# RAG Chatbot with Google Gemini

## Complete Project Documentation

---

## Table of Contents

1. Project Overview
2. System Architecture
3. System Requirements
4. Installation Guide
5. Configuration
6. Project Structure
7. API Documentation
8. Database Schema
9. Core Components
10. Usage Guide
11. Error Handling
12. Performance Optimization
13. Troubleshooting
14. Contributing

---

## Project Overview

This project presents a sophisticated Retrieval-Augmented Generation (RAG) chatbot that leverages Google's Gemini AI models to provide intelligent document-based conversations. Built with modern technologies including LangChain, FastAPI, and Streamlit, the system enables users to upload various document formats and engage in meaningful conversations about their content.

### What This System Does

Our RAG chatbot transforms how users interact with their documents. Instead of manually searching through lengthy PDFs or Word documents, users can simply ask questions in natural language and receive accurate, context-aware responses. The system processes uploaded documents, creates searchable embeddings, and maintains conversation history for a seamless user experience.

### Key Capabilities

**Document Processing Excellence**: The system supports multiple file formats including PDF, DOCX, and HTML files. Each document is intelligently processed, chunked into manageable segments, and indexed for optimal retrieval performance.

**Advanced AI Integration**: We've integrated multiple Google Gemini models (2.0 Flash, 1.5 Flash, and 1.5 Pro) to provide users with flexibility in choosing the most appropriate model for their specific needs.

**Persistent Memory**: Unlike basic chatbots, our system maintains conversation history across sessions, allowing for contextual follow-up questions and continuing discussions where you left off.

**Comprehensive Management**: Users can easily upload new documents, view their document library, and remove files they no longer need, all through an intuitive web interface.

### Technology Foundation

Our technology stack represents a careful balance of performance, scalability, and user experience:

- **Frontend**: Streamlit provides an intuitive, responsive web interface
- **Backend**: FastAPI delivers high-performance API services with automatic documentation
- **AI Engine**: Google Gemini models power our natural language understanding and generation
- **Vector Database**: ChromaDB enables efficient semantic search across document collections
- **Data Storage**: SQLite manages conversation history and document metadata
- **Document Processing**: LangChain handles complex document parsing and chain management

---

## System Architecture

### Architectural Overview

Our system follows a modern, microservices-inspired architecture that separates concerns while maintaining efficient communication between components.

```plaintext
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Streamlit     │    │    FastAPI      │    │  Google Gemini  │
│   Frontend      │◄──►│    Backend      │◄──►│      API        │
└─────────────────┘    └─────────────────┘    └─────────────────┘
         │                       │
         │                       ▼
         │              ┌─────────────────┐
         │              │     SQLite      │
         │              │   (Chat Logs)   │
         │              └─────────────────┘
         │                       │
         ▼                       ▼
┌─────────────────┐    ┌─────────────────┐
│    Session      │    │    ChromaDB     │
│   Management    │    │ (Vector Store)  │
└─────────────────┘    └─────────────────┘
```

### How Data Flows Through the System

**Document Upload Process**: When users upload a document, the FastAPI backend receives the file, processes it through LangChain's document loaders, splits it into semantic chunks, generates embeddings using Google's embedding models, and stores both the embeddings and metadata in ChromaDB.

**Query Processing**: User questions trigger a sophisticated retrieval process where the system searches ChromaDB for relevant document chunks, combines them with conversation history, and sends the enriched context to Google Gemini for response generation.

**Session Management**: Every interaction is tracked and stored in SQLite, enabling the system to maintain context across multiple exchanges and provide personalized experiences.

---

## System Requirements

### Hardware Specifications

For optimal performance, we recommend the following hardware configuration:

**Memory Requirements**: A minimum of 4GB RAM is required, though 8GB or more is strongly recommended for handling multiple documents and maintaining smooth performance during concurrent operations.

