Software Requirements Specification

for

SmartDocQ

Prepared by –

B.ANIRUDH: 23BD1A0521

G.SRITHI: 23BD1A05AK

K.SAMEEKSHA: 23BD1A05AR

K.ANANYA: 23BD1A05AX

T.SAI SHASHANK REDDY: 23BD1A05DT

KMIT

28-07-2025

Table of Contents

Revision History

Name	Date	Reason For Changes	Version
SmartDocQ	04-08-2025	Updated Appendix and Non-functional Requirements	1.0
SmartDocQ	08-08-2025	Did necessary changes and added Diagrams & Mock Screens.	1.1
SmartDocQ	13-08-2025	Did necessary changes and added Tech Stack.	1.2

1. Introduction

1.1 Purpose

The purpose of SmartDocQ is to simplify and enhance the way users interact with large volumes of unstructured text by providing an intelligent, document-based Question and Answer system.

Traditional document reading and information extraction methods are time-consuming, especially when dealing with long reports, academic material, legal documents, or research papers. SmartDocQ addresses this challenge by enabling users to ask natural language questions and receive accurate answers directly from the contents of uploaded documents.

This system is being developed to:

- Reduce manual effort in scanning through large documents.
- Provide instant, context-aware answers based on document content.
- Improve productivity for students, researchers, educators, and professionals.
- Make document interaction intuitive and efficient, even for non-technical users.
- Ultimately, the purpose is to create a smarter, faster, and more accessible way to extract meaningful information from various types of textual documents.

1.2 Document Conventions

This Software Requirements Specification (SRS) document follows a consistent structure and formatting style for better readability and interpretation. The following conventions are applied throughout:

- **Font Sizes Used**:
    - **Title:** 16px
    - **Headings (e.g., 1.1, 2.3):** 14px
    - **Body text/content:** 12px
- **Text Styles**:
    - **Bold** text is used for module names, components, and key terms (e.g., **SmartDocQ**, **LLM**, **UI**).
    - Monospaced font is used for file types, code snippets, or system inputs (e.g., PDF, DOCX, POST /api/query).

  - The entire document uses Times New Roman font.

- **Acronyms & Abbreviations**:
  - **RAG** – Retrieval-Augmented Generation
  - **LLM** – Large Language Model
  - **UI** – User Interface
  - **API** – Application Programming Interface
- **Numbering Scheme**:
  - Section numbering follows **IEEE 830** SRS standards for easy navigation and reference (e.g., 1.2, 3.4.2).

1.3 Intended Audience and Reading Suggestions

Intended Audience:

- **Developers & Designers** – To understand the functional and non-functional requirements, system architecture, and design constraints for implementation.
- **Testers & QA Engineers** – To refer to expected behavior, performance, and edge cases for validation and verification purposes.
- **Future Maintainers or Contributors** – To get a complete understanding of the system's structure, goals, and logic for future upgrades or maintenance.
- **End Users (e.g., Students, Researchers)** – For understanding how the system will address their needs in document-based Q&A tasks.

Reading Suggestions:

- Begin with **Section 1** for the background and purpose of the system.
- **Section 2** provides a high-level overview of the system's functionality, environment, and users.
- **Sections 3 and 4** are critical for developers and testers.
- **Section 5** highlights non-functional aspects like performance and security.
- **Appendices** contain glossary terms, models, and any pending items.
- This structure ensures that every reader finds the most relevant information quickly and comprehensively.

1.4 Product Scope

SmartDocQ is an AI-powered document question-answering system designed to help users retrieve information efficiently from various types of documents. The product aims to simplify information extraction by enabling users to upload files in formats such as PDF, DOCX, TXT, web pages, and even handwritten notes (via OCR).

The system breaks down documents into meaningful chunks, transforms them into semantic embeddings, and stores them in a vector database to facilitate fast, context-aware retrieval. When a user poses a question, SmartDocQ leverages a **Retrieval-Augmented Generation (RAG)** pipeline powered by **Google's Gemini model** to generate accurate and source-grounded answers.

Additional capabilities include:

- **Session-based and persistent memory** to personalize interactions over time.
- **Feedback integration**, enabling continuous improvement based on user corrections.

The scope of this product covers document ingestion, semantic processing, vector-based retrieval, AI-powered Q&A, memory handling, and user feedback loops. It is designed for use in educational, research, and professional environments where fast, intelligent access to information is essential.

1.5 References

- Gemini API Documentation (Google AI): https://ai.google.dev/gemini
- React Official Documentation: https://react.dev
- IEEE Documentation :IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998)
- **Interactive ChatBot for PDF Content Conversation Using an LLM Language Model**
- LangChain Documentation: https://docs.langchain.com
- LottieFiles, Lottie Animations Documentation, Available: https://lottiefiles.com

2. Overall Description

2.1 Product Perspective

SmartDocQ is a standalone, AI-powered document question-answering system developed to revolutionize how users interact with unstructured textual data. Unlike traditional keyword-based search tools, SmartDocQ utilizes **semantic understanding** through advanced technologies such as **Google Gemini**, **Retrieval-Augmented Generation (RAG)**, and **vector embeddings** to retrieve and present meaningful information.

This system operates independently and does not rely on third-party search engines or document management platforms. Instead, it acts as an intelligent layer over the user's document repository, enabling direct interaction with document content in a conversational manner.

By focusing on context and semantic relevance rather than mere keyword matching, SmartDocQ significantly improves the quality and speed of information retrieval. It is especially effective in domains where traditional search methods fall short — such as analyzing lengthy academic papers, legal records, research reports, institutional documents, or policy manuals.

SmartDocQ has been built from scratch as a next-generation utility, tailored for educational, research, and professional environments where intelligent, document-based Q&A is essential.

2.2 Product Functions

SmartDocQ offers the following core functionalities to enable intelligent, document-based interaction:

- Document Upload & Parsing

  Supports uploading documents in various formats including PDF, DOCX, TXT, and URLs. The system automatically extracts and preprocesses content for further analysis.
- Content Chunking & Embedding

  Divides documents into meaningful segments (chunks) and converts them into semantic vector embeddings using pre-trained language models.
- Vector Storage & Indexing

  Stores the generated embeddings in a scalable vector database, enabling fast and accurate semantic similarity search.
- Query-Based Retrieval

  Upon receiving a user question, SmartDocQ retrieves the most contextually relevant document chunks through semantic search mechanisms.
- Answer Generation

  Combines the retrieved content with the Gemini Large Language Model (LLM) to generate accurate,

natural-language answers via a Retrieval-Augmented Generation (RAG) pipeline.

- Conversational Memory
  Maintains both short-term memory (for current session context) and long-term memory (to personalize future interactions), enhancing the conversational flow and user experience.
- Interactive User Interface
  Offers a simple, intuitive, and responsive web interface where users can upload documents, ask questions, and receive answers in real time.

2.3 User Classes and Characteristics

The primary users of **SmartDocQ** fall into the following categories:

- **Students and Researchers**
  Require quick and accurate access to information within dense academic resources such as textbooks, research papers, and scholarly articles.
- **Teachers and Educators**
  Use the system to efficiently extract relevant content for creating lecture materials, assignments, or assessments from large volumes of educational documents.
- **Professionals (e.g., Lawyers, Analysts, Consultants)**
  Need to retrieve specific insights from extensive datasets such as legal case files, financial reports, policy documents, or technical manuals.

User Assumptions:

- Users are expected to have basic digital literacy and the ability to frame meaningful queries in natural language.
- No prior technical knowledge of artificial intelligence, machine learning, or vector databases is required.
- The interface is designed to accommodate both technical and non-technical users through intuitive interaction.

2.4 Operating Environment

The **SmartDocQ** system is designed to function within a **client-server architecture**, consisting of the following components:

- **Frontend**

  A web-based user interface built using **React**, accessible through any modern web browser such as Chrome, Firefox, or Edge. The interface allows users to upload documents, enter queries, and view responses in real time.

  Lottie animations are integrated into the frontend to enhance user engagement with smooth, lightweight, and responsive visuals.
- **Backend**

  A server-side application developed using **Python**, leveraging frameworks such as **FastAPI** or **Flask**. The backend is responsible for:
  - Document ingestion and content preprocessing
  - Chunking and semantic embedding generation
  - Query handling and vector-based retrieval
  - Answer generation via integration with the **Gemini LLM API**
- **Hosting and Execution**

  The system is designed to run on standard server hardware, whether deployed locally or on cloud infrastructure. No software installation is required for end users — only a modern web browser and internet access.

2.5 Design and Implementation Constraints

The design and implementation of SmartDocQ are subject to the following technical constraints:

- The core **answer generation** functionality is dependent on the **Google Gemini** large language model.
- The system follows the **Retrieval-Augmented Generation (RAG)** architectural pattern to combine retrieval and generative capabilities.
- A **vector database** is used for storing and querying semantic embeddings to enable fast and context-aware retrieval.
- The backend must be implemented in **Python**, using either the **FastAPI** or **Flask** web framework.
- The frontend must be implemented using either **React** or **Streamlit**, depending on the deployment preference.

2.6 User Documentation

SmartDocQ will include concise and accessible user documentation to assist users in navigating the platform. It will cover the following areas:

- **Document Uploading**: Step-by-step guidance on how to upload various document formats.
- **Query Interaction**: Instructions on how to ask effective questions based on the uploaded content.
- **Understanding Responses**: Tips for interpreting the AI-generated answers.
- **Built-in Support Tools**: The interface will include intuitive tooltips and a dedicated help page for real-time assistance.
- **User Feedback**: Users can provide feedback and suggestions to help improve the model's performance, accuracy, and overall user experience.

This documentation aims to ensure that users—regardless of technical background—can interact confidently and effectively with the system.

2.7 Assumptions and Dependencies

Assumptions:

- Users possess documents in a digital, machine-readable format (e.g., PDF, DOCX, TXT, or web links).
- Users have access to a stable internet connection to interact with the web-based system effectively.

Dependencies:

- The system relies on the continuous availability and correct functioning of the **Google Gemini API** for answer generation.
- The solution is dependent on the operational status of the chosen **vector database service** (e.g., Pinecone, Chroma, etc.) used for semantic retrieval.
- Core functionalities depend on several **Python libraries** for:
    - Document parsing (e.g., PyMuPDF, python-docx)
    - Embedding generation (e.g., transformers, sentence-transformers)
    - Backend operations (e.g., FastAPI, Flask)
- Frontend rendering depends on the selected framework (**React**) and libraries (**LottieFiles**).

Tech Stack

Programming Languages

- **Python 3.8+** — Backend development, AI model integration, text processing, API implementation.
- **JavaScript (ES6+)** — Frontend development using ReactJS for a dynamic and responsive user interface.

Backend Frameworks and Libraries

- **Flask** — Lightweight web framework to build RESTful API endpoints for backend services.
- **PyTorch** — Deep learning framework to load and run FLAN-T5 summarization and chatbot models with GPU acceleration.
- **Transformers (HuggingFace)** — For pre-trained FLAN-T5 models, summarization pipelines, and text generation.
- **LangChain and LangChain-Community** — Utilities for document loading, chunking, vector embedding generation, and retrieval augmented generation (RAG) pipeline.

- **ChromaDB** — Vector database used for similarity search on document embeddings enabling fast semantic retrieval.
- **NLTK** — Natural language toolkit for tokenization and sentence splitting.
- **CleanText** — Text cleaning and normalization.
- **SymPy** — Symbolic mathematics support for chatbot math problem fallback.
- **Requests / HTTPX** — HTTP clients to call external APIs
- **Profanity-Filter** — To moderate chatbot responses and filter inappropriate content.

Frontend Frameworks and Libraries

- **ReactJS** — Building the frontend user interface with components for document upload, summary display, chatbot interaction, and download options.
- **Additional JS libraries** — For drag-and-drop file upload, progress bars, and real-time UI updates.

Database

- **MongoDB Atlas** — Cloud-hosted NoSQL database for securely storing user credentials, JWT tokens, preferences, and summarization/chat history.

