**Claude Insurance RAG System - Complete Documentation**

**Project Overview**

This project is a **Retrieval-Augmented Generation (RAG) system** specifically designed for insurance document analysis and question-answering. It combines semantic search capabilities with Claude AI's natural language understanding to provide accurate, context-aware responses about insurance policies, forms, and coverage details.

**Key Features**

* **Semantic Search**: Uses sentence transformers to find relevant insurance document chunks
* **Claude AI Integration**: Leverages Claude 3.5 Sonnet for intelligent response generation
* **Form-Specific Queries**: Special handling for insurance form codes (e.g., BM\_00\_20\_07\_01)
* **Web Interface**: Flask-based chat interface for user interactions
* **Smart Boosting**: Keyword-based relevance boosting for form-specific searches
* **Comprehensive Error Handling**: Robust error management and fallback mechanisms

**System Architecture**

┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐

│ User Query │───▶│ Flask Web App │───▶│ RAG System │

└─────────────────┘ └─────────────────┘ └─────────────────┘

│ │

│ ▼

┌─────────────────┐ ┌─────────────────┐

│ Web Interface │ │ Semantic Search │

│ (HTML/JS) │ │ + Embeddings │

└─────────────────┘ └─────────────────┘

│

▼

┌─────────────────┐

│ Claude API │

│ (Response Gen) │

└─────────────────┘

**File Structure and Components**

**1. app.py - Main Application File**

**Purpose**: The core Flask application that orchestrates the entire RAG system.

**Key Components:**

**ClaudeInsuranceRAG Class**

The heart of the system that handles:

* **Initialization**: Loads embeddings, metadata, and configuration
* **Query Processing**: Analyzes and categorizes user queries
* **Semantic Search**: Finds relevant document chunks using embeddings
* **Response Generation**: Uses Claude API to generate contextual responses

**Key Methods:**

**\_\_init\_\_(self, embeddings\_dir="embeddings", model\_name="all-MiniLM-L6-v2")**

* Initializes the RAG system
* Loads pre-computed embeddings and metadata
* Sets up the sentence transformer model
* Initializes Claude API client

**\_detect\_query\_type(self, query)**

* Analyzes query to determine type (form\_specific, coverage\_inquiry, claims\_process, etc.)
* Uses regex patterns to detect insurance form codes
* Returns structured query information for specialized handling

**\_search\_relevant\_chunks(self, query, top\_k=15)**

* Encodes the query using sentence transformers
* Calculates cosine similarity with all document embeddings
* Applies keyword boosting for form-specific queries
* Returns top-k most relevant chunks with metadata

**\_boost\_form\_specific\_chunks(self, chunks, form\_code, boost\_factor=0.5)**

* Enhances relevance scores for chunks from specific insurance forms
* Handles various form code formats and variations
* Improves accuracy for form-specific queries

**\_generate\_claude\_response(self, query, context\_chunks, query\_info)**

* Prepares context from relevant chunks
* Creates specialized prompts based on query type
* Makes API calls to Claude 3.5 Sonnet
* Returns structured response with metadata

**answer\_question(self, query)**

* Main orchestration method
* Combines all components to process queries end-to-end
* Includes timing, confidence scoring, and error handling

**Flask Routes:**

**/ (GET)** - Serves the main chat interface **/chat (POST)** - Handles user messages and returns AI responses **/health (GET)** - System health check endpoint **/stats (GET)** - Returns system statistics and configuration **/test-specific-form/<form\_code> (GET)** - Debug endpoint for form testing **/debug-form/<form\_code> (GET)** - Debug endpoint for search analysis **/simple-test (GET)** - Basic connectivity test **/api-test (POST)** - Direct Claude API test without RAG

**Required Data Files**

The system expects several pre-processed files in the embeddings/ directory:

**1. embeddings.npy**

**Purpose**: NumPy array containing pre-computed embeddings for all document chunks. **Format**: 2D array where each row represents a chunk's embedding vector **Creation**: Generated during preprocessing using sentence transformers

