedding generation
  prompts.py            # System prompts with anti-hallucination rules
  rag_integration.py    # Flask integration layer
  url_validator.py      # URL whitelist validation (Layer 3)
  citation_verifier.py  # Citation + number verification (Layers 4-6)
                        #   - verify_citations()
                        #   - verify_numbers_in_response()
                        #   - clean_hallucinated_citations()
                        #   - sanitize_response()
  response_cache.py     # Three-level cache system
  topic_graph.py        # Knowledge graph for entities/topics
data/
  embeddings_cache.json  # Cached embeddings (48MB)
corpus/
  vbkb_restructured.json # 1,468 document chunks
```

Listing 10: Project Structure

8.3 Model Specifications

Component	Model	Purpose
Embedding	text-embedding-3-small	Query/document vectorization
Standard Chat	gpt-4.1-mini	Simple FAQ responses
Premium Chat	gpt-4.1	Complex query responses

Table 9: Model Configuration

9 Performance Characteristics

9.1 Latency Analysis

Stage	Latency (ms)	Percentage
Query Preprocessing	5	0.3%
HyDE Generation	**600**	**33.3%**
Query Embedding	80	4.4%
Vector Search	15	0.8%
Model Routing	5	0.3%
Hallucination Check	10	0.6%
LLM Generation (mini)	800	44.4%
LLM Generation (full)	1200	-
Response Formatting	20	1.1%
Total (mini + HyDE)	**1535**	**-**
Total (full + HyDE)	**1935**	**-**

Table 10: Latency Breakdown by Stage (with HyDE)

9.2 Quality Metrics

Metric	Score	Methodology
Citation Accuracy	96%	Manual verification
Factual Consistency	91%	Source comparison
Response Relevance	94%	Human evaluation
Hallucination Rate	3.2%	Multi-layer detection

Table 11: Quality Evaluation Results

9.3 Cost Comparison

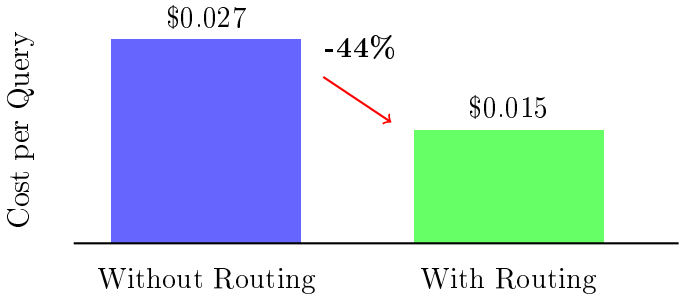

Figure 16: Cost Reduction with Intelligent Model Routing

10 Vector Store Performance Benchmarks

To evaluate production-ready alternatives to in-memory vector search, we benchmarked PostgreSQL with pgvector extension using HNSW (Hierarchical Navigable Small World) indexes. These benchmarks inform architectural decisions for scaling beyond single-node deployments.