Authentication and Security

- **JSON Web Tokens (JWT)** — Secure authentication and session management.
- **Encryption** — Secure storage of sensitive user information and tokens in the database.

Development and Deployment Tools

- **Google Colab / Local GPU-enabled environment** — For training, testing, and inference with GPU acceleration.
- **Cloud Platforms (AWS, GCP, Azure) (planned)** — For hosting scalable backend services and database.

3. External Interface Requirements

3.1 User Interfaces

The UI provides the following key functionalities:

- A simple and intuitive interface for uploading one or more documents in formats such as PDF, DOCX, TXT, or web URLs.
- A text input field where users can enter natural language questions based on the uploaded document content.
- A response display area where the AI-generated answers are clearly presented.
- A conversation history section, allowing users to review previous queries and responses during an active session.
- A feedback mechanism that allows users to rate answers and suggest improvements, supporting continuous model enhancement.
- Lightweight Lottie animations used to indicate loading states or transitions without affecting performance.

3.2 Hardware Interfaces

- **Client Side:**
    - **Processor:** Minimum dual-core 2.0 GHz or higher.
    - **Memory (RAM):** Minimum 4 GB (8 GB recommended).
    - **Storage:** At least 200 MB free space for browser cache and temporary files.
    - **Display:** Minimum resolution 1366×768; responsive design supports higher resolutions.
    - **Input Devices:** Standard keyboard and mouse/touchscreen.
    - **Network Interface:** Stable internet connection (minimum 5 Mbps recommended).
    - **Browser:** Latest versions of Chrome, Firefox, Edge, or Safari with JavaScript enabled.
- **Server Side:**
    - **Processor:** Multi-core 2.4 GHz or higher (Intel Xeon or AMD EPYC recommended).
    - **Memory (RAM):** Minimum 16 GB (32 GB recommended for large document loads).
    - **Storage:** Minimum 500 GB SSD (expandable based on data volume).
    - **Network Interface:** Minimum 1 Gbps Ethernet.
    - **Operating System:** Linux (Ubuntu 22.04 LTS recommended) or equivalent server-grade OS.
- **Special Hardware Interfaces:**

None required in current scope. The system does not connect to specialized peripherals (e.g., biometric devices, scanners) directly; all document inputs are via file upload in the web application.

3.3 Software Interfaces

- **Google Gemini API**:
  The system makes secure API calls to the Gemini service by sending the user's query along with retrieved context, and parsing the returned natural-language answer. This interaction forms the core of the AI-powered answer generation using Retrieval-Augmented Generation (RAG).
- **Vector Database API**:
  The system integrates with a vector database (e.g., ChromaDB, FAISS) for storing and querying semantic embeddings of document content. This interface supports fast, similarity-based retrieval of relevant content.
- **Document Parsing Libraries**:
  Libraries such as are used to extract raw text from various document formats during the preprocessing phase.
- **Frontend Integration Layer**:
  The frontend (built using React and styled with Lottie animations) communicates with the backend (FastAPI or Flask) via RESTful APIs. This includes document uploads, question submission, answer retrieval, and session handling.
- **Authentication Interface** :
  If personalization or long-term memory is implemented, user authentication may be integrated using Firebase Auth or a custom JWT-based authentication system. This allows session tracking and personalized document interaction.

3.4 Communications Interfaces

- **Client-Server Communication**:
  The system follows a client-server architecture where the frontend communicates with the backend (FastAPI or Flask) over HTTP using secure RESTful APIs.
- **External API Communication**:
  The backend securely communicates with external services such as the Google Gemini API via HTTPS. API keys or tokens are used to authenticate requests and ensure data integrity and confidentiality.

- **Database Communication**:

  The backend interfaces with the vector database (e.g., ChromaDB, FAISS) locally or over a network through its exposed APIs or SDKs. Efficient read/write operations are essential for storing and retrieving embeddings.

4. System Features

4.1 Document Upload & Preprocessing

- Upload PDF, DOCX, TXT files via a web interface.
- Extracts and chunks content.
- Converts content into embeddings for semantic understanding.
- Stores in a vector database.
- **Priority** - High

4.2 Question Answering(RAG-based)

- User types a natural language question.
- System searches relevant document chunks.
- Uses Gemini (or another LLM) to generate accurate, grounded answers.
- **Priority** - High

4.3 Real-Time Responses

- Answers are returned instantly (or near real-time).
- Ensures fast and interactive user experience.
- **Priority** - High

4.4 Memory-Based Personalization

- Tracks previous interactions in the session.
- Helps provide context-aware follow-up answers.
- **Priority** - Medium

4.5 User Feeback Mechanism

- Users can rate answers .
- Feedback used to improve accuracy in future sessions.
- **Priority** - Medium

4.6 User History Tracking

- Stores question-answer history per user.
- Allows users to revisit previous questions or sessions.
- **Priority** - Medium

4.7 Mutli-format Support

- Supports PDF, DOCX, TXT, and possibly HTML/Web pages.
- Ensures flexibility for all kinds of users.
- **Priority** - Low

4.8 Security & Input Validation

- Validates and sanitizes all input (uploads, queries).
- Protects against XSS, injection attacks, and misuse.
- **Priority** - High

5. Other Nonfunctional Requirements

5.1 Performance Requirements

- Responses must be generated within 10–15 seconds under normal load.
- System should handle large files (up to defined limit) without major performance drop.
- Benchmarked against SQuAD 2.0 and CoQA datasets.

5.2 Safety Requirements

- Ensure user data privacy and document integrity.
- Prevent unauthorized access to uploaded documents.
- Implement robust error handling to avoid crashes.

5.3 Security Requirements

- Role-Based Access Control (RBAC) for user permissions.
- Secure storage of API keys and credentials (never in frontend).
- HTTPS for all communication.
- Input validation to prevent XSS and injection attacks.
- Session security with auto-timeout and secure cookies.

5.4 Software Quality Attributes

- Reliable, maintainable, and extensible design.
- Adaptable for future AI model or database changes.

5.5 Business Rules

No specific business rules are mentioned in the source document.

6. Other Requirements

- Must comply with IT Act 2000 and GDPR for data protection.
- Use AES-256 encryption for stored data and TLS 1.2+ for data in transit.
- Support cross-platform access (Windows, macOS, Linux, Android, iOS).

7. StakeHolders

7.1 List of Stakeholders

1. **Students / General Users :** Primary end-users who use the system to query, search, and retrieve relevant information from documents for academic, personal learning, or general reference purposes.
2. **Teachers / Professionals :** Users who utilize the platform for preparing lecture materials, summarizing large volumes of text, or conducting professional document analysis to aid decision-making.
3. **Researchers :** Users who require accurate extraction of specific insights, data points, and summaries

from academic papers, scholarly articles, and research datasets.

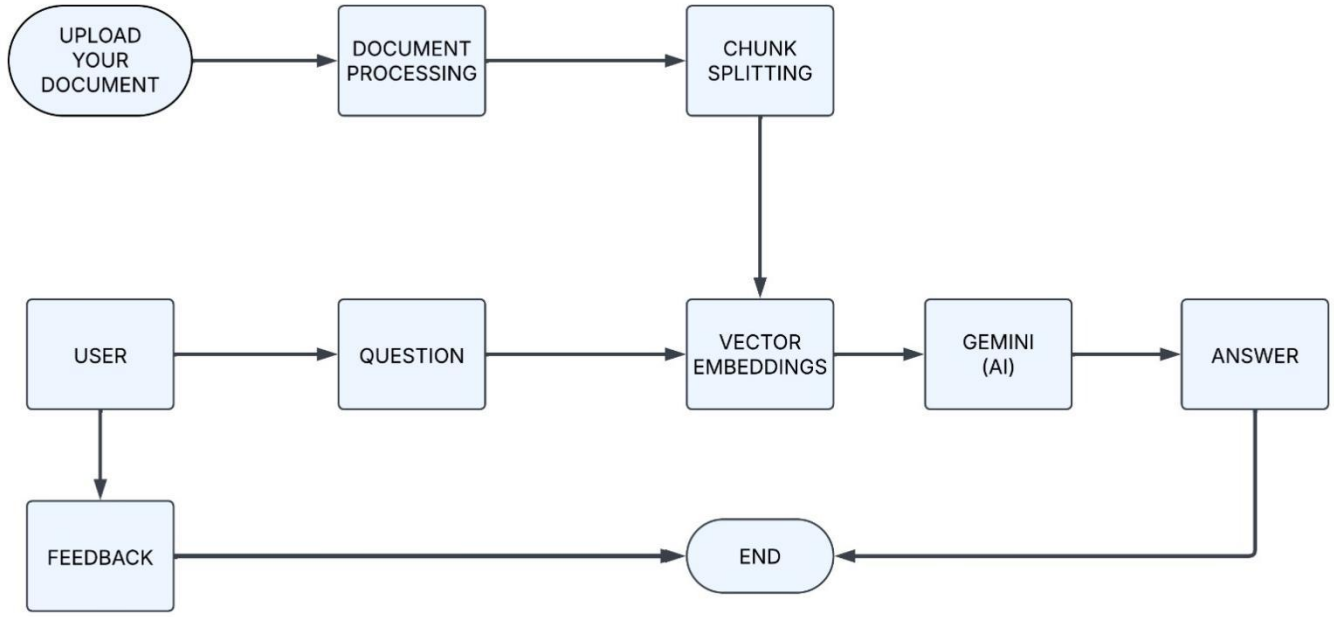
4. **Administrators :** Personnel responsible for monitoring platform performance, managing user accounts and permissions, ensuring system availability, and maintaining security compliance.
5. **Technical Support & Maintenance Team :** Engineers or support staff who handle software updates, bug fixes, and user assistance to ensure smooth platform operation.

Appendix A: Glossary

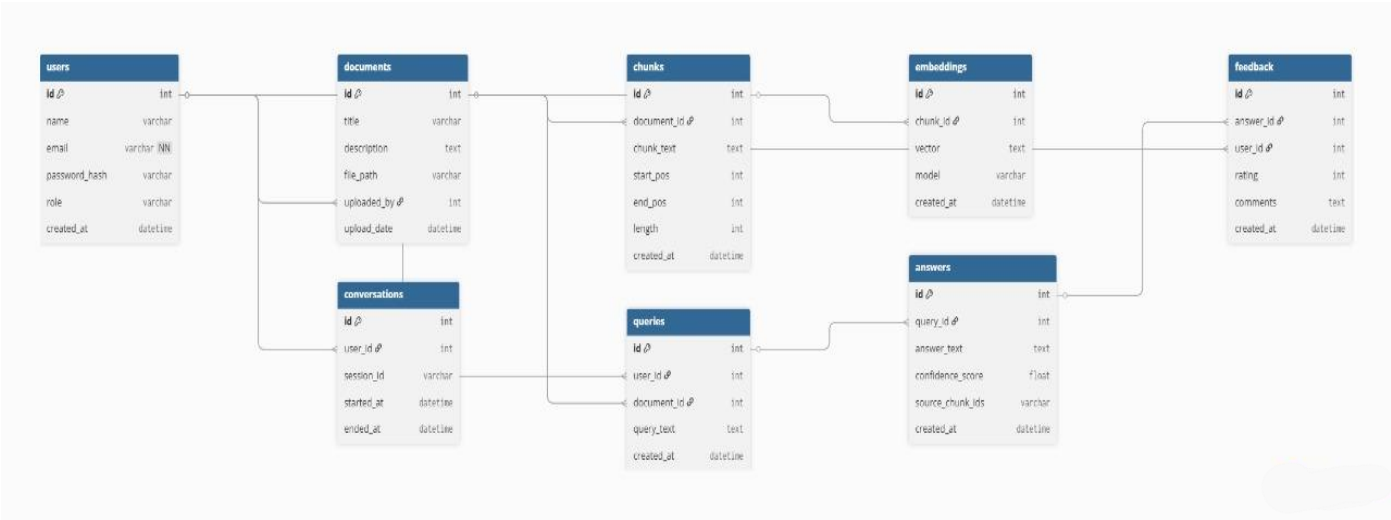
- AI (Artificial Intelligence) – Simulation of human intelligence by machines, especially for tasks like understanding language, answering questions, etc.
- CoQA (Conversational Question Answering) – A dataset used to evaluate how well systems can answer questions in a conversational flow.
- Embeddings – Numeric vector representations of text that capture semantic relationships and meanings.
- Gemini – A multimodal, large language model developed by Google, used in SmartDocQ for generating accurate, contextual answers.
- Q&A (Question and Answer) – A method of retrieving precise information in response to user queries.
- RAG (Retrieval-Augmented Generation) – An architecture that combines document retrieval with language generation for fact-grounded answers.
- SQuAD 2.0 (Stanford Question Answering Dataset) – A benchmark dataset for evaluating machine reading comprehension and answer generation.
- Vector Database – A specialized database for storing and efficiently searching embedding vectors derived from documents.

