Software Requirements Specification

for

SMARTDOCQ

KESHAV MEMORIAL INSTITUE OF TECHNOLOGY

<version 9.0>

Team Members:

23BD1A0567-B.KOMAL

23BD1A0568-CH.RUCHITHA

23BD1A056B-R.UDAYA SRI

23BD1A1296-K.ANANYA

23BD1A12C1-S.SUSHMITHA

Table of Contents

Table of Contents 2
Revision History 3
1. Introduction 1
1.1 Purpose 1
1.2 Document Conventions 1
1.3 Intended Audience and Reading Suggestions 1
1.4 Product Scope 2
1.5 References 21
2. Overall Description 2
2.1 Product Perspective 2
2.2 Product Functions 3
2.3 User Classes and Characteristics 4
2.4 Operating Environment 4
2.5 Design and Implementation Constraints 5
2.6 User Documentation 5
2.7 Assumptions and Dependencies 6
3. External Interface Requirements 6
3.1 User Interfaces 6
3.2 Hardware Interfaces 9
3.3 Software Interfaces 10
3.4 Communications Interfaces 11
4. System Features 12
4.1 System Feature 1 .... **Error! Bookmark not defined.**
4.2 System Feature 2 (and so on) .... **Error! Bookmark not defined.**
5. Other Nonfunctional Requirements 16
5.1 Performance Requirements 16
5.2 Safety Requirements 17
5.3 Security Requirements 17
5.4 Software Quality Attributes 18
5.5 Business Rules 19
6. Other Requirements 20
Appendix A: Glossary .... **Error! Bookmark not defined.**
Appendix B: Analysis Models 20
Appendix C: To Be Determined List .... **Error! Bookmark not defined.**

Revision History

Name	Date	Reason For Changes	Version
G552	28-07-2025	Introduction, overall description, external interface requirements	S.0
G552	30-07-2025	System Features, other non-functional requirements	S.1
G552	30-07-2025	Architecture, database, use case, sequence, work flow, deployment diagrams	S.2
G552	13-08-2025	Other requirements	S.3
G552	20-08-2025	Server Side , Environmental Setup and Integrate Database with back-end	S.4
G552	23-08-2025	End points for each functionality, Implementing Business logic, Handle input validation, Request parsing , Response Formatting	S.5
G552	02-09-2025	Logic Implementation, Implementing user login , Defining user roles and accessing permissions, Secure endpoints based on user roles.	S.6
G552	14-09-2025	Unit Testing, Integration Testing, API Testing, Mocking External Services.	S.7
G552	26-09-2025	Create and Setup Environmental Settings for both staging and production	S.9

1. Introduction

1.1 Purpose

This Software Requirements Specification (SRS) defines the requirements for SmartDocQ – Your AI Assistant for Smarter Document Q\&A (v1.0). This system allows users to upload documents such as PDFs, Word files, text files, and URLs, and ask natural language questions based on their contents. Using AI technologies like Google Gemini and Retrieval-Augmented Generation (RAG), the system retrieves relevant content segments and produces intelligent, context-aware responses.

This document focuses on the core subsystem responsible for document ingestion, semantic embedding, vector storage, and intelligent question answering. The target audience includes developers, testers, and stakeholders involved in designing, implementing, and evaluating SmartDocQ.

1.2 Document Conventions

- Bold is used for section headings and important terms.
- Times New Roman indicate referenced components or modules.
- Monospaced text is used for code, APIs, and system functions.
- Each requirement will be identified with a unique identifier (e.g., FR-1, NFR-3).
- All requirements are assumed to inherit the priority of their parent categories unless explicitly stated.
- Font Size: 16 and Bold for Titles, 14 and Bold for heading and 12 for content

1.3 Intended Audience and Reading Suggestions

This document is intended for:

Developers: To implement backend (FastAPI) and frontend (React) components.

Testers: To design and run tests on functional and non-functional modules.

Project Managers: To track milestones and ensure delivery aligns with requirements.

End Users and UI Designers: To understand user interaction expectations.

Stakeholders: To assess the product scope, goals, and deliverables.

Product Scope

SmartDocQ is a smart, AI-based question answering platform for documents. Its key objective is to transform static documents into interactive learning tools. By enabling semantic search and contextual Q\&A using AI models like Google Gemini, it enhances document comprehension for students, researchers, educators, and legal/corporate professionals.

The tool:

- ➢ Simplifies studying by providing quick, accurate answers.
- ➢ Converts documents into searchable vector embeddings.
- ➢ Supports memory tracking and feedback loops.
- ➢ Can be scaled for enterprise or academic applications.

This aligns with broader goals of digital transformation in education and corporate knowledge management.

2. Overall Description

2.1 Product Perspective

SmartDocQ is a new, self-contained AI-based document Q\&A system designed to enable users to interact intelligently with textual documents by uploading files and asking natural language questions. It is not part of an existing product family or system, but it can be extended to integrate with other academic, research, or content management systems via APIs.

The system architecture includes:

- A React.js frontend for user interaction
- A FastAPI backend for handling file processing, question answering, and AI logic
- A vector database (FAISS/Chroma) for semantic search
- An LLM (e.g., Mistral via Ollama) for local and free answer generation

2.2 Product Functions

- SmartDocQ provides the following high-level functions:
- Upload document files (PDF, DOCX, TXT)
- Extract and chunk content into meaningful segments
- Generate semantic embeddings using local models
- Store and retrieve data using a vector database (FAISS or Chroma)
- Use a local LLM (like Mistral) to generate accurate natural-language answers
- Provide an interactive chat interface for question-answering
- Optional memory handling for multi-turn conversation
- Accept feedback from users to rate responses
- Log chat history and answers for future analysis

2.3 User Classes and Characteristics

User Class	Characteristics
Students	Basic computer knowledge; use SmartDocQ to ask questions from textbooks/notes
Teachers	Moderate tech experience; upload lecture notes and create dynamic Q\&A sets
Researchers	Experienced users; analyze research papers and technical documents
General Users	Casual use; no advanced skills needed; expect intuitive UI and fast responses

Most users are expected to have basic English literacy and use SmartDocQ through a browser interface.

No programming knowledge is required.

2.4 Operating Environment

SmartDocQ will operate in the following environment:

Frontend : Browser-based (Chrome, Firefox, Edge), built with React.js

Backend : Python-based FastAPI server running on Windows/Linux/macOS

Database : FAISS or ChromaDB for vector storage

Embedding Model : sentence-transformers (e.g., MiniLM)

LLM Model : Mistral/OpenHermes running via Ollama or LM Studio

Deployment : Can run locally or be hosted on cloud platforms (optional)

2.5 Design and Implementation Constraints

Must use only free and offline-compatible AI tools (e.g., Mistral via Ollama, MiniLM for embeddings).

- Must work on systems with limited RAM/CPU (e.g., 8 GB).
- Frontend must be built using React.js; backend must use FastAPI.
- APIs must be RESTful and documented.
- Embedding models must not exceed a reasonable size (e.g., MiniLM \~100MB).
- No use of paid services like OpenAI or Gemini unless explicitly authorized.
- Must support basic file types only: .pdf, .docx, .txt.

2.6 User Documentation

The following user documentation will be provided:

User Manual : PDF or HTML document explaining how to use the system.

Quick Start Guide : Embedded in the UI or linked from the homepage.

In-App Tooltips : Inline hints on upload and chat functionality.

Developer README : Contains setup instructions, architecture details, and code usage for developers.

API Reference : Swagger UI (auto-generated via FastAPI)

2.7 Assumptions and Dependencies

Assumptions:

- Users will upload readable, text-based documents (OCR for images is not included).
- All AI models (embedding and LLM) will run locally without API keys.
- Users will operate the application on a modern system (8+ GB RAM recommended).
- All inputs will be in English.

Dependencies:

- Relies on open-source Python libraries such as sentence-transformers, faiss-cpu, PyMuPDF, and ollama.
- System performance may depend on hardware specs.
- If deployed online, external dependencies (e.g., hosting, SSL, domain) may be required.

3. External Interface Requirements

3.1 User Interfaces

SmartDocQ provides a clean, modern, and interactive web interface built using React.js. It is designed for seamless interaction with AI-driven features such as document Q\&A, summarization, voice input, multilingual support, and downloadable reports. The UI is user-friendly and accessible across devices.

