RAG Application | AI Document Chat Assistant - Project Summary

A **Retrieval-Augmented Generation (RAG)** application that enables users to upload documents and ask intelligent questions about their content. The system combines document processing, vector similarity search, and local AI to provide accurate, context-aware responses based on uploaded materials.

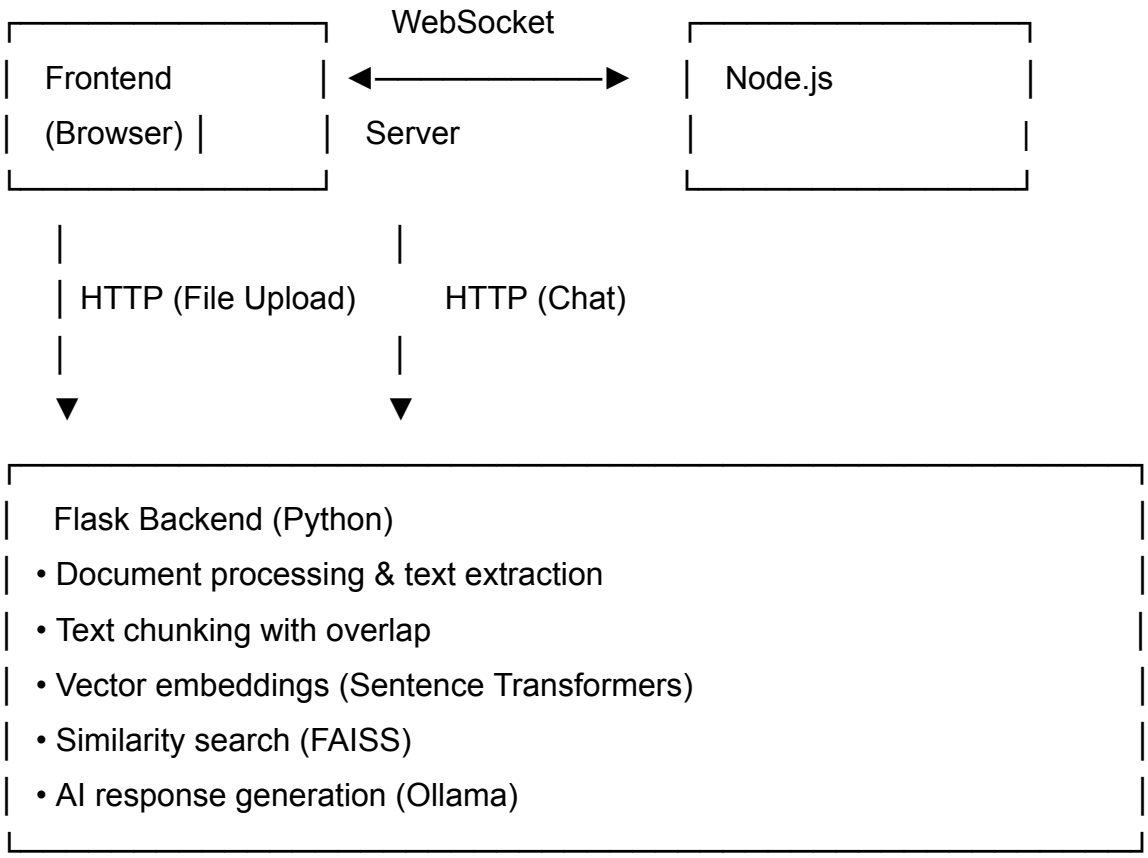
Project Overview

This application demonstrates full-stack development skills, AI/ML integration, and practical implementation of modern RAG architecture. Users can upload documents (PDF, DOCX, TXT), which are automatically processed, chunked, embedded, and indexed for semantic search. Questions asked through the chat interface retrieve relevant context from documents before generating AI responses.

Key Capabilities

- Document upload and processing (multiple formats)
- Semantic search using vector embeddings
- **Selective file querying -** Choose which documents to search per question
- Context-aware AI responses powered by local LLM with source attribution
- Real-time chat interface with WebSocket
- Document management with interactive file selection UI

SYSTEM ARCHITECTURE



TECHNOLOGY STACK

Frontend

- HTML5, CSS3, JavaScript (Vanilla) - Basic
- WebSocket API for real-time communication
- Drag-and-drop file upload
- Responsive design

Backend

Node.js: + Express - WebSocket server for real-time messaging

Flask: (Python) - REST API for document processing and AI

FAISS: - Facebook AI Similarity Search for vector operations

Sentence Transformers: - Text embedding generation (all-MiniLM-L6-v2)

Ollama: - Local LLM runtime (phi4-mini:3.8b)

Document Processing

PyPDF2:

- PDF text extraction

python-docx:

- Microsoft Word document parsing
- Plain text file support
- Smart chunking algorithm with overlap

Data Persistence

- FAISS index files for vector storage
- Pickle serialization for metadata
- SQLite for conversation history

KEY FEATURES IMPLEMENTED

1. Document Processing Pipeline

Multi-format support: PDF, DOCX, TXT files

Smart chunking: 500-character chunks with 50-character overlap for context preservation

Automatic indexing: Documents are immediately available for querying after upload

File listing API: Retrieve all uploaded files with chunk counts

2. Retrieval-Augmented Generation (RAG)

Semantic search: Uses cosine similarity to find relevant document chunks

Selective filtering: Query specific documents by filename selection

Context injection: Top 3 most relevant chunks provided to LLM with source files

Source attribution: Responses show which file each context chunk came from

Fallback handling: Gracefully handles queries without relevant context

3. User Interface

Real-time chat: WebSocket-based instant messaging

Interactive file list: Checkbox-based document selection with chunk counts

Select all/none buttons: Quick file selection controls

File management: Upload, view statistics, clear documents

Visual feedback: Loading states, upload progress, success/error notifications

Statistics dashboard: Real-time chunk count and collection status

Skills Demonstrated & Technical Skills

Full-Stack Development: Python, JavaScript, Node.js

AI/ML Integration: Embeddings, vector search, LLM orchestration

API Design: RESTful APIs, WebSocket communication

Database Management: Vector databases, SQLite

Problem Solving: Debug complex multiprocessing issues

System Architecture: Design scalable RAG pipeline

Software Engineering Practices

Clean Code: Modular design, separation of concerns

Documentation: Comprehensive technical documentation

Error Handling: Graceful failures and user feedback

Version Control: Git with organized commit history

Configuration Management: Environment variables, .gitignore

Domain Knowledge

Natural Language Processing: Text chunking, embeddings, semantic search

RAG Architecture: Context retrieval and augmentation

LLM Integration: Prompt engineering, context management

Vector Operations: Similarity search, indexing strategies