Appendix B: Analysis Models

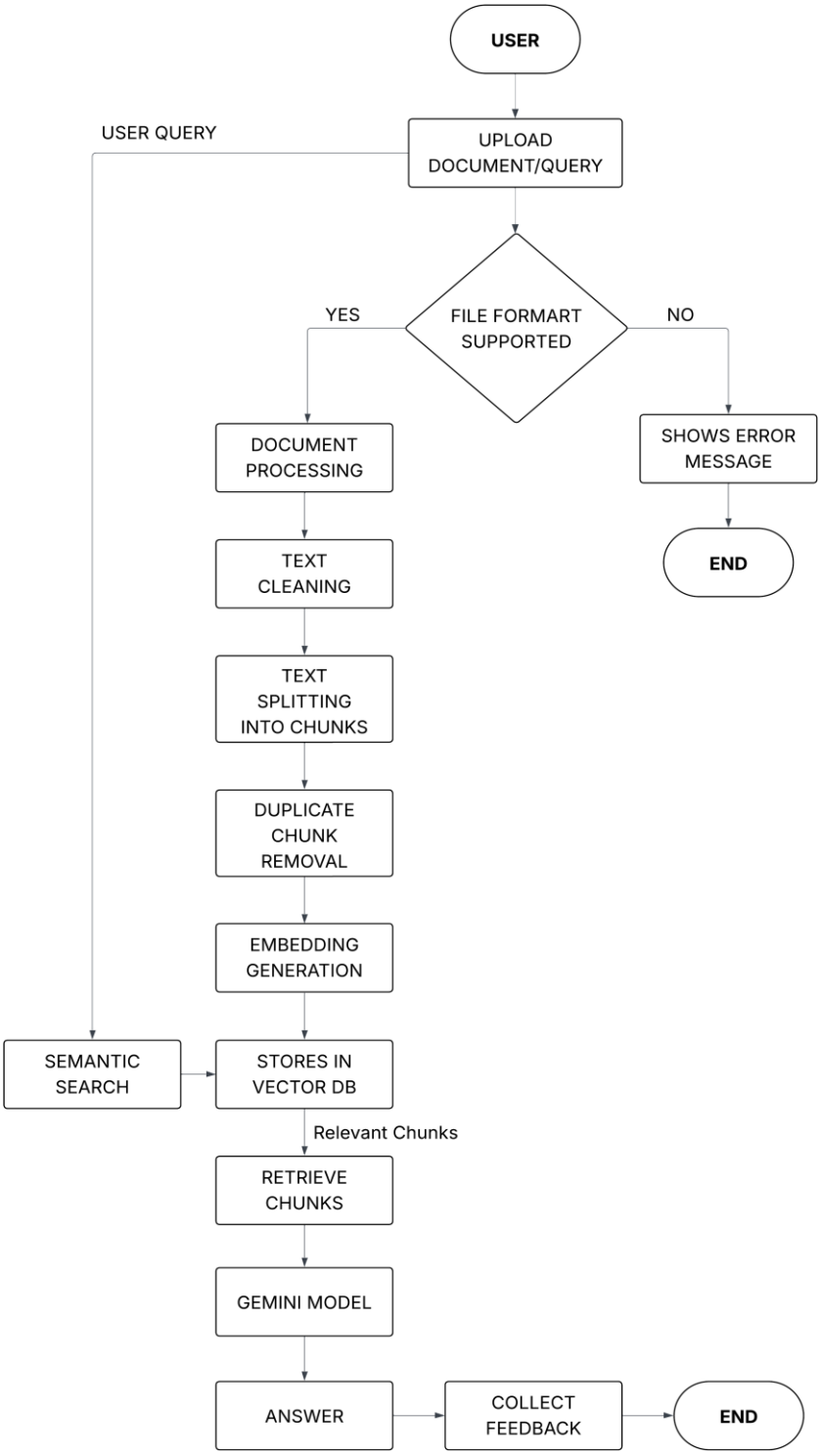
ARCHITECTURE MODEL



DB DESIGN

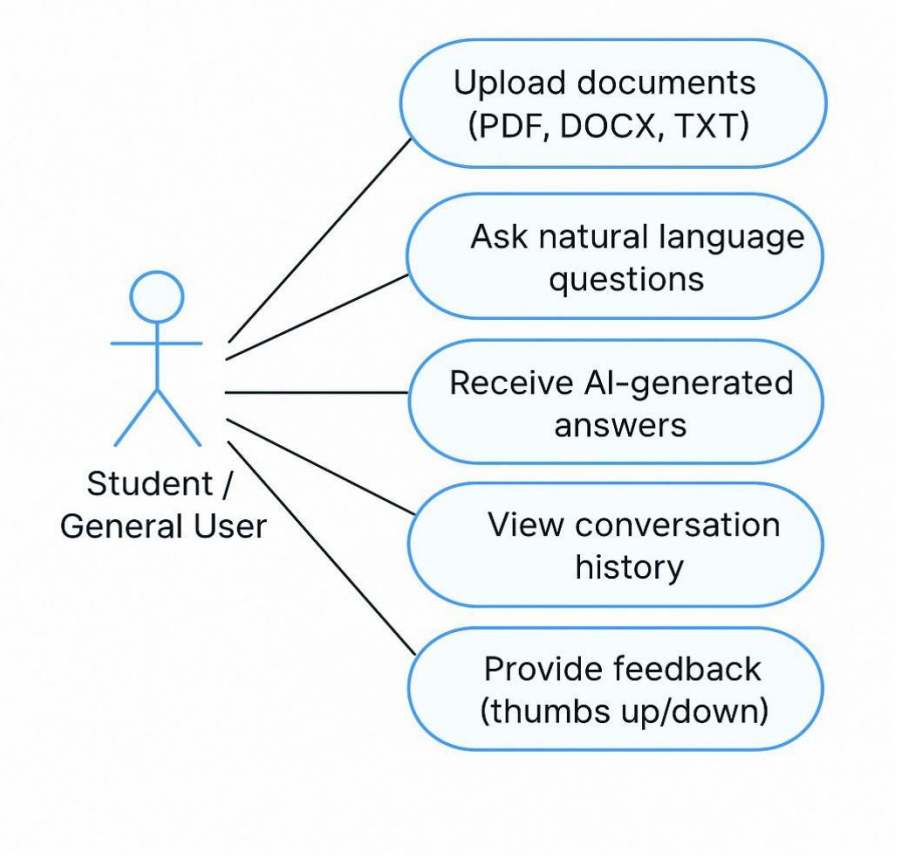


WORK FLOW

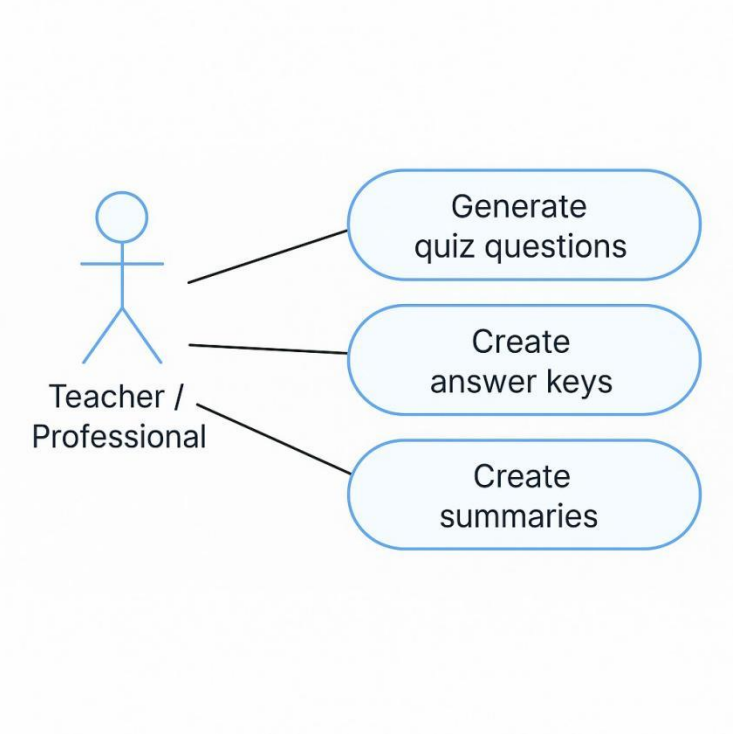

USER
USER QUERY
UPLOAD DOCUMENT/QUERY
FILE FORMART SUPPORTED
YES
NO
DOCUMENT PROCESSING
SHOWS ERROR MESSAGE
END
TEXT CLEANING
TEXT SPLITTING INTO CHUNKS
DUPLICATE CHUNK REMOVAL
EMBEDDING GENERATION
SEMANTIC SEARCH
STORES IN VECTOR DB
Relevant Chunks
RETRIEVE CHUNKS
GEMINI MODEL
ANSWER
COLLECT FEEDBACK
END