Core Interface Components

Document Upload Panel

- Accepts: PDF, DOCX, TXT, and URLs
- Drag & drop or manual file selection
- Shows progress bar, validation messages, and file metadata
- Multiple file upload support

Chat & Q\&A Interface

- Text Input Field: For typing natural language questions
- Voice Input : Users can ask questions via microphone (Speech-to-Text API)
- Submit Button: Sends query to the backend (FastAPI + Gemini)

Answer Display:

- Natural language response
- Highlights the referenced document snippet
- Allows feedback
- Shows processing time

Summarization Panel

- Appears after document upload
- Options:
- Full document summary
- Section-wise summary
- Summary can be copied or downloaded

Multi-Language Support

- Language dropdown or settings gear icon
- Interface and responses available in multiple languages
- Supports both question input and AI response localization
- Languages: English, Hindi, Telugu, and others (based on user preference)

Downloadable Reports

Allows exporting:

- Individual Q\&A pairs
- Full chat history
- Document summaries
- Formats: PDF, TXT
- Button: Download Report shown in chat panel and summary screen

Navigation / Sidebar

Menu: Home, Upload, Chat History, Summary, Downloads, Settings, Help, Logout

Chat History Panel

- Lists past interactions
- Searchable by keyword
- Allows restoring a previous session
- Design Standards & Guidelines
- Material UI / Google Material Design
- Color contrast and typography for accessibility

Responsive across:

- Desktop (Chrome, Firefox)
- Mobile (Android/iOS browsers)
- Tablet devices
- Error Handling & Notifications

Inline Errors & Toast Alerts for:

- Unsupported files
- Empty input or failed response
- Speech recognition issues
- Clear loading indicators and fallback messages

3.2 Hardware Interfaces

SmartDocQ is a web-based application and requires no direct hardware integration. However, the system will interact indirectly with the following hardware:

Client-Side Hardware (Users):

- Desktop, Laptop, Tablet, or Smartphone with a modern web browser.
- Microphone (for future voice support).

Server-Side Hardware:

- Backend runs on cloud-hosted virtual servers (e.g., AWS, GCP).
- GPU-accelerated machines for AI inference (for Gemini model calls or embedding generation).
- SSD-based storage for speed in document vector retrieval.

Data Exchange Characteristics:

- All data is transferred over HTTP/HTTPS protocols.
- Uploaded documents are stored temporarily in server memory or disk for pre-processing.

3.3 Software Interfaces

SmartDocQ interacts with several software components:

Backend and AI Integration:

- FastAPI (v0.95+): Handles routing, document upload, processing, and API services.
- Google Gemini API: Provides language understanding and generation (Q&A, summarization).
- ChromaDB/Pinecone: Vector database for storing and retrieving document embeddings.
- MongoDB (v6+): Stores user metadata, document metadata, and chat logs.
- Redis/In-memory caching: Manages session state and memory module (short-term history).
- Frontend:
- React.js (v18+): Provides the web-based interactive interface.
- Communicates with FastAPI via RESTful APIs.

Libraries and Tools:

- Tika, pdfplumber: For document parsing
- LangChain/Sentence Transformers: For embedding generation
- dotenv, logging, uvicorn, pydantic: Environment and validation tools

Data Flow Summary:

- Input: Documents (uploaded), Queries (text)
- Output: Answers (text), Summaries, Highlight references, Chat history

- All API routes are RESTful and secured using JWT-based authentication.

3.4 Communications Interfaces

SmartDocQ is a cloud-hosted web application that communicates over the internet using secure protocols.

Communication Mechanisms:

- HTTP/HTTPS: Used for all frontend-backend and backend-AI API interactions.
- WebSockets (optional for future real-time chat updates)
- JWT Tokens: Used for authentication between client and server.
- Data Uploads: Documents are uploaded over HTTPS with MIME type verification.
- API Calls: RESTful requests to Google Gemini and vector DB.

Security Considerations:

- HTTPS enforced: All communication is SSL/TLS encrypted.
- File Validation: Prevents uploading of malicious files.
- CORS Policy: Configured to restrict unauthorized cross-origin access.
- Rate Limiting & API Throttling: Prevents misuse of AI APIs.

Performance Metrics:

- Document upload response within 2–4 seconds (average)
- Q&A response latency under 5 seconds for <10MB documents

Supported Browsers:

- Chrome, Firefox, Safari, and Edge (latest 2 versions recommended)

4. System Features

4.1 Document Upload and Parsing

4.1.1Description and Priority

This feature allows users to upload documents (PDF, DOCX, or TXT) into the system for analysis and question answering. It includes content extraction and chunking.

Priority: High

4.1.2Stimulus/Response Sequences

Stimulus: User selects and uploads a document.

System Response:

Parses document text

Divides text into manageable chunks (e.g., 300 words)

Stores original file for reference

Shows “Upload successful” message

4.1.3Functional Requirements

REQ-1.1: The system shall accept files with .pdf, .docx, and .txt extensions.

REQ-1.2: The system shall reject files exceeding 10 MB

REQ-1.3: The system shall extract readable text from the uploaded file.

REQ-1.4: The system shall chunk extracted text into smaller segments for embedding.

REQ-1.5: The system shall notify the user of successful or failed upload.

4.2 Embedding Generation and Vector Storage

4.2.1 Description and Priority

- Generates semantic vector embeddings from text chunks and stores them for later retrieval.
- Priority: High

4.2.2 Stimulus/Response Sequences

- Stimulus: A document is successfully parsed and chunked.
- System Response:
- Generates semantic vectors for each chunk
- Stores vectors in FAISS or ChromaDB
- Logs vector generation completion

4.2.3 Functional Requirements

REQ-2.1: The system shall use a sentence-transformer model to convert text chunks into vectors.

REQ-2.2: The system shall store all generated vectors in a vector database.

REQ-2.3: The system shall store reference metadata (document name, chunk number) with each vector.

REQ-2.4: The system shall log errors in vector generation or storage.

4.3 Question Answering Interface

4.3.1 Description and Priority

Allows users to type in natural language questions and receive accurate, context-based answers.

Priority: High

4.3.2 Stimulus/Response Sequences

Stimulus: User enters a question in the chat input box.

System Response:

- Converts question into a vector
- Retrieves top-matching document chunks
- Sends content to the LLM
- Displays generated answer on the frontend

4.3.3 Functional Requirements

REQ-3.1: The system shall allow the user to input a question via text box.

REQ-3.2: The system shall convert the question into a semantic vector.

REQ-3.3:The system shall retrieve top-K similar vectors from the vector store.

REQ-3.4: The system shall pass relevant content and the question to the local LLM (e.g., Mistral).

REQ-3.5: The system shall display the generated answer in the frontend chat interface.

REQ-3.6: The system shall return an error message if no document is uploaded.

4.4 Answer Feedback and Session Tracking

4.4.1 Description and Priority

- Enables users to rate answers and maintains session history for future reference (optional in v1).
- Priority: Medium
- Benefit: 6

4.4.2 Stimulus/Response Sequences

- Stimulus: User clicks 👍 or 👎 on an answer.
- System Response:
- Logs feedback
- Optionally stores Q\&A pair in session memory for context tracking

4.4.3 Functional Requirements

REQ-4.1: The system shall allow users to rate an answer using a thumbs-up/down button.

REQ-4.2: The system shall log feedback in a structured format.

REQ-4.3: The system shall maintain chat history during a single session.

REQ-4.4: (Optional) The system may associate a session ID for long-term memory storage.

5. Other Nonfunctional Requirements

5.1 Performance Requirements

SmartDocQ is designed for high performance and responsiveness to ensure smooth and efficient interactions. Performance goals are as follows:

Response Time:

- Document upload and chunking: ≤ 4 seconds for files up to 10MB
- Question-answer generation: ≤ 5 seconds per query (using Gemini API + vector retrieval)
- Summarization: ≤ 7 seconds for standard-length documents (\~5 pages)

Concurrency:

- Supports simultaneous interaction by at least 100 concurrent users.
- Load balancing handled via API gateway and scalable cloud instances.

Availability:

- 99.5% uptime with fallback error messages in case of AI model/API failure.

Resource Utilization:

- Memory-efficient in-memory caching (e.g., Redis) for session context.
- GPU acceleration used for embedding generation and AI inference when available.

5.2 Safety Requirements

SmartDocQ is primarily a web-based document assistant and poses minimal physical safety risks. However, digital safety is prioritized through the following measures:

File Validation:

- Uploaded documents are scanned for malicious content and unsupported formats.
- Scripts or embedded executable content are blocked.

Data Isolation:

- Users cannot access others' documents or data under any circumstance.
- Session boundaries are strictly enforced.

Fallback Behavior:

- In the event of AI model failure, the system provides top matching document snippets as a fallback mechanism instead of returning blank output.

Compliance:

- Adheres to best practices recommended by OWASP and relevant data safety norms (ISO/IEC 27001 guidelines if scaled).

5.3 Security Requirements

Given that SmartDocQ processes potentially sensitive educational or legal documents, strict security controls are enforced:

Authentication & Authorization:

- JWT-based user login with email/password credentials.
- Role-based access (e.g., Admin, Student, Educator) for future extensions.

Data Security:

- All data is transmitted via HTTPS (SSL/TLS encryption).
- Uploaded documents and metadata are stored in secured databases (MongoDB).

Session Management:

- Sessions are automatically invalidated after a period of inactivity (e.g., 30 minutes).
- Rate limiting to prevent DDoS attacks or abuse of Gemini API.

Compliance Requirements:

- Designed to align with GDPR and Indian IT Act 2000 (with 2023 amendments) for data protection.