**2. chunks\_metadata.pkl**

**Purpose**: Metadata for each document chunk **Content**: List of dictionaries containing:

python

{

'document\_name': 'BM\_00\_20\_07\_01.pdf',

'chunk\_index': 0,

'page\_number': 1,

'section': 'Coverage Details',

*# ... other metadata*

}

**3. chunks.pkl**

**Purpose**: The actual text content of each document chunk **Format**: List of strings, where each string is a chunk of text **Alignment**: Indices must align with embeddings and metadata

**4. config.pkl**

**Purpose**: System configuration and preprocessing parameters **Content**: Dictionary containing:

python

{

'model\_name': 'all-MiniLM-L6-v2',

'chunking\_method': 'sentence\_based',

'sentences\_per\_chunk': 3,

'overlap\_sentences': 1,

'total\_chunks': 1500,

'total\_documents': 45

}

**Web Interface (Assumed)**

**templates/insurance\_chat.html**

**Purpose**: Frontend chat interface for user interactions **Features** (based on the Flask route):

* Chat input and message display
* Real-time communication with Flask backend
* Display of sources and confidence scores
* Responsive design for various devices

**Key JavaScript Functions** (assumed):

* sendMessage() - Sends user queries to /chat endpoint
* displayResponse() - Shows AI responses with formatting
* showSources() - Displays source documents
* handleErrors() - Manages error states and messages

**Data Preprocessing Pipeline**

While not included in the provided code, the system requires a preprocessing pipeline that:

**1. Document Ingestion**

* Reads insurance documents (PDFs, Word docs, etc.)
* Extracts text content while preserving structure
* Handles various document formats and layouts

**2. Text Chunking**

* Splits documents into manageable chunks
* Maintains context and coherence
* Creates overlapping chunks to preserve continuity

**3. Embedding Generation**

* Uses sentence transformers to create vector embeddings
* Processes all chunks through the same model
* Stores embeddings in efficient NumPy format

**4. Metadata Creation**

* Extracts document metadata (names, pages, sections)
* Creates index mappings between chunks, embeddings, and metadata
* Stores configuration for reproducibility

**Environment Setup**

**Required Environment Variables**

bash

ANTHROPIC\_API\_KEY=sk-ant-your-api-key-here

**Dependencies**

python

flask==2.3.3

flask-cors==4.0.0

python-dotenv==1.0.0

numpy==1.24.3

sentence-transformers==2.2.2

anthropic==0.25.0

scikit-learn==1.3.0

**Installation Steps**

1. Clone the repository
2. Install dependencies: pip install -r requirements.txt
3. Set up environment variables in .env file
4. Ensure preprocessed data files are in embeddings/ directory
5. Run the application: python app.py

**Query Processing Flow**

**1. Query Reception**

* User submits question through web interface
* Flask receives POST request at /chat endpoint
* Basic validation and sanitization

**2. Query Analysis**

* \_detect\_query\_type() analyzes the query
* Identifies form codes using regex patterns
* Categorizes query type for specialized handling

**3. Semantic Search**

* Query encoded using sentence transformer
* Cosine similarity calculated against all embeddings
* Top-k most relevant chunks retrieved

**4. Relevance Boosting**

* Form-specific queries get keyword boosting
* Exact form code matches receive score increases
* Results re-ranked based on boosted scores

**5. Context Preparation**

* Top chunks formatted for Claude consumption
* Source information preserved for citations
* Context length optimized for API limits

**6. Response Generation**

* Specialized prompts created based on query type
* Claude API called with context and query
* Response processed and structured

**7. Result Delivery**

* Response formatted with metadata
* Confidence scores calculated
* Sources and timing information included
* JSON response sent to frontend

**Key Features and Innovations**

**1. Smart Form Recognition**

* Regex patterns detect insurance form codes (e.g., BM\_00\_20\_07\_01)
* Multiple format variations handled automatically
* Specialized processing for form-specific queries

**2. Keyword Boosting Algorithm**

* Semantic similarity enhanced with keyword matching
* Form-specific chunks receive relevance boosts
* Improves accuracy for specific document searches