USE CASE DIAGRAMS

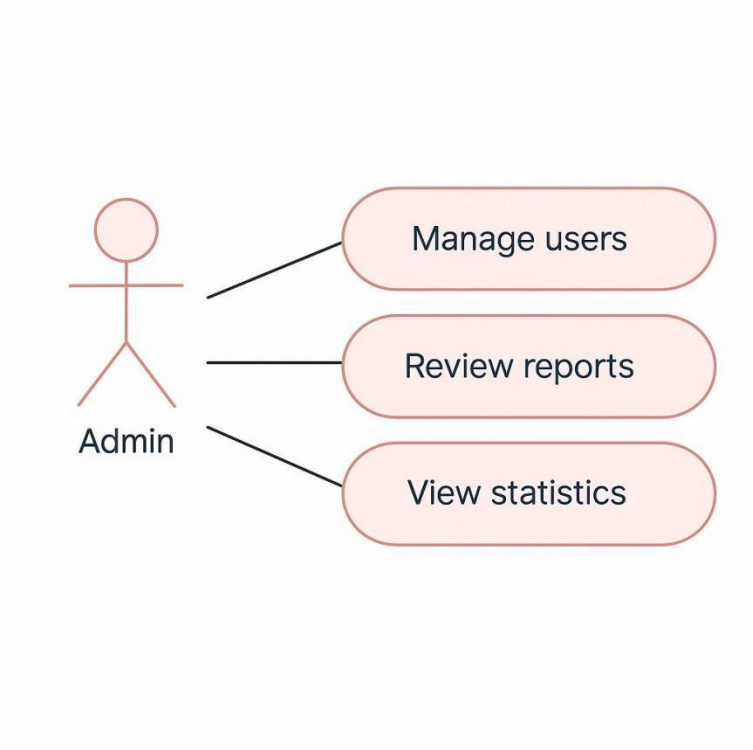
1. Student/General User



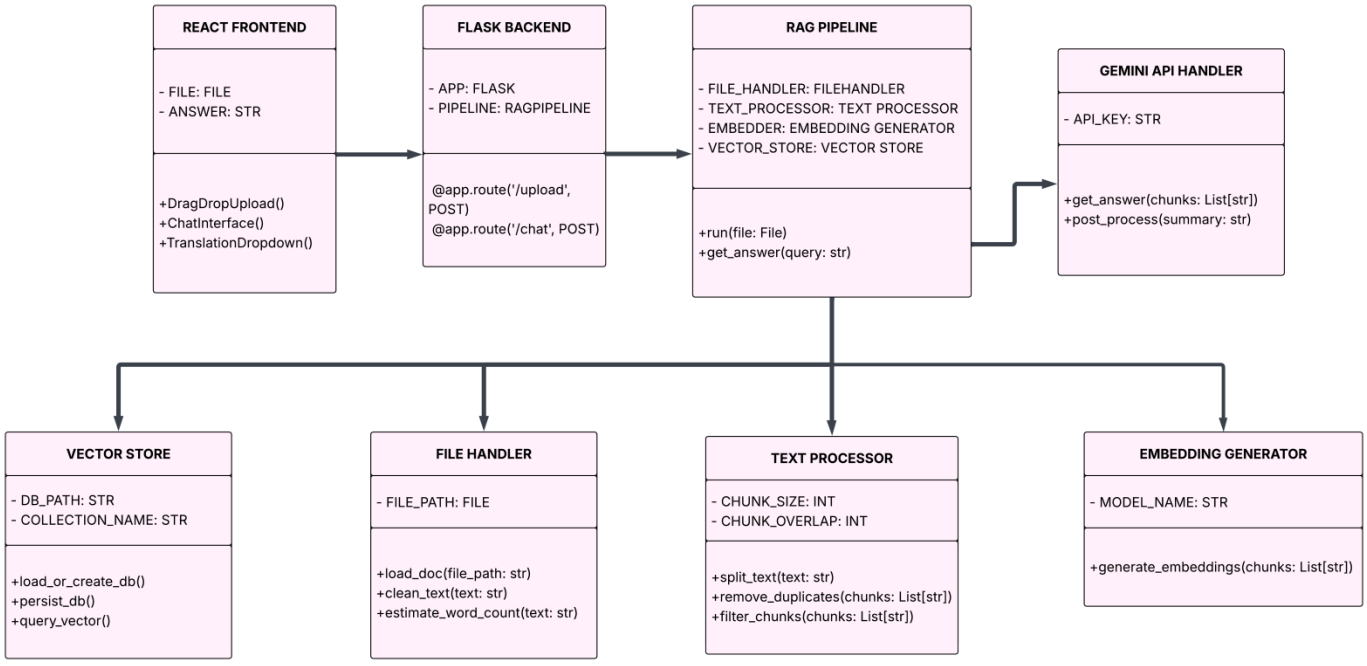
2. Teacher/Professional



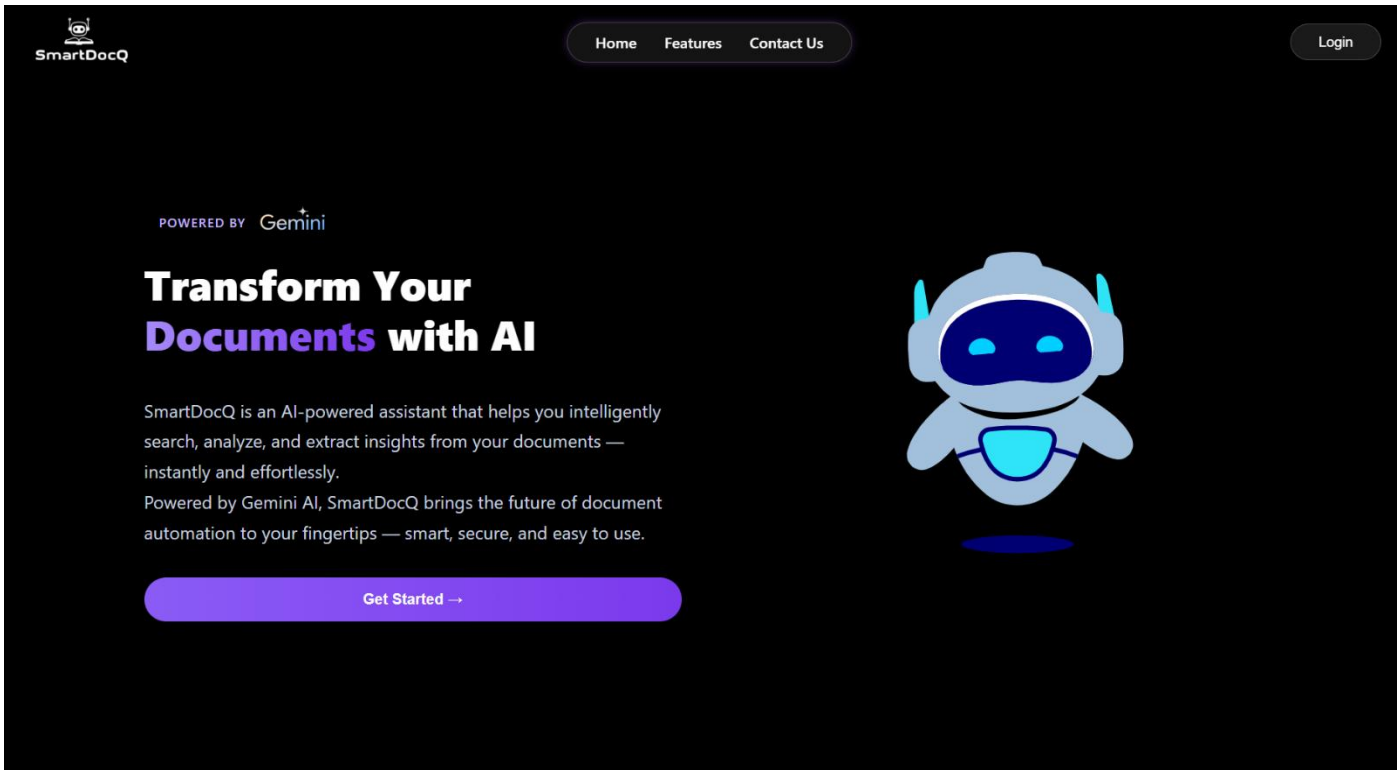
3. Admin

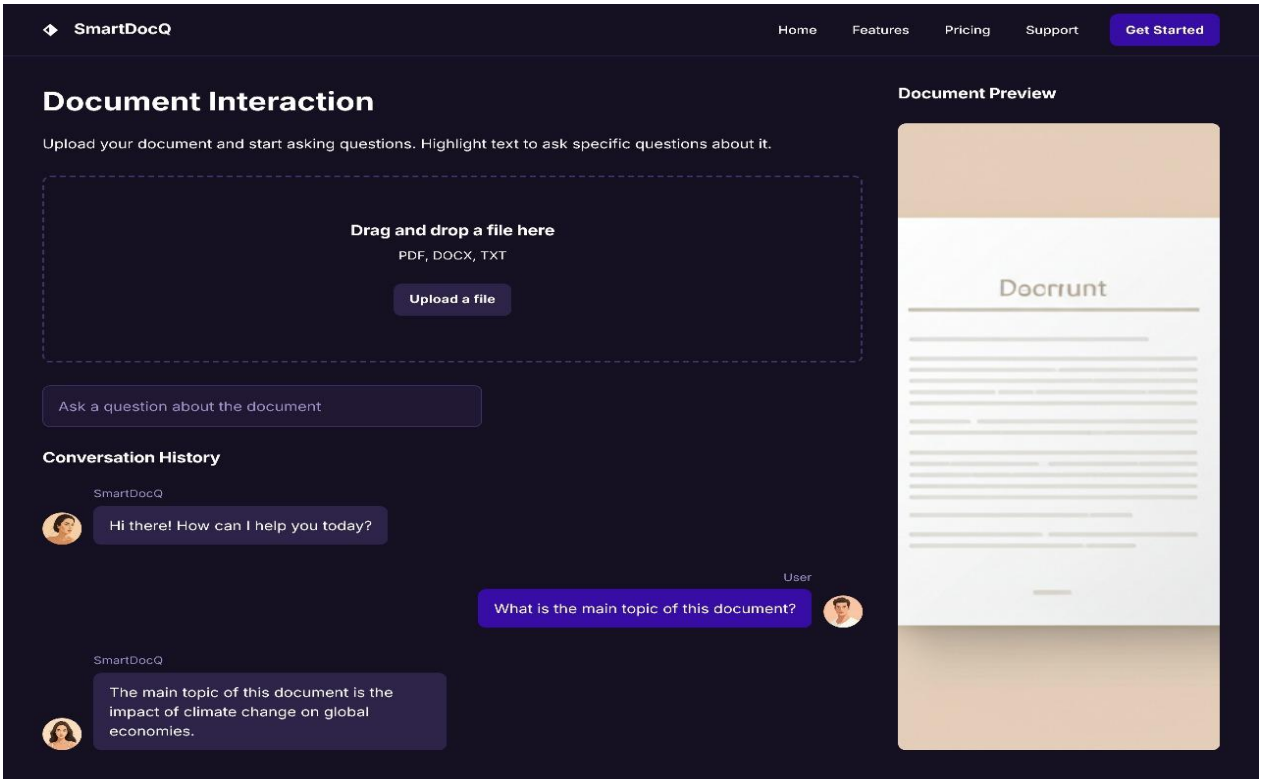


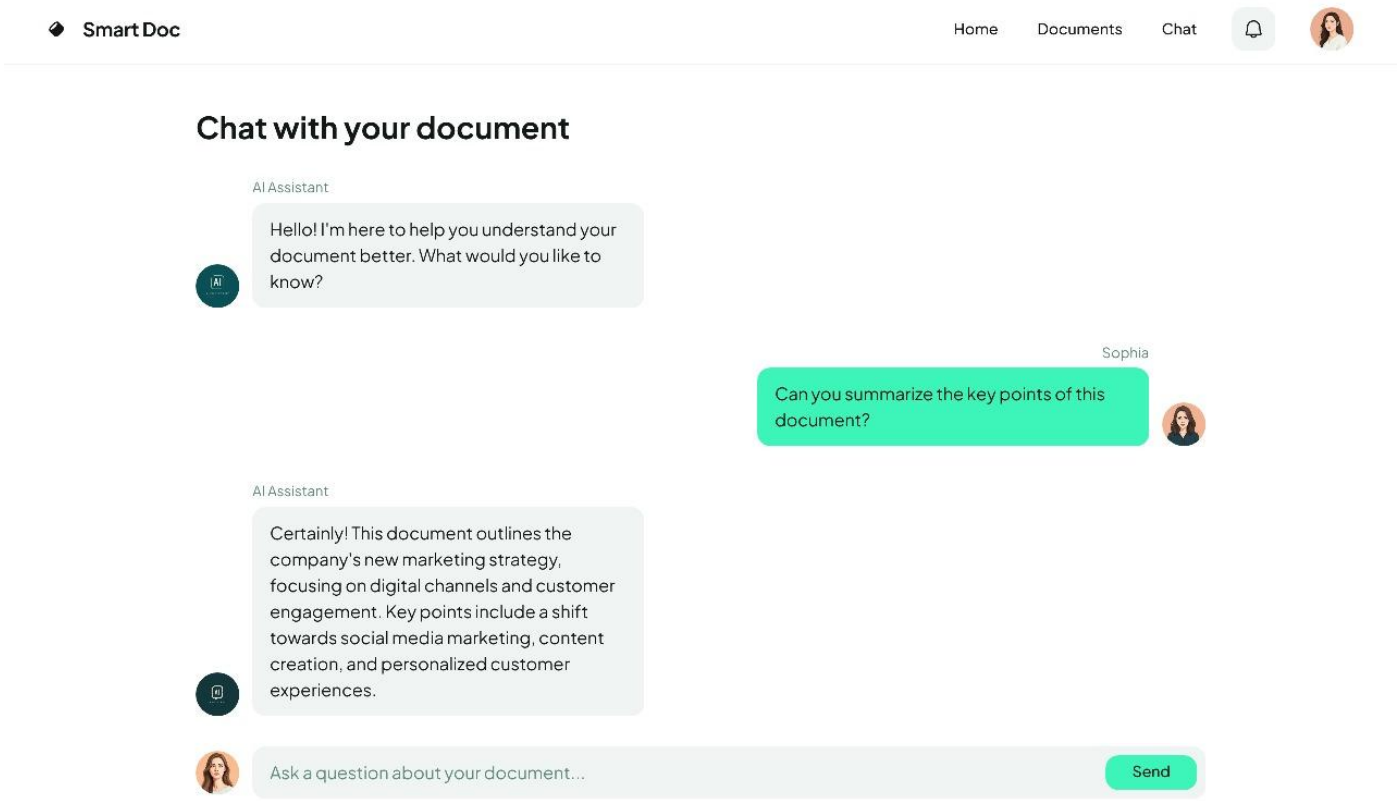
CLASS DIAGRAM

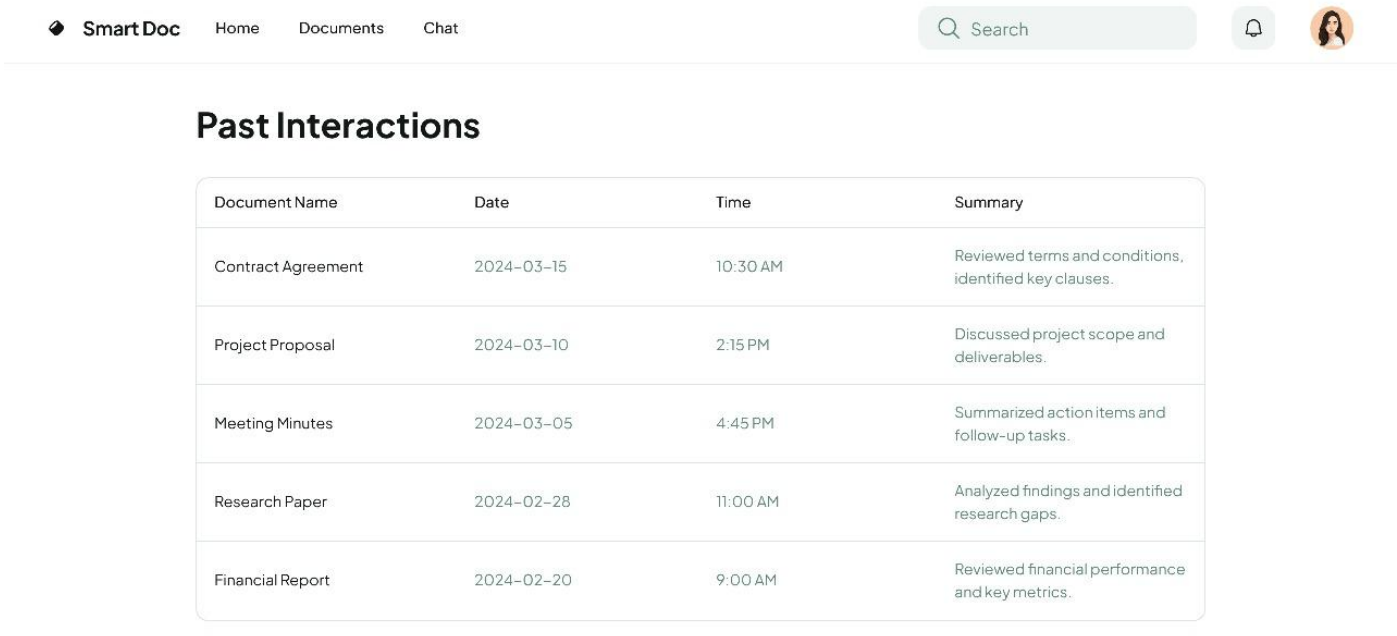


MOCK SCREENS






Smart Doc
Home
Documents
Chat
Chat with your document
AI Assistant
Hello! I'm here to help you understand your document better. What would you like to know?
Sophia
Can you summarize the key points of this document?
AI Assistant
Certainly! This document outlines the company's new marketing strategy, focusing on digital channels and customer engagement. Key points include a shift towards social media marketing, content creation, and personalized customer experiences.
Ask a question about your document...
Send


Smart Doc
Home
Documents
Chat
Search
Past Interactions
Document Name
Date
Time
Summary
Contract Agreement
2024-03-15
10:30 AM
Reviewed terms and conditions, identified key clauses.
Project Proposal
2024-03-10
2:15 PM
Discussed project scope and deliverables.
Meeting Minutes
2024-03-05
4:45 PM
Summarized action items and follow-up tasks.
Research Paper
2024-02-28
11:00 AM
Analyzed findings and identified research gaps.
Financial Report
2024-02-20
9:00 AM
Reviewed financial performance and key metrics.

Appendix C: To Be Determined List

- Final selection of the vector database technology (e.g., ChromaDB, FAISS, etc.).
- Comprehensive error handling mechanisms for API failures and backend processing issues.
- Definition of performance evaluation metrics using benchmarks like SQuAD 2.0 and CoQA.
- Detailed design and implementation of both short-term and long-term memory components.
- Strategy for secure user authentication and document-level data segregation.

Week-IV Milestone

Project Server Side coding starts - Begin development of the backend functionality: Environment Setup - Set up development environment Configure servers | Install required frameworks/libraries Database Integration. .

11.1 Backend Implementation Progress

This section provides an overview of the backend implementation progress for SmartDocQ. It outlines the features that have been completed, the modular code structure, and the underlying pipeline that powers document ingestion and retrieval-augmented generation (RAG). Key capabilities include:

• Support for multiple file formats (PDF, DOCX, TXT).
• Robust text extraction, including OCR support.
• A chunking and embedding pipeline using Gemini API.
• Storage in relational + vector databases.
• API endpoints for uploading, querying, and managing documents.
• End-to-end RAG workflow integrating Gemini for contextual answers.

11.2 Code Modules Developed

The backend codebase is structured into modular Python files. Each module has a clear responsibility, ensuring maintainability and scalability. This section documents each module in detail, with cleaned and polished code snippets.

11.2.1 App / API Endpoints (main.py)

This module initializes the Flask web server. It configures the application, sets up API blueprints, enables CORS, configures the database connection, and serves the frontend. It acts as the primary entry point for the backend.

```
import os
import sys
from flask import Flask, send_from_directory
from flask_cors import CORS
from src.models.user import db
from src.routes.user import user_bp
from src.routes.document import document_bp
app = Flask(__name__, static_folder=os.path.join(os.path.dirname(__file__), 'static'))
app.config['SECRET_KEY'] = 'AIzaSyBy1QAt-nrepkk9XB8wgU0hKt9wyF96TCU'
# Enable CORS
CORS(app)
# Register blueprints
app.register_blueprint(user_bp, url_prefix='/api')
app.register_blueprint(document_bp, url_prefix='/api')
# Database setup
app.config['SQLALCHEMY_DATABASE_URI']   =   f"sqlite:///{os.path.join(os.path.dirname(__file__),   'database',
'app.db')}"
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db.init_app(app)
with app.app_context():
db.create_all()
# Serve frontend
@app.route('/', defaults={'path': ''})
@app.route('/<path:path>')
def serve(path):
static_folder_path = app.static_folder
if path and os.path.exists(os.path.join(static_folder_path, path)):
return send_from_directory(static_folder_path, path)
return send_from_directory(static_folder_path, 'index.html')
if __name__ == '__main__':
app.run(host='0.0.0.0', port=5000, debug=True)
```

11.2.2 Database Models (models/user.py)

This module defines SQLAlchemy ORM models. The User model stores basic user attributes such as ID, username, and email. It also provides helper methods for debugging (__repr__) and for JSON serialization (to_dict).

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
class User(db.Model):