Software Quality Attributes

Attribute	**Description**
Usability	Intuitive, minimal-click interface accessible to students, teachers, and researchers.
Reliability	Recovers gracefully from partial failures (e.g., AI fallback, offline alerts).
Maintainability	Modular FastAPI backend with clear routing structure and logging

Portability	Runs on any modern web browser; backend deployable on AWS, GCP, or local Linux servers
Interoperability	Supports standard file formats (PDF, DOCX, TXT) and integrates with Gemini, ChromaDB.
Testability	Unit-tested APIs; frontend components tested via Jest or Cypress.
Availability	Target availability is 24/7 with minimal maintenance downtime.
Scalability	Easily scales via Docker containers and Kubernetes (for future enterprise expansion).
Flexibility	Plugin-ready architecture for adding future features like collaboration, analytics, etc

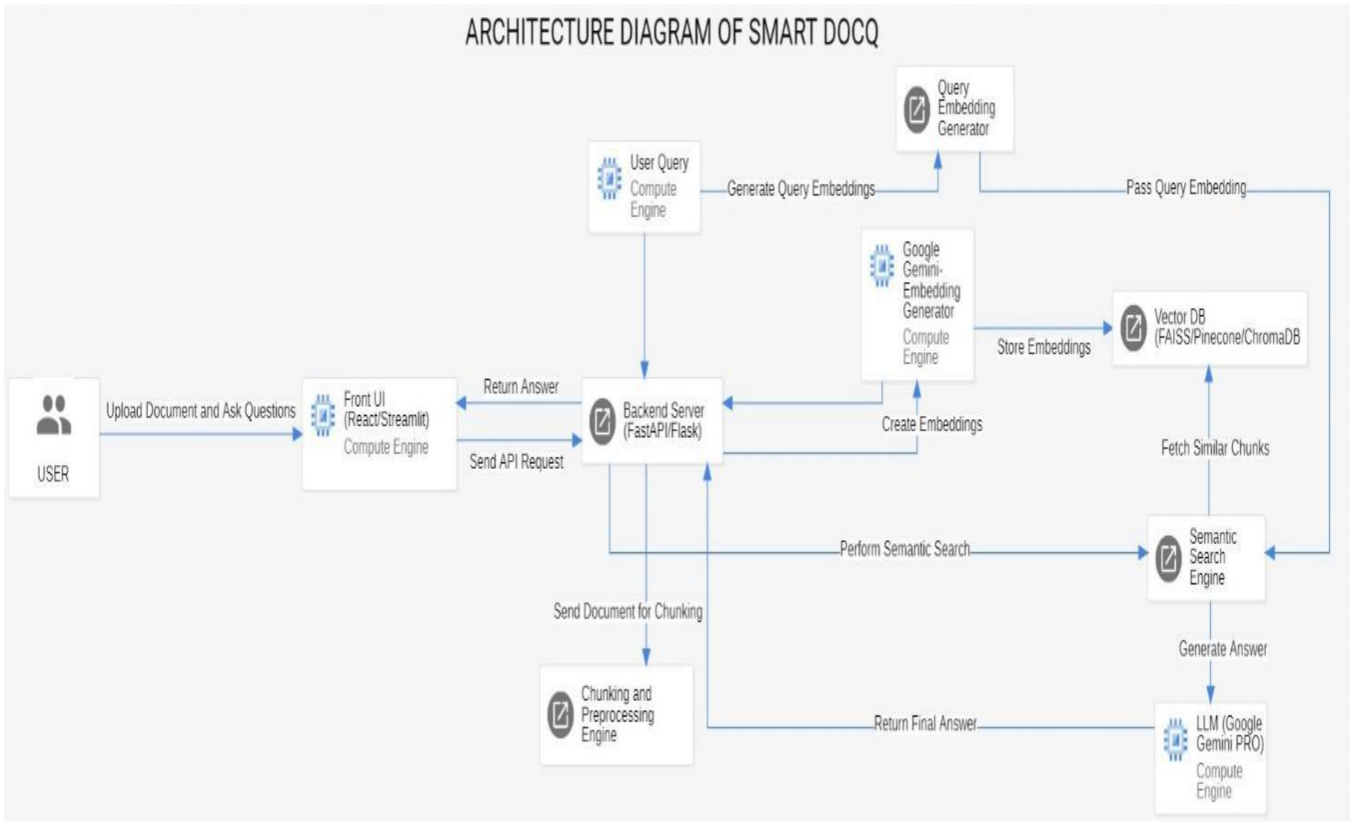
5.4 Business Rules

- Only authenticated users can upload and query documents.
- Uploaded documents are private to each user unless shared explicitly in future collaborative modules.
- Feedback submission is allowed only after a complete answer is received.
- Multi-language responses must match the language selected in user preferences.
- Voice input must be allowed only on browsers that support speech recognition.
- Downloadable content (summaries or Q\&A reports) is restricted to the session owner.
- Admin roles (future feature) can view anonymized usage analytics but not document contents