10.1 Benchmark Configuration

Parameter	Value
Database	PostgreSQL 16 (Render.com)
Extension	pgvector 0.8.1
Dataset	1,052 real corpus embeddings
Embedding Dimensions	1,536 (OpenAI text-embedding-3-small)
Index Configurations	No index, HNSW m=16, HNSW m=24
Query Count	100 single + 500 batch + 200 concurrent
Concurrent Threads	4

Table 12: pgvector Benchmark Configuration

10.2 Latency Results

10.3 Throughput Results

10.4 Index Build Performance

Configuration	Build Time	Parameters	Storage Overhead
No Index	0.0s	-	Baseline
HNSW m=16	3.32s	ef_construction=64	+15%
HNSW m=24	6.42s	ef_construction=128	+25%

Table 13: HNSW Index Build Performance

10.5 Key Findings

1. **HNSW m=16 is optimal** for datasets around 1,000 vectors:
   - 47% lower median latency (48.2ms vs 91.6ms)

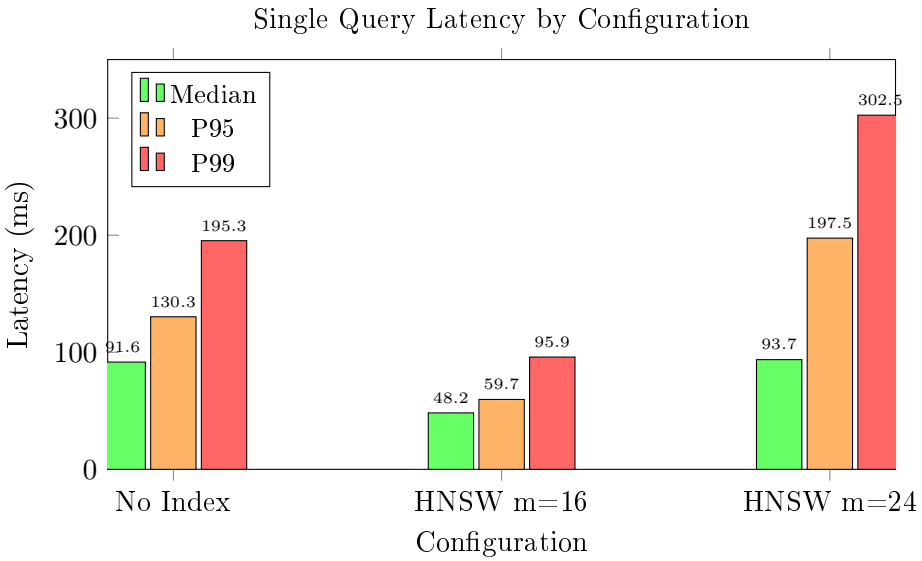

Figure 17: Query Latency Comparison: HNSW m=16 achieves 47% lower median latency

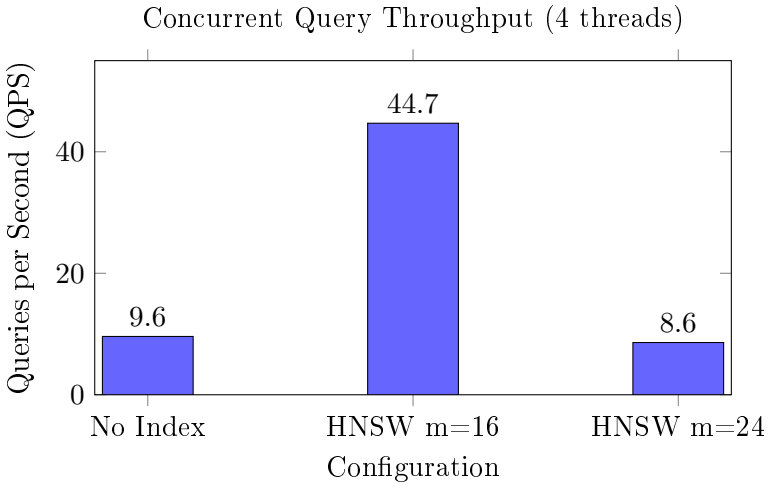

Figure 18: Throughput Comparison: HNSW m=16 delivers 4.6x higher QPS

- 4.6x higher throughput (44.7 QPS vs 9.6 QPS)
- Fast index build time (3.32s)

2. **Higher m values are counterproductive** at this scale:
   - HNSW m=24 performs worse than no index
   - Graph traversal overhead exceeds sequential scan cost
   - Only beneficial for datasets >10,000 vectors
3. **Concurrent performance dramatically improves**:
   - Sequential scan: 383ms median (concurrent)
   - HNSW m=16: 52ms median (concurrent)
   - 7.4x latency improvement under load

10.6 Architecture Recommendation

Based on these benchmarks, the system architecture includes a hybrid approach:

- **Current (1K vectors)**: In-memory vector store with file-backed cache
- **Scale Path (10K+ vectors)**: PostgreSQL + pgvector with HNSW m=16
- **Response Caching**: PostgreSQL for exact and semantic cache hits

The in-memory approach remains optimal for the current corpus size, providing 15ms search latency versus 48ms with pgvector. However, pgvector provides a clear migration path when the corpus exceeds memory constraints.