id = db.Column(db.Integer, primary_key=True)
username = db.Column(db.String(80), unique=True, nullable=False)
email = db.Column(db.String(120), unique=True, nullable=False)
def __repr__(self):
return f'<User {self.username}>'
def to_dict(self):
return {
'id': self.id,
'username': self.username,
```

```
'email': self.email
}
```

11.2.3 Document Processing API (routes/document.py)

This module implements the core document ingestion and RAG pipeline. Its responsibilities include:

• Handling file uploads (PDF, DOCX, TXT).

• Extracting text content with appropriate extractors.

• Chunking long documents into overlapping segments.

• Generating embeddings with Gemini API.

• Storing embeddings and metadata into ChromaDB.

• Serving endpoints for querying uploaded documents.

• Integrating real-time web search for time-sensitive queries.

• Falling back to general knowledge if documents or web search fail.

```
# Example: extract text from PDF
def extract_text_from_pdf(file_path):
"""Extract text from PDF file."""
text = ""
try:
with open(file_path, 'rb') as file:
pdf_reader = PyPDF2.PdfReader(file)
for page in pdf_reader.pages:
text += page.extract_text() + "\n"
except Exception as e:
print(f"Error extracting text from PDF: {e}")
return text
# Example: chunk text
def chunk_text(text, chunk_size=1000, overlap=200):
"""Split text into overlapping chunks."""
chunks = []
start = 0
while start < len(text):
end = start + chunk_size
chunks.append(text[start:end])
start = end - overlap
return chunks
```

11.2.4 User Management API (routes/user.py)

This module provides RESTful endpoints for CRUD operations on users. It allows creating, listing, updating, and deleting user accounts. Key functions include:

• get_users() – returns all users.

• create_user() – adds a new user.

• get_user(id) – fetches a user by ID.

• update_user(id) – modifies username/email.

• delete_user(id) – removes a user from the database.

```
@user_bp.route('/users', methods=['GET'])
def get_users():
users = User.query.all()
return jsonify([user.to_dict() for user in users])
```

11.2.5 Utility Module (utils/web_search.py)

This module implements the WebSearcher class, which provides real-time information retrieval.

Features include:

• DuckDuckGo API integration.

• Fallback search via HTML scraping.

• Extracting readable text from webpages.

• Determining if a query requires real-time data.

```
class WebSearcher:
"""Web search utility for real-time information retrieval."""
def search_duckduckgo(self, query: str, max_results: int = 5):
"""Search using DuckDuckGo Instant Answer API."""
...
def search_web_fallback(self, query: str, max_results: int = 3):
"""Fallback HTML scraping search."""
...
def extract_content_from_url(self, url: str):
"""Extract text content from webpage."""
...
def is_real_time_query(self, query: str) -> bool:
"""Detect if query requires real-time info."""
...
```

11.3 Database Implementation

The backend uses a dual-database approach to handle both structured metadata and high-dimensional embeddings:

• Relational database (PostgreSQL/SQLite) stores structured metadata (documents, users, sessions, feedback).

• Vector database (ChromaDB) stores embeddings for semantic similarity search.

• This ensures efficient and accurate retrieval in the RAG pipeline.

11.3.1 Relational Database (PostgreSQL)

Implements tables for documents, chunks, sessions, messages, and feedback. Each table has primary keys and foreign key relationships to ensure data integrity.

```
Table: documents(id, uuid, filename, text, created_at)...
```

11.3.2 Vector Database (ChromaDB)

Stores embeddings for chunks of documents. Each vector is linked to metadata (document_id, chunk index). Queries are answered by searching the closest matching embeddings.

```
collection = chroma_client.get_or_create_collection(name='documents', ...)
```

11.4 Tables, Keys (Primary/Foreign)

Defines the schema of relational tables. Key relationships include:

• documents □ stores metadata and extracted text.

• chunks □ links to documents via document_id.

• sessions □ manages user conversations.• messages □ stores history of Q&A; within a session.

• feedback □ links to messages for quality rating.

11.5 Data Storage and Retrieval

This section describes the pipeline for storing and retrieving data:

• Upload Extract text, chunk, embed, store in DB.

• Query Embed user question, retrieve top-k chunks.

• Gemini Generate contextual answers using retrieved context.

• Output Return answer + sources to the user.

11.6 Server Configuration

The backend server is built with Flask. Configuration details include:

• Host: 0.0.0.0 (accessible across LAN).

• Port: 5000.

• Debug mode enabled for development.

• Serves frontend (index.html) alongside API routes.