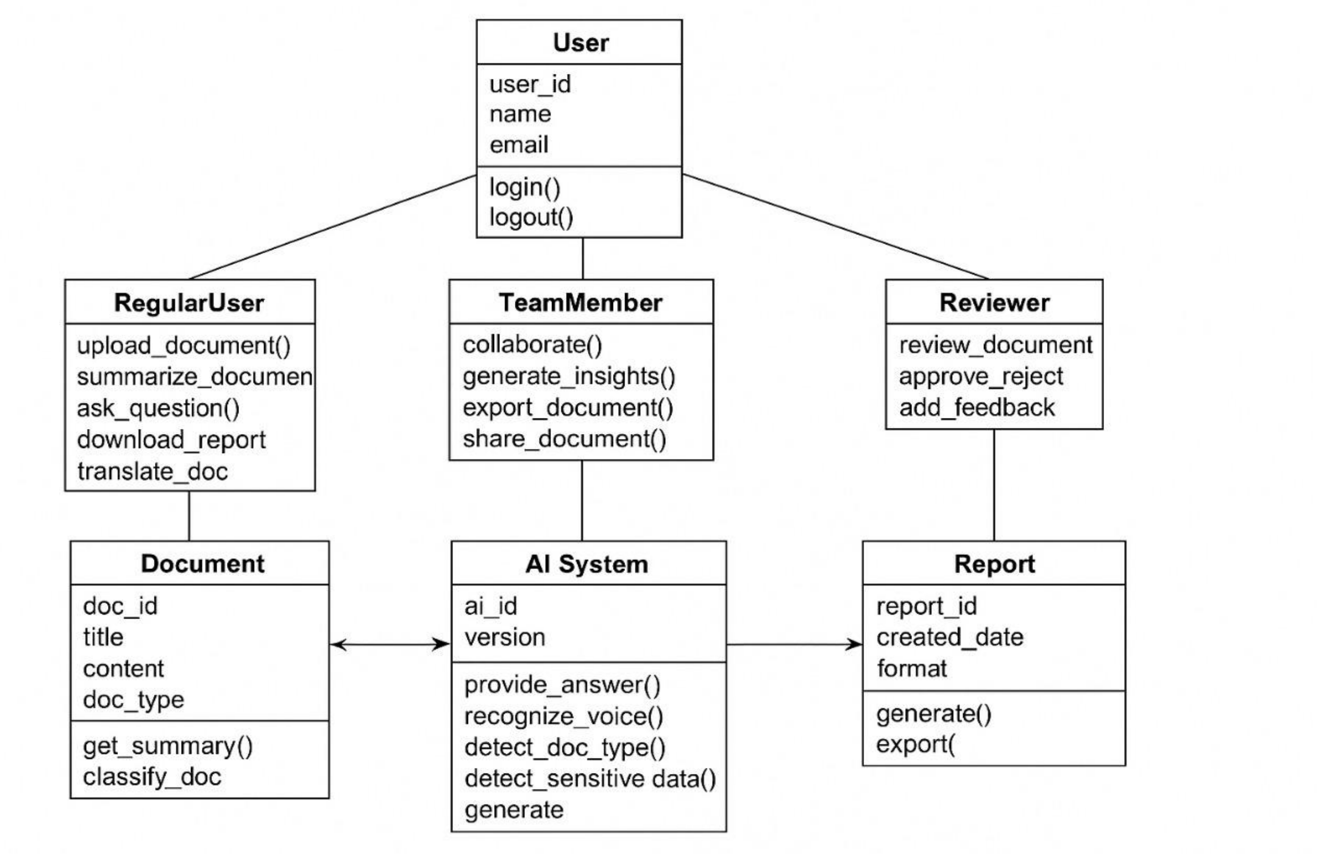
6. Other Requirements

Appendix A: Analysis Models

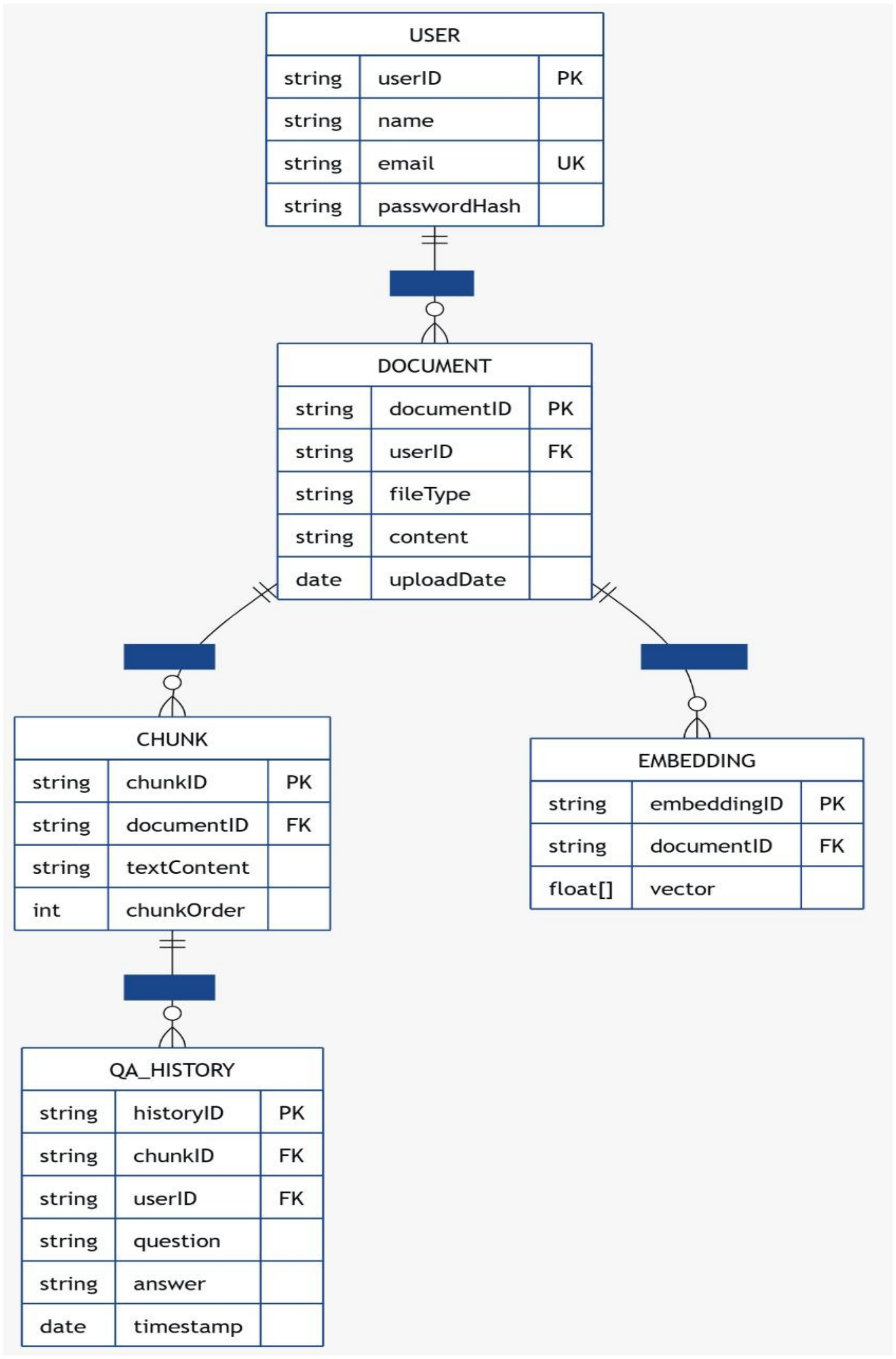
Architecture Diagram


ARCHITECTURE DIAGRAM OF SMART DOCQ
Query Embedding Generator
User Query Compute Engine
Generate Query Embeddings
Pass Query Embedding
Google Gemini-Embedding Generator Compute Engine
Store Embeddings
Vector DB (FAISS/Pinecone/ChromaDB)
USER
Upload Document and Ask Questions
Front UI (React/Streamlit) Compute Engine
Return Answer
Send API Request
Backend Server (FastAPI/Flask)
Create Embeddings
Fetch Similar Chunks
Perform Semantic Search
Semantic Search Engine
Send Document for Chunking
Generate Answer
Chunking and Preprocessing Engine
Return Final Answer
LLM (Google Gemini PRO) Compute Engine