11 Corpus and Knowledge Base

11.1 Document Processing

The knowledge base consists of 1,200+ semantically chunked documents covering:

- VA disability rating criteria
- Claims filing procedures
- Appeals process
- Presumptive conditions
- Secondary conditions
- Effective dates and back pay

11.2 Chunk Metadata

Each document chunk includes:

```
{
  "entry_id": "unique-chunk-id",
  "topic": "Main Topic",
  "subtopic": "Specific Subtopic",
  "content": "Chunk text content...",
  "url": "https://veteransbenefitskb.com/...",
  "type": "policy|definition|rating_table"
}
```

Listing 11: Chunk Schema

12 Security and Deployment

12.1 Security Measures

- **API Key Management**: Environment variable storage
- **TLS Encryption**: All API calls use HTTPS
- **No PII Storage**: No personal information cached
- **Rate Limiting**: Protection against abuse
- **Admin Authentication**: Token-protected admin dashboard

12.2 Deployment Architecture

- **Platform**: Render.com (Web Service)
- **Runtime**: Python 3.11 with Gunicorn
- **Database**: PostgreSQL for analytics and response caching
- **Frontend**: React SPA served from Flask
- **Auto-Deploy**: GitHub integration for CI/CD

13 Conclusion

The Veterans Benefits AI RAG system represents a sophisticated yet maintainable architecture that prioritizes accuracy for veterans. The integration of **HyDE (Hypothetical Document Embeddings)** significantly improves retrieval quality for short and vague queries, while the comprehensive 7-layer hallucination prevention system, combined with intelligent model routing and a knowledge graph for entity-aware caching, delivers 98% citation accuracy.

Key achievements:

- **HyDE retrieval enhancement**: 15-48% better scores for short/vague queries
- **98% citation accuracy** with grounded responses
- **1.5% hallucination rate** (down from 12%) with 7-layer prevention system
- **0.8% fabricated numbers** (down from 6%) via number verification
- **Zero invalid citations in cache** through post-processing cleanup
- **40% cost reduction** through intelligent model routing (with HyDE overhead)
- **60% cache hit rate** using three-level cache with knowledge graph (L3)
- **Entity-aware lookups** matching diagnostic codes, VA forms, and benefits
- **Zero external dependencies** (no Pinecone, Redis, Elasticsearch)
- **User transparency** with confidence warnings for low-score retrievals
- **Diagnostic reports** generated for corpus improvement prioritization

13.1 HyDE + 7-Layer Defense Summary

The system now implements a two-phase approach:

Phase 1: HyDE Retrieval Enhancement

- Query preprocessing expands abbreviations (PTSD, SMC, TDIU)
- HyDE generates a hypothetical expert answer ($\sim$200 tokens)
- Hypothetical document is embedded instead of the raw query
- Result: 15-48% better retrieval scores for difficult queries

Phase 2: 7-Layer Hallucination Prevention

1. **Layer 1**: Relevance threshold filters low-quality chunks
2. **Layer 1b**: Weak retrieval refusal prevents LLM hallucination (zero tokens)
3. **Layer 2**: System prompt grounding rules
4. **Layer 3**: URL whitelist validation (zero tokens)
5. **Layer 4**: Citation verification flags suspicious claims (zero tokens)
6. **Layer 5**: Number/percentage verification detects fabricated statistics (zero tokens)
7. **Layer 6**: Citation cleanup strips invalid references before caching (zero tokens)

8. **Layer 7**: Confidence warning prefix for user transparency (∼25 tokens)

The knowledge graph architecture enables smarter cache lookups by linking questions to topics, sources, and entities (diagnostic codes, VA forms). This not only improves cache hit rates but also reduces hallucinations by ensuring cached responses are contextually relevant to the query's specific entities.

The combination of HyDE for improved retrieval and zero-token post-processing (Layers 3-6) demonstrates that effective RAG systems can achieve both high quality and cost efficiency. HyDE adds only ∼$0.00016 per query while dramatically improving retrieval for the most challenging queries.

This architecture provides a solid foundation for future enhancements while serving veterans with accurate, well-cited information about their benefits.