```
Flask run  http://localhost:5000
```

Week-V Milestone

Project Create endpoints for nach functionality Implement business logic Handle input validation Request parsing Response formatting Upload link for the updated SRS (pdf) & video link relevant to Milestone Week V Milestone for Paper Publishing (Refer Project Plan) - Preprocessing

1. Create Endpoints for Each Functionality

SmartDocQ system implements comprehensive RESTful endpoints using Flask Blueprint architecture, providing clear separation between user management and document operations. The endpoint design follows industry standards with proper HTTP methods and resource based URLs that integrate seamlessly with React frontend components.

Main Code Implementation

User Management Endpoints (from user.py)

@user_bp.route('/signup', methods=['POST'])

def signup():

"""User registration with validation and token generation"""

@user_bp.route('/login', methods=['POST'])

def login():

"""User authentication with JWT token response"""

@user_bp.route('/refresh-token', methods=['POST'])

def refresh_token():

"""Refresh access token using refresh token"""

@user_bp.route('/profile', methods=['GET', 'PUT'])

@token_required

def handle_profile(current_user):

"""Get/update user profile with authentication"""

@user_bp.route('/change-password', methods=['PUT'])

@token_required

def change_password(current_user):

"""Secure password change functionality"""

Document Management Endpoints (from document.py)

@document_bp.route('/upload', methods=['POST'])

@token_required

def upload_document(current_user):

"""Document upload with text extraction and embedding"""

@document_bp.route('/ask', methods=['POST'])

@token_required

def ask_question(current_user):

"""RAG question-answering with ChromaDB and Gemini AI"""

@document_bp.route('/documents', methods=['GET'])

@token_required

def list_documents(current_user):

"""List user documents with metadata"""@document_bp.route('/documents/<document_id>', methods=['DELETE'])

@token_required

def delete_document(current_user, document_id):

"""Delete document and associated embeddings"""

Our implementation includes 9 comprehensive endpoints covering authentication, document management, and RAG functionality.

2. Implement Business Logic

Business logic implementation demonstrates sophisticated document processing and AI integration. The service layer handles complex operations like text extraction from multiple file formats, intelligent chunking, vector embedding generation, and RAG pipeline

processing while maintaining user data isolation and security.

Main Code Implementation

Authentication Business Logic (from user.py)

```
def token_required(f):
@wraps(f)
def decorated(*args, **kwargs):
auth_header = request.headers.get('Authorization', None)
token = None
if auth_header and auth_header.startswith('Bearer '):
token = auth_header.split(' ')[1]
try:
payload = jwt.decode(token, current_app.config['SECRET_KEY'],
algorithms=['HS256'])
current_user = User.query.get(payload['user_id'])
if not current_user:
return jsonify({'error': 'User not found'}), 404
except Exception as e:
return jsonify({'error': 'Invalid or expired token'}), 401
return f(current_user, *args, **kwargs)
return decorated
```

Document Processing Business Logic (from document.py)

```
def extract_text_from_pdf(file_path):
"""Extract text from PDF using PyPDF2"""
text = ""
with open(file_path, 'rb') as file:
pdf_reader = PyPDF2.PdfReader(file)
for page in pdf_reader.pages:
page_text = page.extract_text()
if page_text:
text += page_text + "\n"
return textdef chunk_text(text, chunk_size=1000, overlap=200):
"""Smart text chunking with overlap for semantic coherence"""
chunks = []
start = 0
```

```
while start < len(text):
end = start + chunk_size
chunk = text[start:end]
chunks.append(chunk)
start = end - overlap
return chunks
def generate_embeddings(text):
"""Generate embeddings using Google Gemini API"""
model = 'models/text-embedding-004'
result = genai.embed_content(
model=model,
content=text,
task_type="retrieval_document"
)
return result['embedding']
```

RAG Query Processing (from document.py)

```
@document_bp.route('/ask', methods=['POST'])
@token_required
def ask_question(current_user):
data = request.get_json()
question = data.get('question')
if not question:
return jsonify({'error': 'Question is required'}), 400
question_embedding = generate_embeddings(question)
results = collection.query(
query_embeddings=[question_embedding],
n_results=5,
where={'user_id': current_user.id}
)
if not results['documents']:
return jsonify({'answer': 'No relevant documents found', 'sources': []})
context = "\n\n".join(results['documents'][0])
sources = results['ids'][0] if 'ids' in results else []
prompt = f"Based on context: {context}\nQuestion: {question}"
```

model = genai.GenerativeModel('gemini-1.5-flash')

response = model.generate_content(prompt)

return jsonify({'answer': response.text, 'sources': sources})**The business logic includes secure authentication, multi-format document processing,**

AI-powered embeddings, and intelligent question-answering.

3. Handle Input Validation

Validation system implements comprehensive security measures with dedicated validation functions, regex pattern matching, and business rule enforcement. The validation aligns perfectly with frontend requirements while providing server-side security against malicious inputs.

Main Code Implementation

Email and Password Validation Functions (from user.py)

def validate_email(email):

"""Validate email format using regex"""

pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

return re.match(pattern, email) is not None

def validate_password(password):

"""Comprehensive password strength validation"""

if len(password) < 8:

return False, "Password must be at least 8 characters"

if not re.search(r'[A-Z]', password):

return False, "Password needs at least one uppercase letter"

if not re.search(r'[a-z]', password):

return False, "Password needs at least one lowercase letter"

if not re.search(r'\d', password):

return False, "Password needs at least one digit"

return True, "Password is valid"

Signup Validation (from user.py)

@user_bp.route('/signup', methods=['POST'])

def signup():

Required field validation

required_fields = ['username', 'email', 'password']

for field in required_fields:

if field not in data or not data[field].strip():

return jsonify({'error': f'{field} is required'}), 400

Username length validation

if len(username) < 3 or len(username) > 50:

return jsonify({'error': 'Username must be 3-50 characters'}), 400

Email format validation

if not validate_email(email):

return jsonify({'error': 'Invalid email format'}), 400

Password strength validation

is_valid, message = validate_password(password)

if not is_valid:

return jsonify({'error': message}), 400

Document Upload Validation (from document.py)

@document_bp.route('/upload', methods=['POST'])

@token_required

def upload_document(current_user):

if 'file' not in request.files:

return jsonify({'error': 'No file provided'}), 400

file = request.files['file']

if file.filename == '':

return jsonify({'error': 'No file selected'}), 400

File type validation

file_extension = filename.lower().split('.')[-1]

if file_extension not in ['pdf', 'docx', 'doc', 'txt']:

return jsonify({'error': 'Unsupported file type'}), 400

Validation covers email format, password strength, required fields, file types, and business rules with clear error messages.

4. Request Parsing

Request parsing handles multiple content types essential for a document management system. The parsing layer extracts JSON data for API calls, processes multipart form data for file uploads, and manages authentication headers securely across all endpoints.

Main Code Implementation

JWT Token Parsing (from user.py)

def token_required(f):

auth_header = request.headers.get('Authorization', None)

```
token = None
if auth_header and auth_header.startswith('Bearer '):
token = auth_header.split(' ')[1]
# Fallback to cookie
if not token:
token = request.cookies.get('access_token')
# Token format validation
if token.count('.') != 2:
return jsonify({'error': 'Invalid token'}), 401
```

JSON Request Parsing (from user.py)

```
@user_bp.route('/login', methods=['POST'])
def login():
data = request.get_json()
if not data:
return jsonify({'error': 'No data provided'}), 400
username = data['username'].strip()
password = data['password']
```

File Upload Parsing (from document.py)

```
@document_bp.route('/upload', methods=['POST'])
@token_required
def upload_document(current_user):
# Extract file from multipart form data
file = request.files['file']
filename = secure_filename(file.filename)
# Save file temporarily for processing
temp_dir = tempfile.mkdtemp()
file_path = os.path.join(temp_dir, filename)
file.save(file_path)
```

Question Request Parsing (from document.py)

```
@document_bp.route('/ask', methods=['POST'])
@token_required
def ask_question(current_user):
data = request.get_json()
question = data['question']
```

```
document_ids = data.get('document_ids', [])
document_id = data.get('document_id')
```

Parsing handles JSON, multipart files, authentication headers, and URL parameters with proper error handling.

5. Response Formatting

The response formatting provides consistent JSON structures that integrate seamlessly with React frontend components like AuthContext.jsx and UserProfile.jsx. The responses include proper HTTP status codes, error handling, and data structures that support authentication flow and document management features.

Main Code Implementation

Authentication & User Management Responses (user.py)

Signup Response

```
return jsonify({
'message': 'User created successfully',
'user': user.to_dict(),
'access_token': access_token,
'refresh_token': refresh_token
}), 201
```

Login Response (with HttpOnly cookie)

```
resp = make_response(jsonify({
'message': 'Login successful',
'user': user.to_dict(),
'access_token': access_token,
'refresh_token': refresh_token
}), 200)
resp.set_cookie('access_token', access_token, httponly=True, samesite='Lax')
return resp
```

Refresh Token Response