**3. Query Type Classification**

* Automatic categorization of query intent
* Specialized prompts for different query types
* Optimized processing paths for common patterns

**4. Robust Error Handling**

* Multiple fallback mechanisms for file loading
* API key validation and recovery
* Graceful degradation when components fail

**5. Performance Optimization**

* Reduced context chunks for faster responses
* Configurable top-k parameters
* Efficient numpy operations for similarity calculations

**Configuration and Customization**

**Embedding Model Selection**

The system uses all-MiniLM-L6-v2 by default, but supports any sentence-transformer model:

python

rag\_system = ClaudeInsuranceRAG(model\_name="your-preferred-model")

**Search Parameters**

Adjust relevance and performance:

python

*# In \_search\_relevant\_chunks method*

top\_k = 15 *# Number of chunks to retrieve*

boost\_factor = 0.5 *# Keyword boosting strength*

**Claude API Settings**

Configure model and response parameters:

python

response = client.messages.create(

model="claude-3-5-sonnet-20241022",

max\_tokens=800,

temperature=0.1,

*# ... other parameters*

)

**Monitoring and Debugging**

**Debug Endpoints**

* /debug-form/<form\_code> - Analyze form-specific searches
* /test-specific-form/<form\_code> - Test with manual chunk selection
* /simple-test - Basic system connectivity
* /api-test - Direct Claude API testing

**Logging**

Comprehensive logging throughout the system:

* Query processing steps
* Search result analysis
* API call monitoring
* Error tracking and debugging

**Health Monitoring**

* /health endpoint for system status
* /stats endpoint for system metrics
* Response time tracking
* Confidence score monitoring

**Security Considerations**

**API Key Management**

* Environment variable storage
* No hardcoded credentials
* Validation and error handling

**Input Validation**

* Query sanitization
* Request size limits
* Error message sanitization

**CORS Configuration**

* Controlled cross-origin access
* Appropriate security headers

**Deployment Considerations**

**Development**

bash

python app.py

*# Runs on localhost:5000 with debug mode*

**Production**

* Remove debug mode
* Use production WSGI server (e.g., Gunicorn)
* Set appropriate environment variables
* Implement proper logging and monitoring

**Scaling**

* Consider caching frequently accessed chunks
* Implement request rate limiting
* Monitor API usage and costs
* Consider embedding model caching

**Future Enhancements**

**Potential Improvements**

1. **Caching Layer**: Redis for frequently accessed results
2. **Advanced Chunking**: Semantic-aware document splitting
3. **Multi-modal Support**: Handle images and tables in documents
4. **User Sessions**: Maintain conversation context
5. **Analytics Dashboard**: Usage patterns and performance metrics
6. **A/B Testing**: Compare different models and configurations

**Scalability Options**

1. **Microservices Architecture**: Separate embedding and generation services
2. **Vector Database**: Replace NumPy arrays with specialized vector DB
3. **Load Balancing**: Distribute requests across multiple instances
4. **Async Processing**: Handle concurrent requests efficiently

**Troubleshooting Guide**

**Common Issues**

**"API key not found"**

* Check environment variables
* Verify .env file configuration
* Ensure API key format is correct

**"Embeddings not found"**

* Verify embeddings/ directory exists
* Check file naming (with/without extensions)
* Ensure preprocessing completed successfully

**"No relevant chunks found"**

* Check query formatting
* Verify document content alignment
* Review embedding quality and coverage

**"Claude API errors"**

* Verify API key validity and credits
* Check request format and parameters
* Monitor rate limits and usage

**Performance Issues**

* Reduce top\_k parameter for faster searches
* Optimize chunk size and overlap
* Consider model size vs. accuracy tradeoffs
* Monitor memory usage with large document sets

**Conclusion**

This Claude Insurance RAG system represents a sophisticated approach to domain-specific question answering, combining the power of semantic search with advanced language models. The modular architecture allows for easy customization and scaling, while the robust error handling ensures reliable operation in production environments.

The system's particular strength lies in its specialized handling of insurance documents and forms, making it an ideal solution for insurance companies, brokers, and professionals who need quick, accurate access to policy information and coverage details.