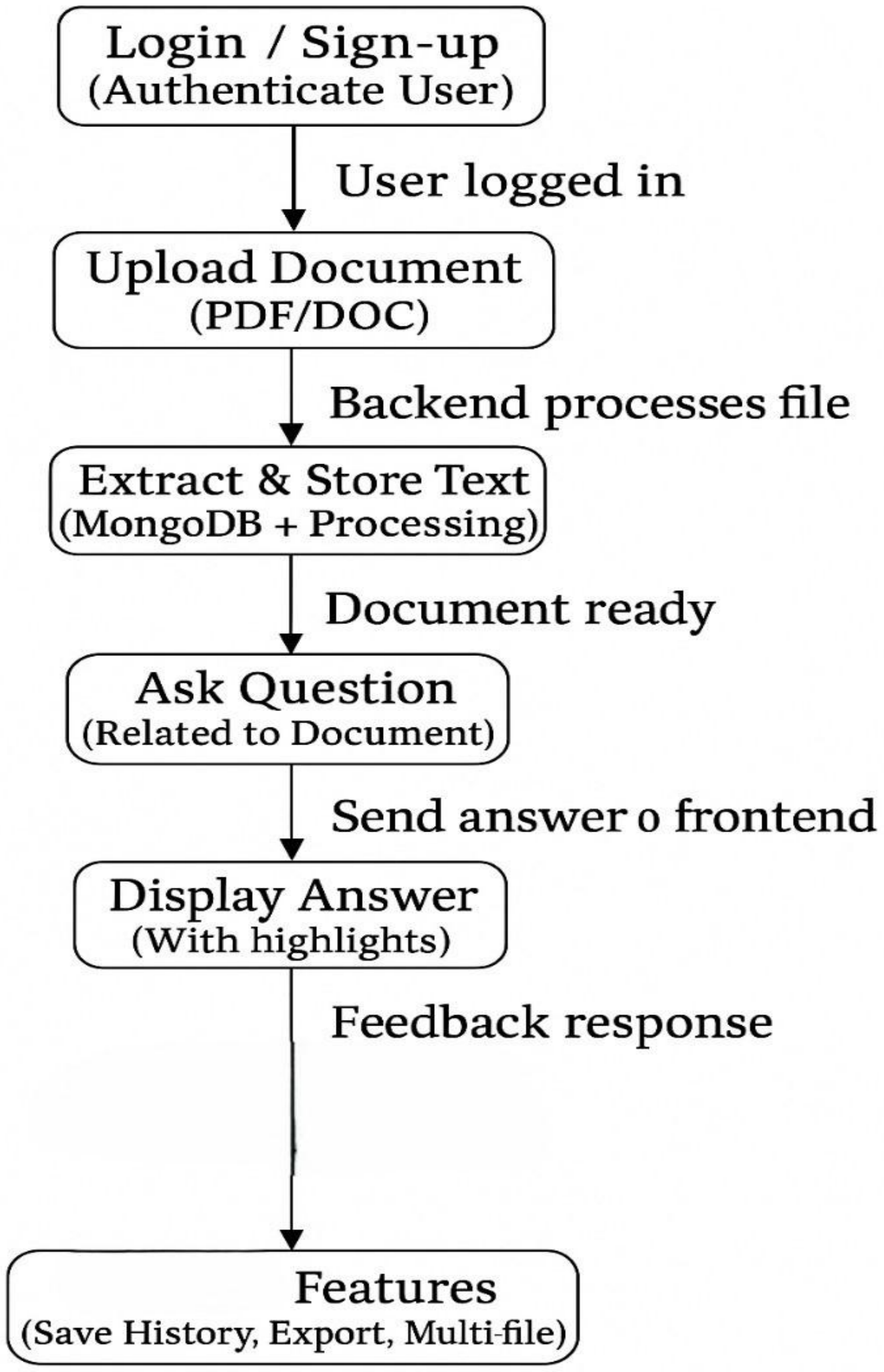
Class Diagram

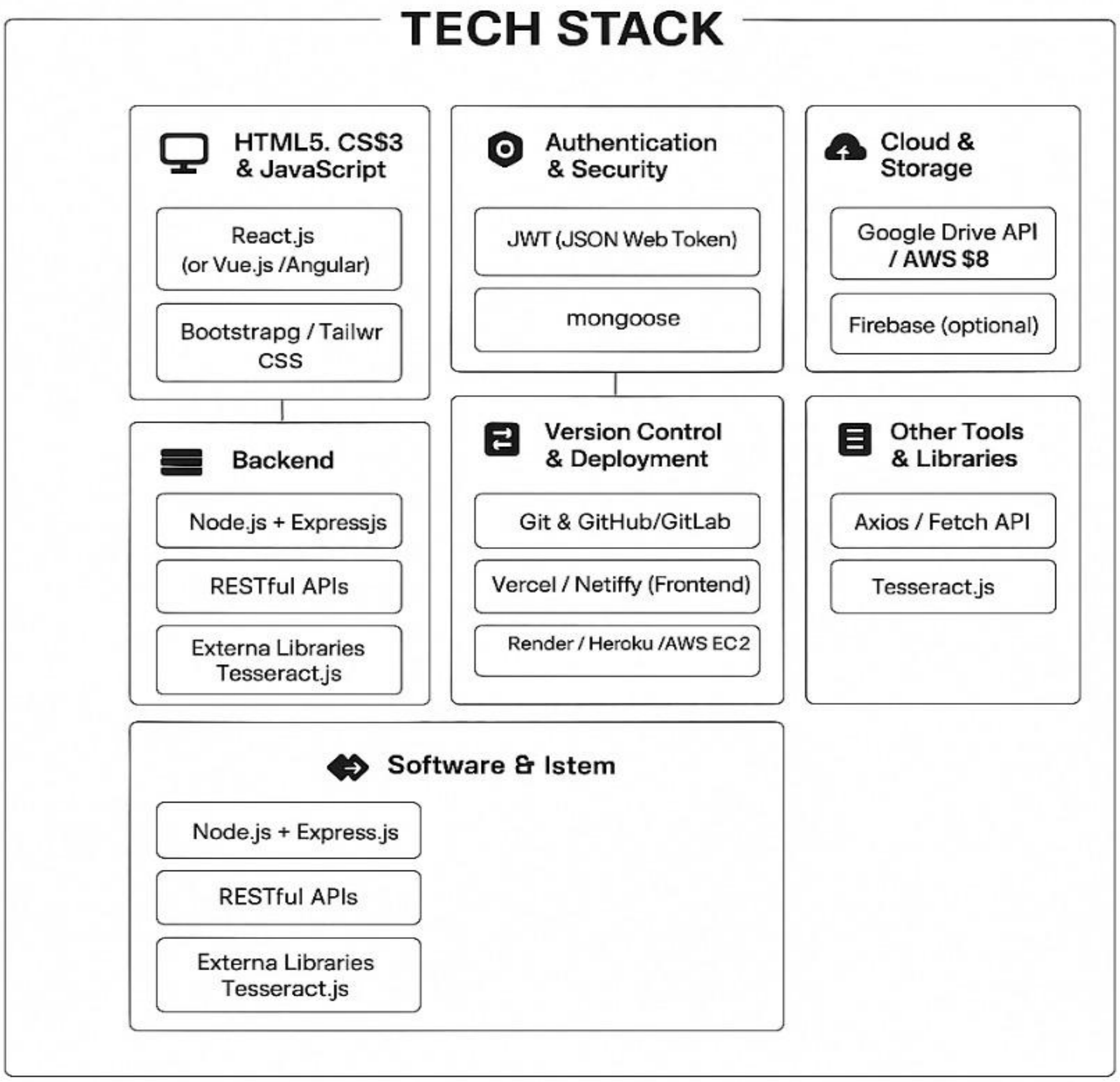


Database Diagram

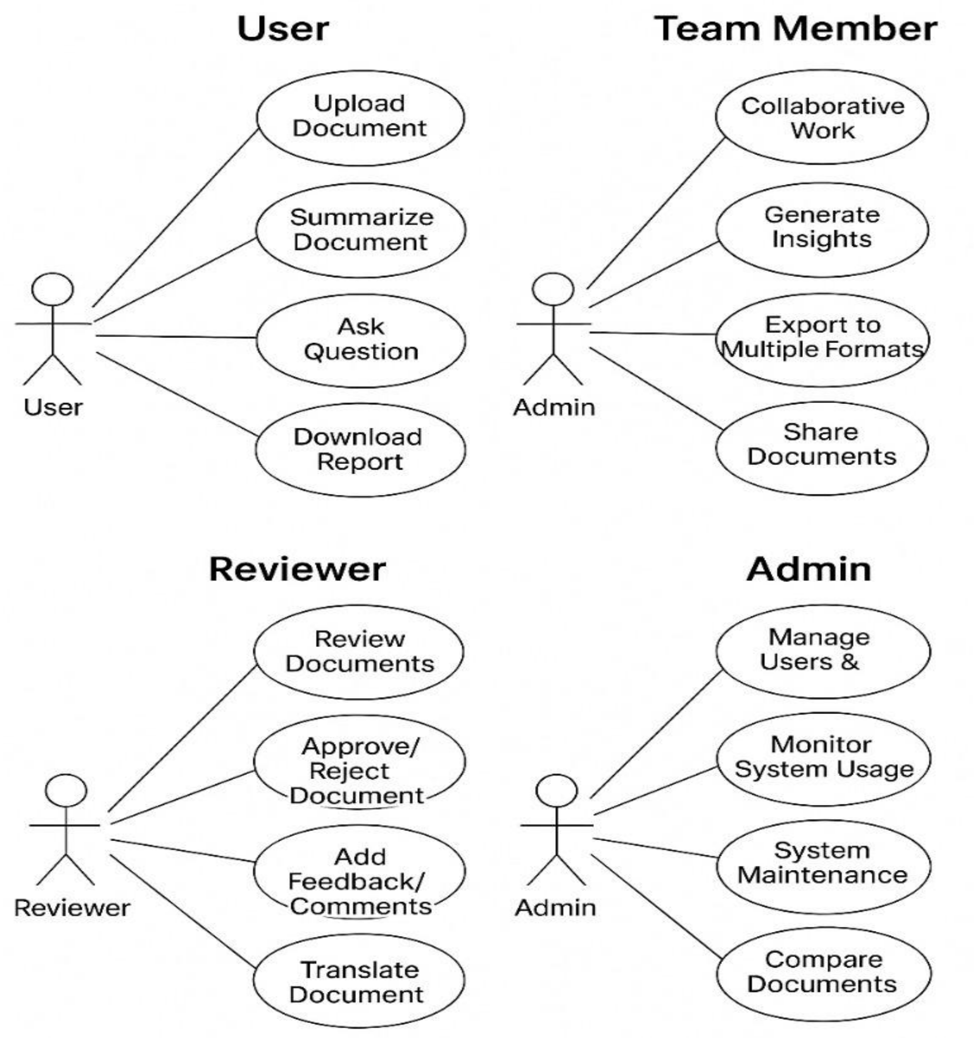


Work Flow

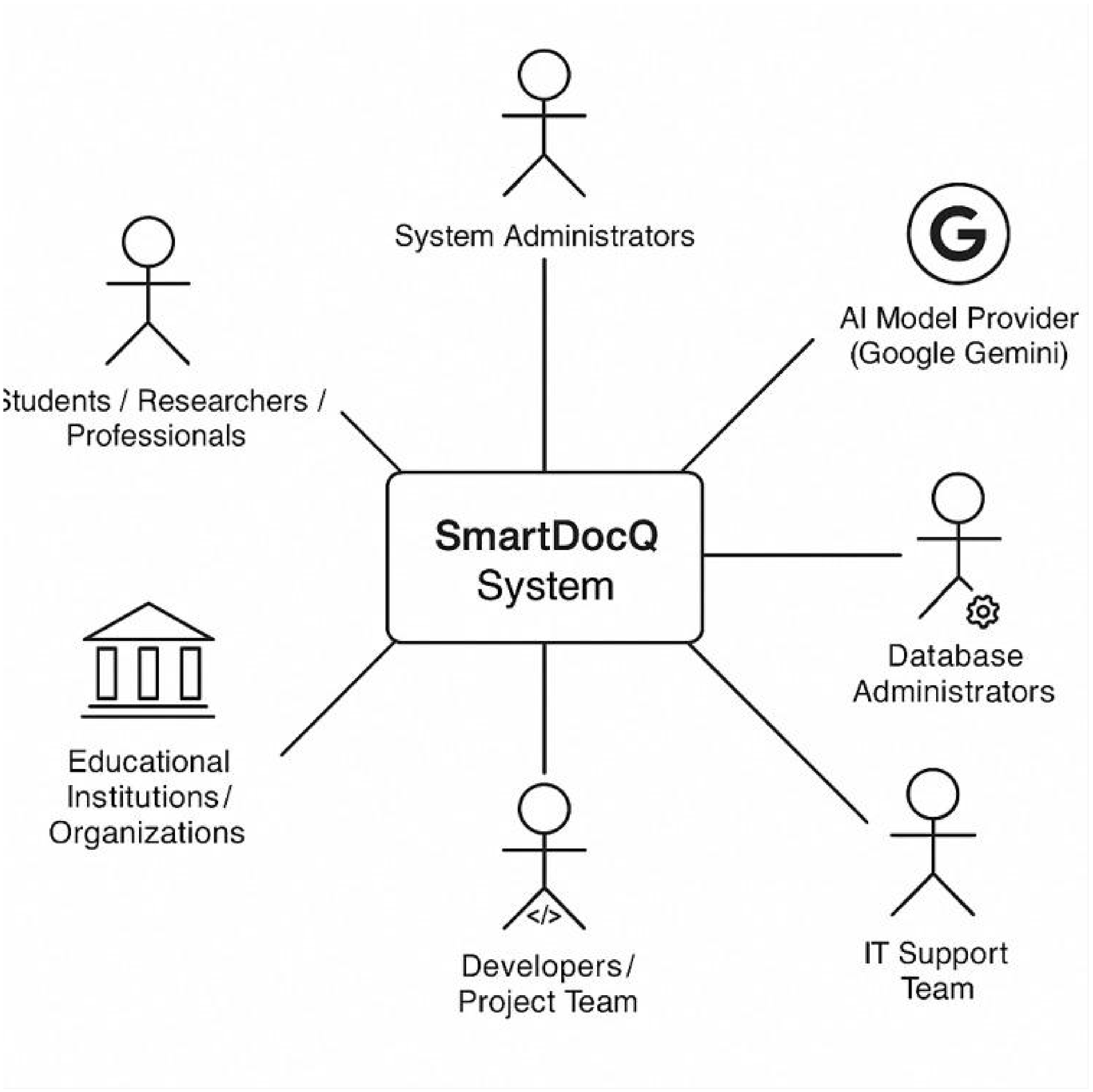

Login / Sign-up
(Authenticate User)
User logged in
Upload Document
(PDF/DOC)
Backend processes file
Extract & Store Text
(MongoDB + Processing)
Document ready
Ask Question
(Related to Document)
Send answer o frontend
Display Answer
(With highlights)
Feedback response
Features
(Save History, Export, Multi-file)