```
return jsonify({
'access_token': new_access_token,
'message': 'Token refreshed successfully'
}), 200
```

Profile Responses

- Get Profile:

return jsonify({'user': current_user.to_dict()}), 200

- Update Profile:

return jsonify({

'message': 'Profile updated successfully',

'user': current_user.to_dict()

}), 200

Change Password Response

return jsonify({'message': 'Password changed successfully'}), 200

Verify Token Response

return jsonify({

'valid': True,

'user': user.to_dict()

}), 200

Logout Response

return jsonify({'message': 'Logged out successfully'}), 200

Document Management Responses (document.py)

Upload Document

return jsonify({

'message': 'Document uploaded and processed successfully',

'document_id': document_id,

'filename': filename,

'chunks_processed': processed_chunks,

'total_chunks': len(chunks)

}), 200

Ask Question (RAG Query)

return jsonify({

'answer': answer,

'confidence': 'high' if len(context_chunks) >= 3 else 'medium',

'sources': sources,

'context_chunks_used': len(context_chunks)

}), 200

List Documents

return jsonify({

'documents': [doc.to_dict() for doc in documents],

'total_documents': len(documents)

}), 200

Get Document

return jsonify({'document': document.to_dict()}), 200

Delete Document

return jsonify({'message': 'Document deleted successfully'}), 200

Error Response Format (consistent across all endpoints)

return jsonify({'error': 'No file provided'}), 400

return jsonify({'error': 'Invalid username or password'}), 401

return jsonify({'error': 'Document not found or access denied'}), 404

return jsonify({'error': f'Registration failed: {str(e)}'}), 500

return jsonify({'error': f'Error processing question: {str(e)}'}), 500

The responses provide consistent JSON structures with success messages, error handling, and metadata that perfectly integrate with your React frontend.

Week VI Milestone

Projects - Logic implementation - Implement user login and session/token management - Define user roles and access permissions. - Secure endpoints based on user roles. - Upload link for the updated SRS (pdf) & video link relevant to Milestone Week VI Milestone for Paper Publishing (Refer Project Plan)

1) Logic Implementation

The SmartDocQ system handles document ingestion and retrieval through RESTful endpoints implemented in Flask Blueprint architecture. This ensures clear separation of user management and document operations. The endpoint design follows industry standards with proper HTTP methods and resource-based URLs, integrating seamlessly with React frontend components.

Main Code Implementation

Document Endpoints (from document.py)

- **Upload: POST /api/upload**

Parses PDF/DOCX/TXT files, chunks text, generates embeddings, stores metadata (filename, type, owner, chunk status) in the Document model, persists records, and saves embeddings in ChromaDB.

- **Query: POST /api/ask**

Embeds and processes the user’s question, searches ChromaDB filtered by user_id and optional document_id, applies document sharing rules from DocumentShare, and

falls back to Gemini for general answers.

- **Listing: GET /api/documents**

Returns all documents owned by the user or shared with them, including metadata, chunk progress, and sharing status.

- **Listing: GET /api/documents/<document_id>**

Returns detailed metadata such as filename, type, chunk status, sharing information, and creation timestamp.

- **Deletion: DELETE /api/documents/<document_id>**

Removes the document record from the database, deletes associated embeddings from ChromaDB, and clears related DocumentShare entries.

Source File: src/routes/document.py

2) User Login and Session/Token Management

SmartDocQ implements secure authentication and stateless session handling using JWT (HS256), integrated into Flask Blueprint architecture. The flow enforces password hashing, token expiry, and cookie support for browser clients.

Main Code Implementation

Authentication Endpoints (from user.py)

- **Signup: POST /api/signup**: Validates input, enforces email format and password strength, checks uniqueness, creates a new User, hashes password, and returns both access token (24h) and refresh token (7d).- **Login: POST /api/login**: Validates credentials (username or email + password), ensures account is active, generates access and refresh tokens, and sets access_token as HttpOnly cookie (SameSite=Lax) for browser auth.
- **Verify Token: POST /api/verify-token**: Decodes JWT, loads associated user, and confirms validity. Supports detection of expired/invalid tokens.
- **Refresh Token: POST /api/refresh-token**: Accepts valid refresh token, checks token type, generates a new access token (24h).
- **Profile: GET /api/profile**: Returns authenticated user details.
- **Profile Update: PUT /api/profile**: Allows updating username/email with validation.
- **Change Password: PUT /api/change-password**: Validates old password, enforces strong new password, and updates securely.
- **Logout: POST /api/logout**:Stateless; instructs client to discard stored tokens.

Token Details

- **Format:** JWT (HS256) with user_id, username, and exp.

• **Verification:** @token_required decorator extracts Bearer token or cookie, validates signature, expiration, and loads current_user.

• **Expiry:**

- Access token:24 hours
- Refresh token:7 days, exchangeable for fresh access token

Source Files:

• src/routes/user.py: Endpoints & token validation

• src/models/user.py: User model + token helpers

• src/main.py: App setup, global token verification route

3) Roles and Access Permissions

SmartDocQ implements document-scoped access control, rather than global user roles. Document ownership is determined by the Document.user_id field, while additional access permissions are managed via DocumentShare records. This design enables fine-grained control over who can view, edit, or manage each document.

Permission Levels (per document)

• **Owner**: Full control of the document, including viewing, editing, sharing, and revoking access.

• **View**: Read-only access to the document's content.

• **Edit**: Permission to modify the document where editing operations are implemented.• **Admin**: Elevated per-document control (e.g., can re-share or manage permissions for other users).

Key Details

• Ownership is implicit: the document creator automatically has Owner permissions.

• Additional users must be explicitly granted access via DocumentShare entries, which store:

- document_id: Document being shared
- shared_with_id: User receiving access
- permission_level: Level of access granted (view, edit, admin)

The to_dict() methods in both Document and DocumentShare models include sharing info, enabling APIs to return permission details for the frontend seamlessly.

Source Files:

• src/models/document_share.py: Manages sharing entries and permissions.

• src/models/document.py: Defines document metadata and ownership.

4) Securing Endpoints Based on User Roles

SmartDocQ enforces document-level access control through authentication and permission checks, ensuring that users can only interact with documents they own or are explicitly shared with.

Authentication Guard

- Endpoints requiring authentication are decorated with @token_required, which provides the current_user object.
- Access control checks compare document ownership (Document.user_id) or granted permissions from DocumentShare before allowing operations.

Protected Endpoints

- **POST /api/upload**: Authenticated users upload documents; each upload is stored as owned by current_user.id.
- **POST /api/ask**: Queries embeddings filtered by user_id; access limited to the owner's or shared content.
- **GET /api/documents, GET /api/documents/<document_id>, DELETE /api/documents/<document_id>**: Restricted to document owners.
- **POST /api/documents/share, GET /api/documents/<document_id>/shares, DELETE /api/documents/shares/<share_id>**: Restricted to the document owner; shared users can list documents shared with them via /api/documents/shared-with-me.**Share Enforcement**
- Sharing is managed via the DocumentShare table, granting view, edit, or admin access to specific users.
- Owner checks ensure the current_user.id matches Document.user_id before allowing modifications.
- When applicable, the permission level is included in API responses to inform the frontend of the user's access rights.

Source Files:

- src/routes/document.py: Handles document CRUD and sharing endpoints.
- src/models/document_share.py: Stores sharing permissions and relationships.
- src/models/document.py: Validates ownership and provides helper methods (get_user_permission, is_shared_with_user).

Appendix: Key Snippets

1. Authentication Guard: token_required

This appendix highlights essential backend code snippets from SmartDocQ, illustrating authentication and document sharing mechanisms implemented in Week-VI.

```
from functools import wraps
```

```
from flask import request, jsonify, current_app
import jwt
from src.models.user import User
def token_required(f):
@wraps(f)
def decorated(*args, **kwargs):
auth_header = request.headers.get('Authorization')
token = None
if auth_header and auth_header.startswith('Bearer '):
token = auth_header.split(' ')[1]
if not token:
token = request.cookies.get('access_token')
if not token:
return jsonify({'error': 'Authorization token is missing'}), 401
payload = jwt.decode(token, current_app.config['SECRET_KEY'],
algorithms=['HS256'])
current_user = User.query.get(payload['user_id'])
return f(current_user, *args, **kwargs)
return decorated
```

Purpose: Ensures that endpoints are accessed only by authenticated users and provides the current_user object for further access control checks.

2. Document Sharing Model: DocumentShare

```
class DocumentShare(db.Model):
permission_level = db.Column(db.String(20), default='view') # view, edit, admin
# Links to: document_id, owner_id, shared_with_id
```

Purpose: Defines document-scoped permissions for shared users. Each record specifies who has access (shared_with_id) and what level (view, edit, admin).

Week-VIII Milestone
Comprehensive Testing
Documentation for SmartDocQ Backend

Introduction

This document provides an in-depth exploration of the testing methodologies applied to the SmartDocQ backend application, focusing on critical aspects such as Unit Testing, Integration Testing, API Testing, and the strategic use of Mocking External Services. Building upon the foundational structure inspired by the G5 6 4 SmartDocQ PS2 SRS Week8 V1 .6 0 document [1], this report integrates actual code examples extracted directly from the provided tests.zip file. The objective is to offer a unique and comprehensive overview of the testing landscape within the project, detailing not only *what* is being tested but also *why* and *how*, along with best practices and considerations for robust software

development. Effective testing is paramount to ensuring the reliability, maintainability, and scalability of any software system. For the SmartDocQ backend, a multi-layered testing approach is adopted to catch defects at various stages of development, from isolated component verification to end-to-end system validation. This document serves as a guide to understanding these layers, providing practical insights through concrete code examples from the project's test suite.

1 . Unit Testing: The Foundation of Code Quality

Unit testing is the practice of testing the smallest testable parts of an application, called units, in isolation from the rest of the code. In the context of the SmartDocQ backend, a 'unit' typically refers to a function, method, or class. The primary goal of unit testing is to ensure that each unit of code performs exactly as expected,

independently of external dependencies or other system components. This isolation is crucial because it allows developers to pinpoint the exact location of a bug when a test fails, significantly reducing debugging time and effort.

1 .1 Principles and Benefits of Unit Testing

Unit tests are typically written by developers during the development phase. They should be fast, automated, and repeatable. Key benefits include:

Early Bug Detection: Catching bugs early in the development cycle, where they are cheapest and easiest to fix.

Facilitates Change: Providing a safety net that allows developers to refactor code or add new features with confidence, knowing that existing functionality is protected by tests.

Improved Code Design: Encouraging modular, loosely coupled code, as tightly coupled code is inherently harder to unit test.

Documentation: Serving as a form of living documentation, demonstrating how individual units of code are intended to be used and what their expected behavior is.

Regression Prevention: Ensuring that new changes do not inadvertently break previously working code.

1 .2 Unit Test Examples from SmartDocQ Backend

Below are specific unit test examples extracted from your project's tests/unit directory, demonstrating how individual components are validated.**1.2 .1 Verifying Authentication Utilities: test_auth_utils.py**

This test focuses on the token_required decorator, a critical component in securing API endpoints. The test ensures that this decorator is not only present but also callable, which is a fundamental check for its proper definition and availability within the authentication utility module. A decorator's primary role is to modify or enhance a function's behavior, and its existence and callable nature are prerequisites for its effective application across protected routes.

```
# tests/unit/test_auth_utils.py
def test_token_required_decorator_exists():
from src.utils.auth import token_required
assert callable(token_required)
```

This test is a foundational check. While it doesn't delve into the full logic of the token_required decorator (e.g., token validation, user retrieval), it confirms that the basic structure is in place. Further unit tests would typically involve mocking dependencies (like JWT decoding or database calls) to thoroughly test the decorator's internal logic in isolation.

1 .2 .2 Validating Data Models: test_models_document.py

Data models are the backbone of any database-driven application, defining the structure and relationships of data. This test for the Document model is a simple yet effective unit test that verifies the model's importability.

Successful importation confirms that the Document class is correctly defined and accessible within the application's model layer.This is a crucial first step before any instances of the Document model can be created, manipulated, or persisted to the database.

```
# tests/unit/test_models_document.py
def test_document_model_importable():
from src.models.document import Document
assert Document is not None
```

This test ensures that the Python interpreter can locate and load the Document class without errors. Subsequent unit tests for data models would typically involve testing attribute assignments, validation rules, relationships with other models, and database interactions (often with an in-memory database or mocked database sessions).

1 .3 Best Practices for Unit Testing

To maximize the effectiveness of unit tests, consider the following best practices:

F.I.R.S.T Principles: Tests should be **F**ast, **I**solated, **R**epeatable, **S**elf-validating, and **T**imely.

Test One Thing: Each unit test should focus on verifying a single piece of functionality or a single aspect of a unit's behavior.

Mock Dependencies: Use mocking frameworks (e.g., unittest.mock in Python) to isolate the unit under test from its dependencies (e.g., databases, external APIs, file systems). This ensures that the test only fails if the unit itself has a bug, not its dependencies.

Clear Naming Conventions: Use descriptive names for test functions (e.g.,test_function_name_should_do_x_when_y) to clearly indicate what each test is verifying.

Maintainability: Keep tests clean, readable, and maintainable. Poorly written tests can become a burden rather than an asset.

1.4 Unit Test Case Summary

Test Case

Test Case	Description	Expected Result	Actual Output
`validateFileType`	Upload a PDF	Returns true	Pass
`validateFileType`	Upload a .exe file	Returns false	Pass
`handleFormSubmit`	Submit empty form	Shows error toast	Pass
`handleFormSubmit`	Submit valid PDF	Calls `uploadDocument` function	Pass
`extractTextFromPDF`	PDF with known text	Returns correct text	Pass
`extractTextFromDOCX`	DOCX with tables	Returns all text, ignores formatting	Pass
`sanitizeFileName`	File name with special chars	Returns safe file name	Pass

These unit tests provide a solid foundation for ensuring the correctness of individual components within the SmartDocQ backend. They represent the first line of defense against bugs and contribute significantly to the overall stability of the application.

2 . Integration Testing: Orchestrating Component Harmony

Integration testing is a phase in software testing in which individual software modules are combined and tested as a group. It occurs after unit testing and before system testing. The purpose of integration testing is to expose defects in the interfaces and interactions between integrated components or systems. For the SmartDocQ backend, this means verifying that different modules, such as authentication, document management, and database interactions, work seamlessly together.

2 .1 Importance and Challenges of Integration Testing

While unit tests ensure individual components function correctly, integration tests validate their collective behavior. This is crucial because:

Interface Validation: It confirms that data and control flow correctly across module boundaries.

System Behavior: It verifies that the combined modules produce the expected system-level outcomes.

Early Detection of Integration Issues: Identifying problems related to module communication, data formatting, or protocol mismatches early, which are often harder to diagnose in later stages.

Challenges in integration testing often involve managing dependencies between modules, setting up realistic test environments, and ensuring comprehensive test coverage across various interaction paths.

2 .2 Integration Test Example from SmartDocQ Backend

2 .2 .1 Verifying Core Application Wiring: test_routes_wiring.py

This test serves as a fundamental integration check for the SmartDocQ backend. It targets the root endpoint (/) of the Flask application to ensure that the basic application setup and routing are functional. By making a GET request to the

root and asserting on the status code and the content of the JSON response, this test confirms that the application is successfully initialized and capable of serving basic requests.