**Storage Considerations**: Ensure at least 2GB of free disk space for system dependencies, document storage, and database files. Additional space will be needed based on the size and number of documents you plan to process.

**Processing Power**: While the system can run on basic hardware, a multi-core processor significantly improves document processing speed and overall responsiveness.

### Software Prerequisites

**Python Environment**: Python 3.8 or newer is required. We recommend using Python 3.9 or 3.10 for the best compatibility with all dependencies.

**Operating System**: The system is designed to work across platforms including Windows, macOS, and Linux distributions.

**Network Access**: A stable internet connection is essential for communicating with Google's Gemini API services.

---

## Installation Guide

### Step 1: Repository Setup

Begin by cloning the project repository to your local machine:

bash

```bash
git clone <repository-url>
cd rag-chatbot-gemini
```

### Step 2: Environment Preparation

Create an isolated Python environment to avoid dependency conflicts:

bash

```bash
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

### Step 3: Dependency Installation

Install all required packages using the provided requirements file:

bash

```bash
pip install -r requirements.txt
```

### Step 4: Environment Configuration

Create a `.env` file in your project root directory and add your Google API credentials:

plaintext

```plaintext
GOOGLE_API_KEY=your_gemini_api_key_here
```

**Important**: Obtain your API key from the Google AI Studio platform and ensure it has access to Gemini models.

### Step 5: Database Initialization

The SQLite database will be created automatically when you first run the application. No manual database setup is required.

---

## Configuration

### Environment Variables

The system uses environment variables for secure configuration management:

| Variable Name | Purpose | Required | Default Value |
|---|---|---|---|
| `GOOGLE_API_KEY` | Authentication key for Google Gemini API | Yes | None |

### Model Selection

Choose from three available Gemini models based on your needs:

**Gemini 2.0 Flash Experimental** (Default): The latest model offering improved performance and capabilities, ideal for most use cases.

**Gemini 1.5 Flash**: A balanced option providing good performance with faster response times, suitable for real-time conversations.

**Gemini 1.5 Pro**: The most capable model for complex reasoning tasks, recommended for advanced document analysis.

### Database Configuration

**SQLite Database**: The system automatically creates `rag_app.db` in your project directory to store conversation history and document metadata.

**ChromaDB Vector Store**: Document embeddings are stored in the `./chroma_db/` directory, which is created automatically during first use.

---

## Project Structure

Understanding the project structure helps with maintenance and customization:

```plaintext
rag-chatbot-gemini/
├── .env                    # Environment variables (create this)
├── requirements.txt        # Python dependencies
├── rag_app.db             # SQLite database (auto-generated)
├── chroma_db/             # ChromaDB vector store (auto-generated)
├── app.log                # Application logs
├── main.py                # FastAPI application entry point
├── streamlit_app.py       # Streamlit frontend entry point
├── langchain_utils.py     # RAG and LangChain logic
├── chroma_utils.py        # Vector database operations
├── db_utils.py            # SQLite database operations
├── pydantic_models.py     # Data models and validation schemas
├── api_utils.py           # API client utilities
├── chat_interface.py      # Streamlit chat interface components
└── sidebar.py             # Streamlit sidebar functionality
```

Each file serves a specific purpose in the application architecture, promoting maintainability and code organization.

---

## API Documentation

### Base Configuration

All API endpoints are available at:

```
http://localhost:8000
```

The FastAPI framework automatically generates interactive API documentation accessible at `http://localhost:8000/docs`.

## Core Endpoints

### Chat Interaction Endpoint

**POST** `/chat`

This endpoint processes user queries and returns AI-generated responses based on uploaded documents.

**Request Format**:

```json
{
  "question": "What are the main findings in the research document?",
  "session_id": "optional-session-identifier",
  "model": "gemini-2.0-flash-exp"
}
```

**Response Format**:

```json
{
  "answer": "Based on the research document, the main findings include...",
  "session_id": "session-identifier",
  "model": "gemini-2.0-flash-exp"
}
```

### Document Upload Endpoint

**POST** `/upload-doc`

Upload and index documents for retrieval-augmented generation.

**Request**: Multipart form data containing the document file **Supported Formats**: PDF, DOCX, HTML files **File Size Limit**: Recommended under 10MB for optimal performance

**Response**:

```json
{
  "message": "File research_paper.pdf has been successfully uploaded and indexed.",
  "file_id": 42
}
```

### Document Management Endpoints

**GET** `/list-docs`

Retrieve information about all uploaded documents.

**Response**:

```json
[
  {
    "id": 1,
    "filename": "research_paper.pdf",
    "upload_timestamp": "2024-01-15T10:30:00"
  },
  {
    "id": 2,
    "filename": "user_manual.docx",
    "upload_timestamp": "2024-01-15T11:45:00"
  }
]
```

**POST** `/delete-doc`

Remove a specific document from the system.

**Request**:

```json
{
  "file_id": 42
}
```

**Response**:

```json
{
  "message": "Successfully deleted document with file_id 42 from the system."
}
```

---

## Database Schema

### SQLite Database Structure

#### Application Logs Table

This table maintains a complete history of all chat interactions:

| Column Name | Data Type | Description |
|---|---|---|
| id | INTEGER PRIMARY KEY | Unique identifier for each interaction |
| session_id | TEXT | Groups related conversations together |
| user_query | TEXT | The user's original question |
| gpt_response | TEXT | The AI-generated response |
| model | TEXT | Which Gemini model was used |
| created_at | TIMESTAMP | When the interaction occurred |

#### Document Store Table

This table tracks all uploaded documents:

| Column Name | Data Type | Description |
|---|---|---|
| id | INTEGER PRIMARY KEY | Unique identifier for each document |
| filename | TEXT | Original filename as uploaded |
| upload_timestamp | TIMESTAMP | When the document was uploaded |

### ChromaDB Vector Store

**Collection Structure**: Documents are stored in ChromaDB collections with rich metadata including file identifiers, source information, and page numbers for PDFs.

**Embedding Model**: The system uses Google's `models/embedding-001` for generating high-quality vector representations of document content.

## Core Components

### LangChain Integration (langchain_utils.py)

This module serves as the brain of our RAG system, orchestrating the complex process of retrieval-augmented generation.

**Primary Function**: The `get_rag_chain(model)` function creates a sophisticated processing pipeline that combines document retrieval with conversation history to generate contextually aware responses.

**Key Features**: The system implements history-aware retrieval, meaning it considers previous conversation turns when searching for relevant document chunks, and maintains context across multiple exchanges for more natural conversations.

### Vector Database Management (chroma_utils.py)

ChromaDB operations are centralized in this module to ensure consistent handling of document embeddings.

**Document Processing**: The `load_and_split_document()` function intelligently processes different file types and splits them into optimal chunks of 1000 characters with 200-character overlap for maximum retrieval effectiveness.

**Indexing Operations**: Documents are indexed with comprehensive metadata, allowing for precise retrieval and management operations.

**Cleanup Capabilities**: The system can cleanly remove documents from the vector store while maintaining database integrity.

### Database Operations (db_utils.py)

All SQLite interactions are managed through this module, providing a clean abstraction layer for data persistence.

**Conversation Management**: Functions handle the storage and retrieval of chat history, enabling session continuity and conversation context.

**Document Tracking**: The system maintains detailed records of uploaded files, including timestamps and metadata for efficient management.

### API Backend (main.py)

The FastAPI application provides a robust, scalable backend with comprehensive error handling and logging.

**Security Features**: CORS middleware enables secure cross-origin requests while maintaining appropriate security boundaries.

**File Handling**: Sophisticated upload processing with validation, temporary storage, and cleanup procedures.

**Session Management**: Automatic session generation and tracking for seamless user experiences.

### Frontend Interface (streamlit_app.py)

The Streamlit application provides an intuitive, responsive interface for end users.

**State Management**: Sophisticated session state handling ensures consistent user experiences across page refreshes and interactions.

**Component Integration**: Seamless integration between sidebar controls and main chat interface for optimal usability.

---

## Usage Guide

### Getting Started

#### Starting the Backend Service

Launch the FastAPI backend server with the following command:

```bash
uvicorn main:app --reload --host 0.0.0.0 --port 8000
```

The `--reload` flag enables automatic reloading during development, while the host and port configuration makes the service accessible from other machines if needed.

#### Launching the Frontend

In a separate terminal, start the Streamlit interface:

```bash
streamlit run streamlit_app.py
```

### Accessing Your Application

Open your web browser and navigate to `http://localhost:8501` to access the full application interface.

## Document Management Workflow

### Uploading Documents

Navigate to the sidebar and use the file uploader component. The system accepts PDF files (including complex multi-page documents), Microsoft Word documents (.docx format), and HTML files with preserved formatting.

After selecting your file, click the "Upload" button. The system will process your document, create searchable embeddings, and confirm successful indexing.

### Managing Your Document Library

Use the "Refresh Document List" button to view all uploaded files with their upload timestamps. This feature helps you keep track of your document collection and manage storage efficiently.

To remove documents you no longer need, select the file from the dropdown menu and click "Delete Selected Document". This action removes both the file record and its associated embeddings.

## Interactive Chat Experience

### Selecting the Right Model

Choose from the available Gemini models based on your specific needs. For general document Q&A, Gemini 2.0 Flash offers excellent performance. For complex analysis requiring deeper reasoning, consider Gemini 1.5 Pro.

### Crafting Effective Queries

Ask specific, detailed questions to get the most accurate responses. Instead of "What is this document about?", try "What are the key recommendations in section 3 of the policy document?"

Reference specific sections, page numbers, or concepts when you want targeted information from particular parts of your documents.

### Understanding Responses

AI responses include expandable details showing which document sections were used to generate the answer, helping you verify information and explore source material further.

The conversation maintains context automatically, so you can ask follow-up questions without repeating background information.

### Best Practices for Optimal Performance

#### Document Optimization

Keep individual files under 10MB for the best processing speed and performance. Ensure your documents contain clear, readable text rather than scanned images when possible.

Use native file formats (actual PDFs rather than scanned documents, original Word files rather than converted formats) for the most accurate text extraction and processing.

#### Query Optimization

Be specific in your questions to receive more targeted and useful responses. Provide context when asking about technical terms or concepts that might have multiple meanings.

Use the conversation history feature by building on previous questions rather than starting fresh each time.

---

## Error Handling

### Understanding Common Issues

#### File Upload Problems

**Unsupported File Types**: The system currently supports PDF, DOCX, and HTML files. If you encounter an error, verify your file format matches one of these supported types.

**File Size Limitations**: Large files may cause processing timeouts. If you're having trouble with a large document, consider splitting it into smaller sections.

**File Corruption**: Damaged or corrupted files will produce processing errors. Try re-saving your document in its native format before uploading.

**System Response Example**:

```json
{
  "status_code": 400,
  "detail": "Unsupported file type. Allowed types are: .pdf, .docx, .html"
}
```

**API Communication Issues**

**Authentication Problems**: Verify that your Google API key is correctly set in the `.env` file and has the necessary permissions for Gemini model access.

**Rate Limiting**: Google's API services have usage limits. If you encounter rate limiting errors, wait a few minutes before making additional requests.

**Network Connectivity**: Ensure stable internet connectivity, as the system requires communication with Google's services for AI processing.

**Database and Storage Issues**

**Permission Problems**: Ensure the application has write permissions in its directory for creating and modifying database files.

**Disk Space**: Monitor available storage space, as vector embeddings and chat history can accumulate over time.

**Database Integrity**: If you encounter database errors, the system includes automatic recovery mechanisms, but you may need to restart the application in severe cases.

## Logging and Monitoring

The application maintains detailed logs in the `app.log` file, recording all significant events including user interactions, errors, and system status information.

**Log Format Example**:

```
INFO:root:Session ID: abc123, User Query: What are the main topics?, Model: gemini-2.0-flash-exp
INFO:root:Session ID: abc123, AI Response: The main topics include...
ERROR:root:Failed to process document: Invalid file format
```

## Performance Optimization

### Document Processing Efficiency

#### Intelligent Chunking Strategy

Our system uses a carefully tuned chunking strategy with 1000-character segments and 200-character overlap. This configuration balances retrieval accuracy with processing efficiency, ensuring that related information stays together while maintaining searchability.

#### Batch Processing Capabilities

When uploading multiple documents, the system can process them efficiently through parallel operations, reducing overall processing time and improving user experience.

#### Embedding Cache Management

Frequently accessed document sections are cached in memory to reduce API calls and improve response times for common queries.

### Vector Search Optimization

#### Search Parameter Tuning

The system retrieves the top 2 most relevant document chunks for each query (`k=2`), providing a balance between comprehensive context and response speed.

#### Index Maintenance

ChromaDB automatically optimizes its indexes, but you can improve performance by periodically restarting the application to clear temporary data and refresh connections.

#### Memory Management

Monitor your system's memory usage, especially when working with large document collections. The vector database loads embeddings into memory for fast access.

### API Performance Enhancement

#### Connection Management

The system maintains persistent connections to both the SQLite database and ChromaDB to minimize connection overhead and improve response times.

#### Asynchronous Operations

FastAPI's asynchronous capabilities ensure that long-running operations don't block other requests, maintaining system responsiveness during document processing.

#### Response Optimization

Similar queries are temporarily cached to reduce API calls to Google's services while maintaining response freshness.

### Scalability Considerations

#### Horizontal Scaling Options

For high-traffic scenarios, you can deploy multiple FastAPI instances behind a load balancer, with shared access to the same database and vector store.

#### Vertical Scaling Strategies

Increase system RAM to handle larger document collections and more concurrent users. Consider SSD storage for faster database operations and vector store access.

#### Resource Monitoring

Monitor CPU usage during document processing and memory consumption during vector operations to identify potential bottlenecks before they impact performance.

---

## Troubleshooting

### Application Startup Issues

#### Import and Module Errors

If you encounter missing module errors, verify that your virtual environment is activated and all dependencies are properly installed:

bash

```bash
python --version  # Verify Python version
pip list  # Check installed packages
pip install -r requirements.txt --force-reinstall  # Reinstall if needed
```

#### Port Conflicts

If the application fails to start due to port conflicts, check which processes are using the required ports:

```bash
netstat -an | grep 8000  # Check if port 8000 is in use
netstat -an | grep 8501  # Check if port 8501 is in use
```

You can modify the port numbers in the startup commands if conflicts exist.

## Document Processing Problems

### Upload Success Without Indexing

If documents upload but don't appear in search results, check the ChromaDB directory permissions and ensure the vector database is properly initialized:

```bash
ls -la chroma_db/  # Check directory permissions
```

If problems persist, you can reset the vector database:

```bash
rm -rf chroma_db/  # Remove existing vector database
# Restart the application to reinitialize
```

### Text Extraction Issues

Some PDF files may have text extraction problems due to formatting or encoding issues. Try converting problematic files to a different format or re-saving them before uploading.

## API and Connectivity Issues

### Google Gemini API Problems

Verify your API key configuration and test connectivity:

```bash
echo $GOOGLE_API_KEY  # Verify the key is set
# Test API connectivity (requires curl)
curl -H "Authorization: Bearer $GOOGLE_API_KEY" \
  https://generativelanguage.googleapis.com/v1/models
```

### Rate Limiting Solutions

If you encounter rate limiting, implement these strategies:

- Reduce the frequency of requests
- Consider upgrading your Google API quota
- Implement exponential backoff in your request patterns

## Chat and Session Issues

### Conversation History Problems

If chat history isn't persisting properly, check:

- SQLite database file permissions
- Available disk space for database growth
- Application logs for session-related errors

### Session State Management

Streamlit session state issues can often be resolved by:

- Refreshing the browser page
- Clearing browser cache
- Restarting the Streamlit application

## Advanced Debugging

### Enable Debug Logging

For detailed troubleshooting information, enable debug-level logging by modifying the `main.py` file:

```python
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)
```

#### Database Inspection

You can directly inspect the SQLite database using database browser tools or command-line sqlite3 to understand data storage issues:

```bash
sqlite3 rag_app.db
.tables  # List all tables
.schema application_logs  # View table structure
SELECT * FROM application_logs LIMIT 5;  # Sample data
```

---

## Contributing

### Development Environment Setup

#### Getting Started with Development

Fork the repository on your preferred platform and create a local development environment:

```bash
git clone <your-fork-url>
cd rag-chatbot-gemini
git checkout -b feature/your-new-feature
```

#### Development Workflow

Create a dedicated virtual environment for development to avoid conflicts with other projects:

```bash
python -m venv dev-env
source dev-env/bin/activate  # On Windows: dev-env\Scripts\activate
pip install -r requirements.txt
pip install -r requirements-dev.txt  # If development dependencies exist
```

## Code Quality Standards

### Python Style Guidelines

We follow PEP 8 conventions for Python code formatting and organization. Use tools like `black` or `autopep8` to ensure consistent formatting:

```bash
pip install black
black .  # Format all Python files
```

### Type Annotations

Include type hints in function signatures and variable declarations to improve code readability and enable better IDE support:

```python
def process_document(file_path: str, file_id: int) -> Dict[str, Any]:
    """Process a document and return metadata."""
    pass
```

### Documentation Standards

Add comprehensive docstrings to all functions and classes using the following format:

python

```python
def example_function(param1: str, param2: int) -> bool:
    """
    Brief description of the function.

    Args:
        param1: Description of the first parameter
        param2: Description of the second parameter

    Returns:
        Description of what the function returns

    Raises:
        ValueError: Description of when this exception is raised
    """
    pass
```

## Testing Guidelines

### Unit Test Development

Create comprehensive unit tests for new functionality. Place test files in a `tests/` directory with descriptive names:

python

```python
# tests/test_chroma_utils.py
import unittest
from chroma_utils import load_and_split_document

class TestChromaUtils(unittest.TestCase):
    def test_document_splitting(self):
        """Test that documents are split correctly."""
        # Test implementation
        pass
```

### Integration Testing

Develop integration tests that verify the interaction between different system components, particularly API endpoints and database operations.

## Submission Process

### Pull Request Guidelines

Before submitting a pull request:

1. Ensure all tests pass locally
2. Update documentation for any new features
3. Add or update type hints as appropriate
4. Follow the established code formatting standards

### Pull Request Description

Include a comprehensive description of your changes:

- What problem does this solve?
- How have you tested the changes?
- Are there any breaking changes?
- What documentation updates are included?

## Feature Development Guidelines

### Planning New Features

Before implementing significant new features, consider:

- How does this fit with the existing architecture?
- What are the performance implications?
- How will this affect the user experience?
- What additional dependencies might be required?

### Maintaining Backward Compatibility

When modifying existing functionality, ensure that changes don't break existing user workflows or API contracts. If breaking changes are necessary, provide clear migration guidance.

---

This comprehensive documentation provides everything needed to understand, deploy, and contribute to the RAG Chatbot system. The combination of powerful AI capabilities, robust architecture, and user-friendly interface makes this an excellent foundation for document-based AI applications.

For additional support or questions not covered in this documentation, please refer to the project's issue tracker or contribute to the community discussions.