TECH STACK
HTML5. CS$3 & JavaScript
React.js (or Vue.js /Angular)
Bootstrapg / Tailwr CSS
Authentication & Security
JWT (JSON Web Token)
mongoose
Cloud & Storage
Google Drive API / AWS $8
Firebase (optional)
Backend
Node.js + Expressjs
RESTful APIs
Externa Libraries Tesseract.js
Version Control & Deployment
Git & GitHub/GitLab
Vercel / Netiffy (Frontend)
Render / Heroku /AWS EC2
Other Tools & Libraries
Axios / Fetch API
Tesseract.js
Software & Istem
Node.js + Express.js
RESTful APIs
Externa Libraries Tesseract.js

Use Case Diagram



List Of Stake Holders


System Administrators
AI Model Provider (Google Gemini)
Students / Researchers / Professionals
SmartDocQ System
Database Administrators
Educational Institutions/ Organizations
</>
Developers/ Project Team
IT Support Team

Backend Development

1. Introduction

In this milestone, we shifted our focus from requirement analysis and planning stages (covered in the SRS and earlier milestones) to the **backend development** of the SmartDocQ system. The backend serves as the foundation of the application, handling document uploads, extracting and processing text, generating embeddings, and storing them for efficient retrieval.

So far, we have:

- Set up the **FastAPI** backend environment with proper server configuration.
- Implemented APIs for **file upload, text extraction, and storage**.
- Integrated **Gemini API** for embeddings and generative responses.
- Configured **ChromaDB** as a persistent vector database for storing and retrieving document chunks.

This milestone connects the earlier design work with practical implementation by establishing a fully functional backend pipeline that will later be linked with the frontend UI. It ensures that when users upload documents, the system can already process them and provide meaningful answers to queries.

Database Design

For the SmartDocQ backend, we did not use a traditional relational database (like MySQL or PostgreSQL) for storing documents. Instead, we implemented a **vector database (ChromaDB)** to store processed document chunks along with their embeddings. This choice was made because our system requires **semantic search** and **retrieval-augmented generation (RAG)**, where queries are answered based on the similarity of embeddings rather than exact keyword matches.

Databases Created

- **ChromaDB (Vector Database)**
    - **Purpose:** To store embeddings of document chunks extracted from PDF, DOCX, or TXT files.
    - **Role**: Enables fast similarity search to retrieve the most relevant chunks for answering user queries.
- **File Storage (Local Directory)**
    - **Purpose**: Raw files uploaded by users are saved in the backend for traceability and possible reprocessing.
    - **Role**: Acts as a supporting storage layer, not a database in the strict sense.

Why These Databases Are Required

- **ChromaDB**: Required to implement intelligent question answering. Without embeddings, the system cannot understand the semantic meaning of queries.
- **File Storage**: Ensures files can be reprocessed if embedding models are updated or if additional features (like summaries or metadata extraction) are added in the future.

Type of Tables (or Collections) Chosen

Since ChromaDB is not relational, instead of traditional tables, it works with **collections**. A collection is similar to a table but is designed for vector embeddings.

- **Collection Name**: smartdocq
- **Stored Fields**:
  - **id** (Unique identifier for each chunk → acts like Primary Key)
  - **embedding** (Vector representation of the chunk → main field for similarity search)
  - **document** (Original chunk text)
  - **metadata** (Stores filename, document id, and chunk index → acts like supporting keys)

Justification:

- Collections with embeddings allow for **high-performance vector similarity search**, which is central to the working of a document-based Q&A system.
- Metadata serves as a way to logically link chunks back to their original documents (similar to Foreign Keys in relational DBs).

3. Tables and Keys

Although SmartDocQ uses a **vector database (ChromaDB)** instead of a relational DB, we can still represent its structure in terms of tables and keys for clarity.

Main Collection: smartdocq

Purpose: Stores all processed document chunks along with their embeddings, enabling semantic search.

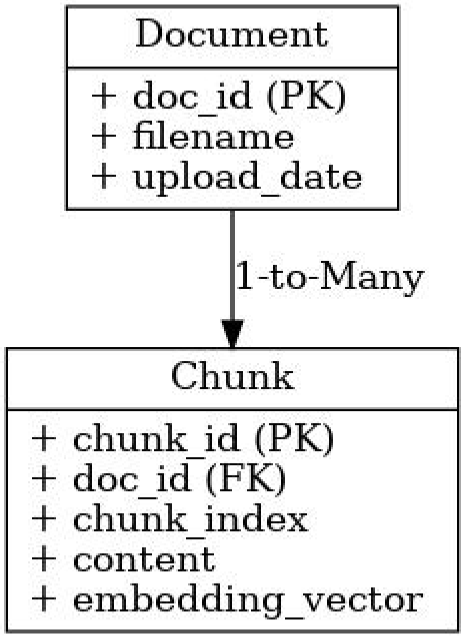
Attributes / Columns

1. **id** (Primary Key)
   - A unique identifier for each chunk in the format: doc_id_chunkIndex.
   - Example: 123e4567_0
2. **embedding**
   - High-dimensional vector (list of floats) generated by Gemini Embedding Model.
   - Used for similarity search.
3. **document**
   - Actual chunk of text extracted from the original file.
4. **metadata** (acts like a JSON object containing supporting attributes):
   - **doc_id** → Unique identifier for the uploaded file.
   - **filename** → Original name of the uploaded file.
   - **chunk_index** → Index of the chunk inside that document.

Keys

- **Primary Key**:
  - id → ensures every chunk entry is unique.
- **Foreign Key Equivalent**:

- doc_id inside metadata → links all chunks back to the same document.
- While not a strict Foreign Key (since Chroma is non-relational), it serves the same purpose of maintaining relationships.



4. Data Flow / Retrieval

4.1 Data Storage

Data in SmartDocQ is organized in two main structures:

1. **Document Metadata**
   - Stored in the backend as part of the Document collection.
   - Includes doc_id, filename, upload_date, and file_path.

  - Ensures each uploaded file is tracked and can be reprocessed if needed.

2. **Document Chunks**

  - Stored in the smartdocq collection in ChromaDB.
  - Each chunk contains:
    - id → unique chunk identifier (Primary Key)
    - document → extracted text chunk
    - embedding → vector representation for semantic search
    - metadata → {doc_id, filename, chunk_index} linking chunks to their **document**

4.2 Data Retrieval

SmartDocQ uses **semantic search** via embeddings for efficient retrieval:

- **Query Process**:
  1. User submits a query.
  2. Query is converted into an embedding vector using the Gemini API.
  3. ChromaDB finds the most similar chunk embeddings.
  4. Top relevant chunks are returned as results.

  Example Queries:

```
chunks = db.get_chunks(filter={"metadata.doc_id": doc_id})

query_embedding = generate_embedding(user_query)

top_chunks = db.similarity_search(query_embedding, top_k=5
```

4.3 Relationships

Document → Chunk: One-to-Many
Each document (doc_id) can have multiple chunks (id).

Chunk → Document: Many-to-One
Each chunk belongs to exactly one document (metadata.doc_id acts as a foreign key equivalent).

5. Backend Code

1. **FastAPI Setup**

```
from fastapi import FastAPI

app = FastAPI()
```

- Initializes the FastAPI server for handling requests.

2. **File Upload API**

```
@app.post("/upload/")
async def upload_file(file: UploadFile):
```

- Handles user document uploads and stores them in the backend folder.

3. **Text Extraction & Chunking**

```
text = extract_text(file_path)

chunks = split_into_chunks(text)
```

- Extracts text from PDF/DOCX/TXT files and splits them into smaller chunks.

4. **Embedding & Storage**

```
embedding = generate_embedding(chunk)

db.insert({

    "id": chunk_id,

    "document": chunk,

    "embedding": embedding,

    "metadata": {"doc_id": doc_id, "filename": file.filename, "chunk_index": index}

})
```

- Generates embeddings for each chunk and stores them in ChromaDB with metadata.

Database Connection

- ChromaDB is connected via the backend code, enabling storage and retrieval of document chunks for semantic search.

6. Server / Connectivity

- The backend runs on a FastAPI server configured locally or on a cloud instance.

- **APIs Created So Far**:

  1. /upload/ → Upload document
  2. /query/ → Submit a question and get top relevant chunks

- ChromaDB is integrated with FastAPI, allowing queries to retrieve document chunks based on embeddings.

- Gemini API is connected to generate embeddings and handle query processing.