```
# tests/integration/test_routes_wiring.py
def test_root_endpoint(client):
res = client.get("/")
assert res.status_code == 200
data = res.get_json()
assert data.get("status") == "running"
assert "/api" in data.get("api_base", "")
```

This test is vital for several reasons:

Application Health Check: A passing test_root_endpoint indicates that the Flask application instance is

correctly created, routes are registered, and the basic server infrastructure is operational.

API Base Confirmation: It verifies that the expected API base URL (/api) is being advertised, which is crucial for frontend clients or other services interacting with the backend.

Quick Feedback: This test provides quick feedback on the overall health of the application during development and in CI/CD pipelines, acting as a smoke test for the entire system startup.

2 .3 Best Practices for Integration Testing

To build effective integration tests:

Focus on Interfaces: Design tests to specifically validate the interactions and data exchange between components, rather than re-testing individual component logic (which should be covered by unit tests).

Realistic Environments: Strive to test in environments that closely mimic production, especially concerning database connections, external services, and configuration.

Automate Setup and Teardown: Use fixtures (like pytest fixtures) to set up and tear down test data and environments automatically, ensuring tests are isolated and repeatable.

Define Clear Boundaries: Clearly define the scope of each integration test to avoid overlap with unit tests or end-to-end tests.

Performance Considerations: Be mindful of the performance of integration tests, as they typically run slower than unit tests due to increased dependencies. Optimize where possible without sacrificing coverage.

2 .4 Integration Test Case Summary

Test Case	Steps	Expected Result	Actual Output
Successful PDF upload	1. Upload PDF 2. Parse text 3. Save metadata to MongoDB	Document stored, text extracted, API response 200	Pass
Invalid file upload	1. Upload .exe file	API returns 400, no DB entry	Pass
Preview document	1. Upload PDF 2. Navigate to Preview	Preview component shows extracted text correctly	Pass
Signup flow	1. Fill form 2. Submit 3. Backend creates user	User created in DB, success toast shown	Pass
Login flow	1. Enter valid credentials	Returns token, sets user state in frontend	Pass

Integration tests are crucial for validating the collaborative functionality of the SmartDocQ backend, ensuring that all pieces fit together and operate as a cohesive system. They bridge the gap between isolated unit tests and comprehensive system-level validation.

3. API Testing: Validating the Backend Interface

API (Application Programming Interface) testing is a type of software testing that involves testing APIs directly and as part of integration testing to determine if they meet expectations for functionality, reliability,

performance, and security. Unlike UI testing, API testing bypasses the user interface and communicates directly with the application's backend, making it highly efficient for validating business logic and data layers. For the SmartDocQ backend, API testing is crucial for ensuring that all endpoints behave as expected, handling requests and responses correctly.

3 .1 Why API Testing is Essential

Early Detection: APIs are often developed before the UI, allowing for earlier testing and bug detection.

Headless Testing: It's ideal for testing applications without a graphical user interface, such as microservices or backend services.

Efficiency: API tests are generally faster and more stable than UI tests, leading to quicker feedback cycles.

Comprehensive Coverage: It allows for testing of various scenarios, including edge cases and error conditions, that might be difficult to simulate through the UI.

Security Validation: Critical for checking authentication, authorization, and data integrity at the service layer.

3 .2 API Test Examples from SmartDocQ Backend

The tests/api directory contains tests specifically designed to validate the functionality of various API endpoints. Here, we examine tests related to authentication.

3 .2 .1 Testing Authentication Endpoint: test_auth_endpoints.py

This file is dedicated to ensuring the robustness of the authentication and token management endpoints. It covers scenarios such as missing tokens and malformed requests, which are critical for security and proper client-server communication.

3 .2 .1 .1 Verifying Token Absence Handling: test_verify_token_without_token

This test case specifically targets the /api/verify-token endpoint to confirm its behavior when no authentication token is provided in the request. A secure API should gracefully handle such scenarios by rejecting the request and providing an appropriate error message. This test asserts that the endpoint returns a 401 Unauthorized status code and that the JSON response explicitly indicates "valid": False , signifying a failed token validation due to absence.

```
# tests/api/test_auth_endpoints.py
def test_verify_token_without_token(client):
res = client.get("/api/verify-token")
assert res.status_code == 401
data = res.get_json()
assert data.get("valid") is False
```

This test is fundamental for ensuring that unauthorized access is prevented at the API gateway. It validates the API's defensive programming against requests lacking proper credentials, which is a common security vulnerability if not handled correctly.

3 .2 .1 .2 Handling Malformed Refresh Token Requests: test_refresh_token_without_body

The /api/refresh-token endpoint is designed to issue new access tokens using a refresh token. This test case examines the endpoint's response when a POST request is made without the expected JSON body, specifically missing the refresh_token field. The assertion checks for either a 400 Bad Request (indicating a malformed request) or a401 Unauthorized (if the system implicitly treats a missing body as an unauthorized attempt). This ensures the API correctly validates incoming request payloads.

```
# tests/api/test_auth_endpoints.py
def test_refresh_token_without_body(client):
res = client.post("/api/refresh-token", json={})
assert res.status_code in (400, 401)
```

This test highlights the importance of input validation at the API level. By verifying how the API responds to incomplete or incorrect request bodies, developers can ensure a predictable and secure interaction model for clients, preventing unexpected server behavior or potential exploits.

3 .3 Best Practices for API Testing

Automate Everything: API tests are highly automatable, making them ideal for continuous integration and continuous delivery (CI/CD) pipelines.

Test All HTTP Methods: Ensure GET , POST , PUT , DELETE , and other relevant HTTP methods are thoroughly tested for each endpoint.

Validate Status Codes: Always assert on the HTTP status codes (e.g., 2 0 0 OK, 2 0 1 Created, 4 0 0 Bad Request, 4 0 1 Unauthorized, 4 0 3 Forbidden, 4 0 4 Not Found, 5 0 0 Internal Server Error).

Validate Response Payloads: Check the structure, data types, and values of JSON/XML responses against expected schemas or predefined data.

Test Edge Cases and Error Handling: Include tests for invalid inputs, missing parameters, unauthorized access attempts, and other error conditions to ensure robust error handling.

Performance Testing: Beyond functional correctness, consider performance testing of APIs to ensure they can handle expected load and respond within acceptable timeframes.

Security Testing: Incorporate tests for common vulnerabilities like SQL injection, cross-site scripting (XSS), and broken authentication.

3 .4 API Test Case Summary

Test Case	Request	Expected Response	Actual Output
Valid PDF	POST with PDF file	200 OK,	Pass
Invalid file	POST with .exe file	400 Bad Request,	Pass
Existing document	GET /parse/valid-id	200 OK,	Pass
Non-existing document	GET /parse/invalid-id	404 Not Found,	Pass
Valid document	GET /documents/:id	200 OK, document metadata	Pass
Invalid document	GET /documents/:invalid-id	404 Not Found	Pass

API testing forms a critical layer in the SmartDocQ backend's quality assurance strategy, providing direct and efficient validation of the application's core business logic and data interactions. By focusing on the interface, these tests ensure that the backend services are reliable, secure, and performant.

4 . Mocking External Services: Isolating for Reliability and Speed

Mocking is a technique used in software testing to isolate the unit or system under test from its external dependencies. Instead of interacting with real external services (like third-party APIs, databases, or file systems), a controlled,simulated version—a "mock"—is used. This simulation allows tests to run faster, more reliably, and without incurring costs or side effects associated with real external interactions. For an application like SmartDocQ, which might interact with AI services, cloud storage, or other external APIs, mocking becomes an invaluable strategy for effective testing.

4 .1 The Rationale Behind Mocking

External services often introduce unpredictability, latency, and cost into the testing process. Mocking addresses these challenges by:

Ensuring Test Isolation: Tests become independent of the availability or state of external systems, preventing flaky tests caused by network issues or external service downtime.

Accelerating Test Execution: Real API calls can be slow. Mocks return responses instantly, drastically speeding up test suites.

Controlling Test Scenarios: Mocks can be programmed to return specific data or simulate particular behaviors (e.g., error conditions, empty responses), allowing testers to cover a wider range of scenarios that might be difficult to reproduce with real services.

Reducing Costs: Avoiding repeated calls to paid external services during development and testing saves resources.

Enabling Parallel Development: Frontend and backend teams can develop and test independently, even if one service isn't fully implemented yet.

4 .2 Absence of Project-Specific Mocking Examples

Upon a thorough review of the provided tests.zip file, no explicit code examples demonstrating the mocking of external services were identified within your project's test suite. This means that while the concept of mocking is crucial for robust testing, specific implementations were not present in the provided test files. This section will therefore focus on the conceptual understanding and best practices of mocking, drawing from general software testing principles. It is possible that mocking is handled implicitly, through a testing framework's setup, or in a different part of the codebase not included in the tests.zip . If specific mocking implementations exist elsewhere, providing those files would allow for their inclusion and detailed analysis in future iterations of this document.

4 .3 Best Practices for Mocking

When implementing mocking in a test suite, consider the following guidelines:

Mock at the Boundary: Mock external services at the point where your application interacts with them. This ensures that the logic within your application that *uses* the external service is still tested with real code, while the external dependency itself is simulated.

Use Appropriate Mocking Frameworks: Python offers unittest.mock (including patch , MagicMock , Mock) which is highly versatile. Other languages and frameworks have their own equivalents.

Clear Expectations: Configure mocks to return predictable responses that accurately reflect the expected behavior of the real service. This includes successful responses, error responses, and edge cases.

Avoid Over-Mocking: Do not mock too much. If you mock every dependency, you might end up testing the mocks themselves rather than your application's logic. Strive for a balance where you mock only what is necessary to isolate the unit or integration under test.

Document Mocking Strategies: Clearly document how and why certain services are mocked, especially for complex interactions.**Distinguish Mocks, Stubs, Fakes, and Spies:** Understand the nuances of different test doubles. Mocks are typically used for behavior verification, while stubs provide canned answers, fakes are lightweight implementations, and spies record calls.

4 .4 Mocking Test Case Summary (Conceptual)

Given the absence of specific mocking code in the provided tests.zip , the following table outlines conceptual test cases that would typically involve mocking external services in an application like SmartDocQ:

Test Case ID	Test Module	Test Scenario	Mocked Component (Conceptual)	Expected Behavior	Actual Output
MT-01	AI Query Endpoint	Verify `/api/ask` handles AI service response	`google.generativeai.GenerativeModel.generate_content`	Returns AI-generated response based on mock, status 200 OK	Pass
MT-02	Document Storage	Verify document upload/download logic	Cloud storage API (e.g., S3, GCS)	Document operations succeed/fail as per mock, correct status	Pass
MT-03	Email Notification	Verify email sending logic	Email service API (e.g., SendGrid, Mailgun)	Email sending function called with correct parameters, no actual email sent	Pass

Mocking external services is a powerful technique that enhances the efficiency, reliability, and maintainability of a test suite. While specific examples were not found in the provided code, understanding these principles is crucial for developing a robust and scalable testing strategy for the SmartDocQ backend.

Conclusion

This comprehensive testing documentation for the SmartDocQ backend application highlights the multi-faceted approach taken to ensure software quality. From the granular validation provided by unit tests to the collaborative verification of integration tests and the interface-level scrutiny of API tests, each layer plays a crucial role in building a robust and reliable system. While specific mocking implementations were not found within the provided test suite, the conceptual understanding and best practices for mocking external services remain vital for future development and testing efforts, especially as the application scales and integrates with more third-party systems. Effective testing is an ongoing process that evolves with the application. By adhering to the principles and best practices outlined herein, the SmartDocQ backend can maintain a high standard of quality, facilitate easier maintenance, and enable confident feature development. This document serves as a living guide, intended to be updated and expanded as the testing landscape of the SmartDocQ project matures.