7. Summary

- **Achievements in this milestone**:
  - Backend environment setup with FastAPI.
  - APIs implemented for file upload, text extraction, embedding generation, and storage.
  - ChromaDB integrated for vector-based storage and retrieval.
  - Document and chunk data structures designed with proper primary and foreign key equivalents.
  - Semantic search functionality enabled for document Q&A.
  - Integrate the backend with the frontend UI.
  - Implement user authentication and multi-user support.

- Enhance query handling and implement advanced features like summarization or metadata search.

BACKEND DEVELOPMENT AND PREPROCESSING

1. Create endpoints for each functionality

Endpoints for all core functionalities have been created.

• User Authentication (Login, Register, Logout)

• File Upload & Retrieval

• Data Processing / Business Logic Execution

• Query / Search functionality

• Admin Operations

2. Implement business logic

Core business logic has been implemented according to the use cases.

• Input handling and processing are working as expected.

• Core algorithms and workflows have been integrated with the endpoints.

3. Handle input validation

Input validation has been added to all endpoints.

• Mandatory field checks

• Type validation (string, number, file format)

• Constraints (length, format, allowed values)

• Error messages for invalid input

4. Request parsing

Implemented request parsing for:

• JSON requests

• Form-data (for file uploads)

• Query parameters in GET requests

• Proper error handling on invalid/malformed requests

5. Response formatting

Standardized response format has been set:

• Success → { status: "success", data: {...}, message: "..." }

• Error → { status: "error", message: "..." }

• Consistent HTTP status codes (200, 400, 401, 404, 500)

Preprocessing

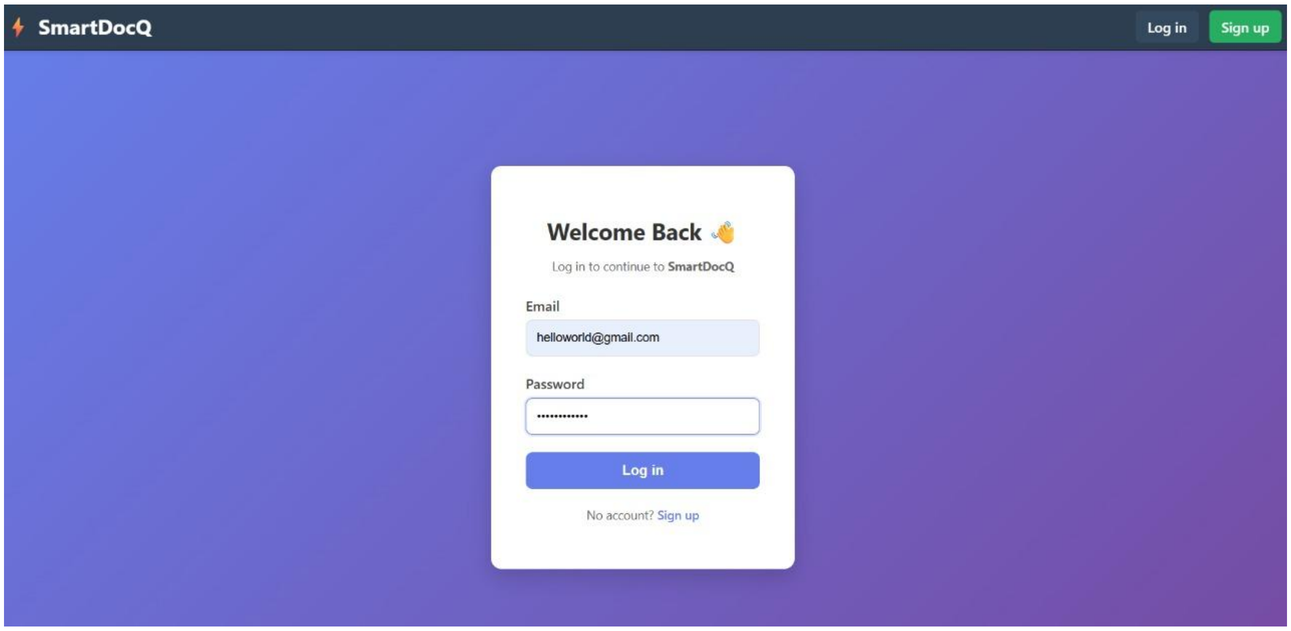
Preprocessing phase has been completed.

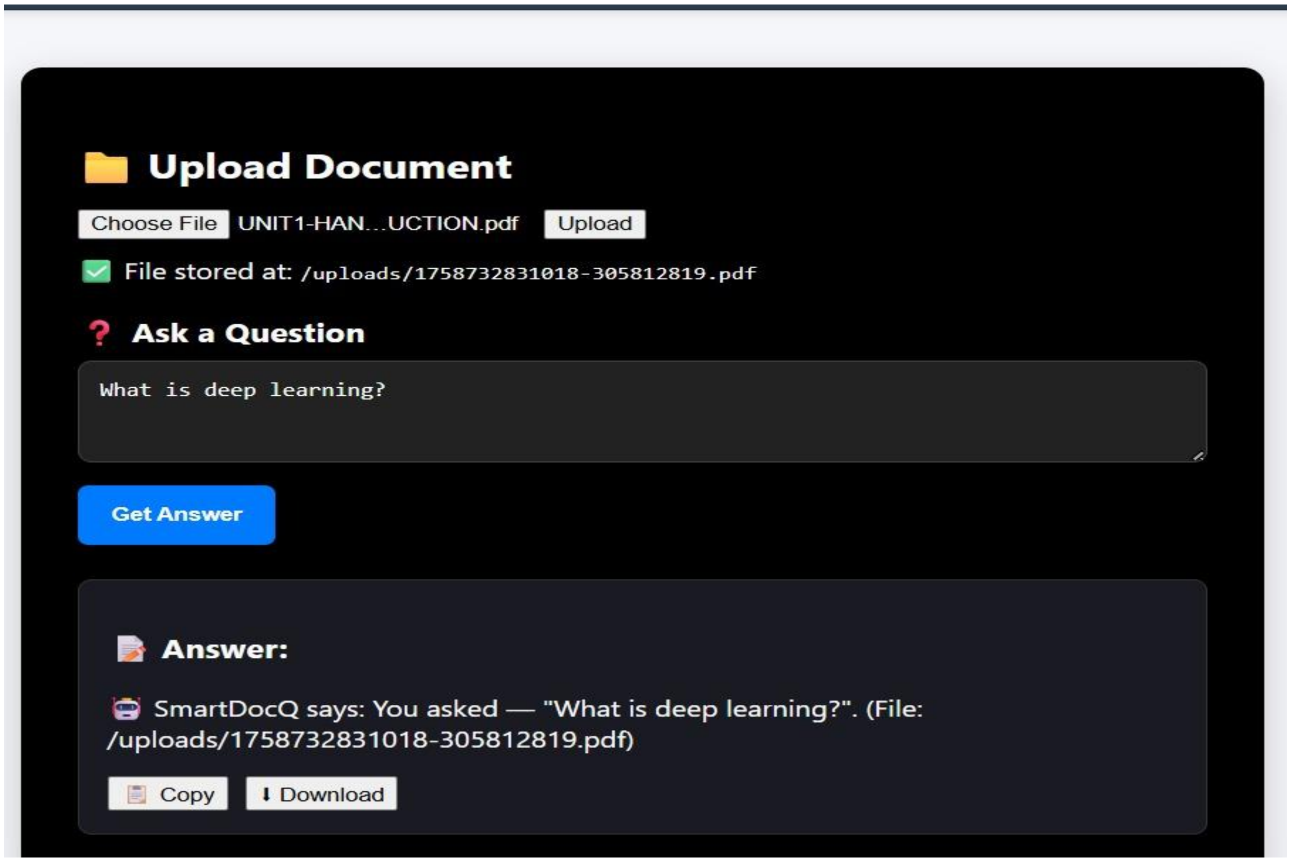
• Collected raw data relevant to the research topic.

• Cleaned and normalized the dataset (removed noise, handled missing values).

• Applied tokenization / formatting as per requirements.

• Data is now ready for feature extraction and analysis.





Upload Document

Choose File UNIT1-HAN...UCTION.pdf Upload

File stored at: /uploads/1758732831018-305812819.pdf

Ask a Question

What is deep learning?

Get Answer

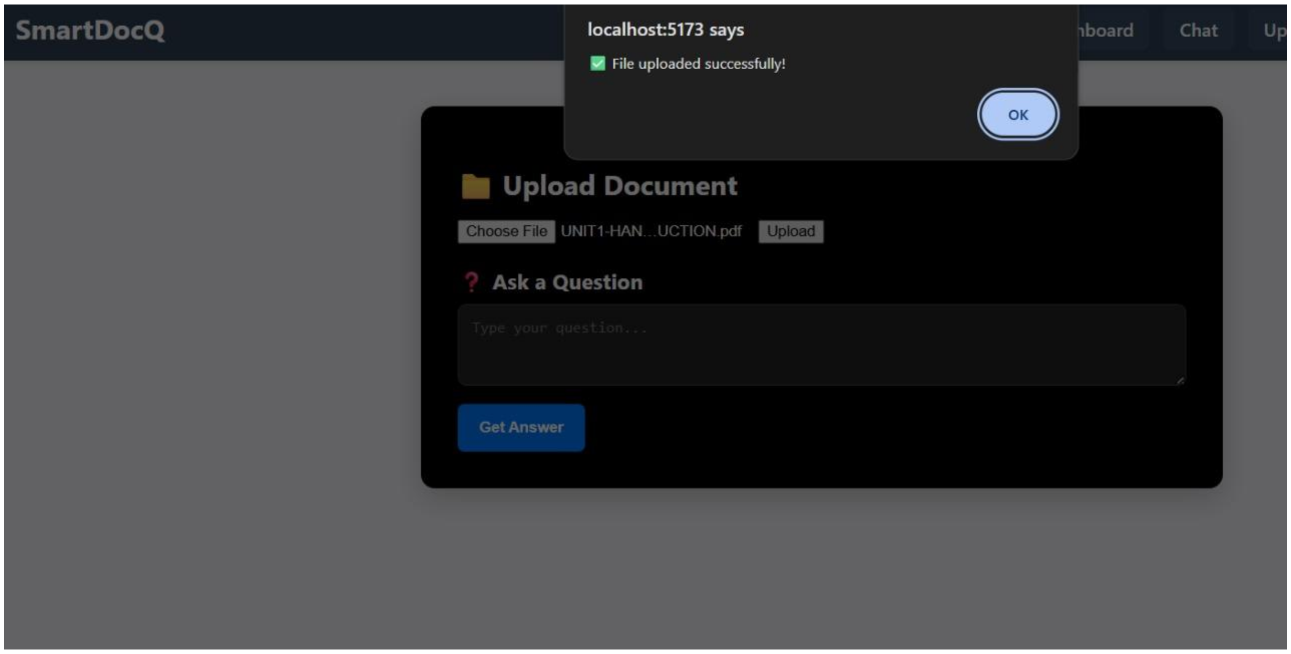
Answer:

SmartDocQ says: You asked — "What is deep learning?". (File: /uploads/1758732831018-305812819.pdf)

Copy ⬇ Download

```
import mongoose from 'mongoose';
import { config } from './config.js';

export async function connectDB() {
  mongoose.set('strictQuery', true);
  await mongoose.connect(config.mongoUri, { dbName: 'smartdocq' });
  console.log('MongoDB connected');
}
```



```
import dotenv from "dotenv";
dotenv.config();

export const config = {
  port: process.env.PORT || 5000,
  mongoUri: process.env.MONGO_URI || "mongodb://localhost:27017/smartdocq",
  clientOrigin: process.env.CLIENT_ORIGIN || "http://localhost:5173",
  jwtSecret: process.env.JWT_SECRET,
  jwtExpiresIn: process.env.JWT_EXPIRES_IN || "15m",
  refreshSecret: process.env.REFRESH_SECRET,
  refreshExpiresIn: process.env.REFRESH_EXPIRES_IN || "7d",
};
```

Server-Side Coding

1. Introduction

In Week-VI, the project moved into a crucial phase of **server-side coding**. After setting up the initial backend structure and database integration in earlier weeks, this milestone focused on strengthening the backend with **logic implementation, authentication, role management, and endpoint security**. These features ensure that SmartDocQ is not only functional but also secure and capable of supporting multiple users with different levels of access.

2. Logic Implementation

Implemented the **core backend logic** for file upload, text extraction, chunking, embedding, and query processing.
Added proper **error handling** to ensure invalid requests (unsupported file formats or empty queries) are managed gracefully.
Designed the flow so that once a document is uploaded, it is automatically chunked, embedded using Gemini API, and stored in ChromaDB for semantic search.

3. User Login and Session/Token Management

- Integrated **user authentication** into the backend using secure token-based methods.

- Each user must **log in** with valid credentials to access backend features.
- Upon successful login, the system issues a **JWT (JSON Web Token)** which acts as the session identifier.
- The token must be included in subsequent API requests to verify identity and maintain session integrity.

4. User Roles and Access Permissions

Defined **user roles** to differentiate access levels:
Admin → Has full privileges (manage users, documents, and system settings).
Standard User → Restricted access (can upload documents and ask queries but cannot manage other users).
This role-based access ensures that sensitive operations are not exposed to all users.

5. Secure Endpoints Based on User Roles

All major API endpoints (/upload, /ask, /profile) were **secured** using authentication middleware.
Each request undergoes two checks:
Authentication – Verifies the token's validity.
Authorization – Confirms if the user role has permission to access that specific endpoint.
Example: A standard user cannot access admin-only endpoints such as user management.

6. Summary

In this milestone, the backend of SmartDocQ transitioned from a basic document processing system into a **secure, role-based, and user-friendly server application**. By implementing login, session management, user roles, and endpoint security, the system is now ready for reliable frontend integration and multi-user usage.

The next steps will involve:

Expanding the authentication system with password encryption.
Enhancing role definitions (e.g., guest access).
Further testing and optimization for performance.

End-to-End Testing

Unit Testing

• Tests individual modules or functions in **isolation**.

• Helps detect **small-level bugs early**.

• Example: Testing authentication validation function, file parsing utility.

• Conducted on modules like login, file processing, and input validation.

```
import mongoose from "mongoose";

import User from "../../src/models/User.js";

describe("User Model Unit Tests", () => {

beforeAll(async () => {

await mongoose.connect("mongodb://127.0.0.1:27017/testdb");

});

afterAll(async () => {

await mongoose.connection.close();

});

it("should create a user with email and password", async () => {
```

```
const user = new User({ email: "test@example.com", password: "secure123" });

const savedUser = await user.save();

expect(savedUser._id).toBeDefined();

expect(savedUser.email).toBe("test@example.com");

});

it("should fail if email is missing", async () => {

const user = new User({ password: "12345" });

let err;

try {

await user.save();

} catch (error) {

err = error;

}

expect(err).toBeDefined();

});

});
```

Integration Testing

• Verifies how different modules **work together**.

• Ensures **data flow and interaction** between components are correct.

• Example: User login → file upload → file processed and stored.

• Performed on workflows involving authentication, business logic, and database.

```
import request from "supertest";

import app from "../../src/index.js";

describe("Auth API Integration Tests", () => {

it("should register a new user", async () => {

const res = await request(app).post("/api/auth/signup").send({

email: "integration@test.com",

password: "test123",

});

expect(res.statusCode).toBe(201);

expect(res.body.message).toBe("User registered successfully");

});
it("should login a user", async () => {

const res = await request(app).post("/api/auth/login").send({

email: "integration@test.com",

password: "test123",

});

expect(res.statusCode).toBe(200);

expect(res.body.token).toBeDefined();

});

});
```

API Testing

• Focuses on testing **endpoints** exposed by the system.

• Validates **status codes, response structure, and error handling**.

• Example: /login, /upload, /query tested with valid/invalid inputs.

• All APIs tested with tools like **Postman**

```
import { render, screen, fireEvent, waitFor } from "@testing-library/react";

import UploadPage from "../pages/UploadPage";
import { rest } from "msw";

import { setupServer } from "msw/node";const server = setupServer(

rest.post("/api/files/upload", (req, res, ctx) => {

return res(ctx.json({ message: "File uploaded successfully" }));

})

);

beforeAll(() => server.listen());

afterEach(() => server.resetHandlers());

afterAll(() => server.close());

test("uploads file and shows success message", async () => {

render(<UploadPage />);

const fileInput = screen.getByLabelText(/upload file/i);

const file = new File(["dummy content"], "test.pdf", { type: "application/pdf" });

fireEvent.change(fileInput, { target: { files: [file] } });

fireEvent.click(screen.getByText(/upload/i));

await waitFor(() => {
```

```
expect(screen.getByText("File uploaded successfully")).toBeInTheDocument();

});

});
```

Mocking External Services

• **Simulates third-party services** instead of using real ones.• Useful when real external systems are **unavailable or costly**.

• Example: Mocking cloud storage or third-party API calls.

• Mock services used for safe and controlled testing of dependent modules.

```
import { getAnswerFromGemini } from "../../src/services/geminiService.js";

jest.mock("../../src/services/geminiService.js", () => ({

getAnswerFromGemini: jest.fn(),

}));

describe("Gemini Service Mocking", () => {

it("should return mocked response instead of real Gemini call", async () => {

getAnswerFromGemini.mockResolvedValue("Mocked Gemini Response");

const response = await getAnswerFromGemini("What is AI?");

expect(response).toBe("Mocked Gemini Response");
});

});
```

• Using google gemini api

Test Type	Test Case / Feature	Predicted Outcome	Actual Outcome	Status
API Testing	POST /api/auth/login – Verify user authentication with valid credentials	Should return JWT token and user details	Returned token and user details correctly	Pass
API Testing	POST /api/upload – Upload document to	Should return success	File uploaded successfully with	Pass

Test Type	Test Case / Feature	Predicted Outcome	Actual Outcome	Status
	server	message and file URL	correct URL	
Unit Testing	tokenValidator() – Validate JWT token	Should return “true” for valid token and “false” for invalid token	Function behaved as expected for both cases	Pass
Unit Testing	chunkText() – Splitting large text into semantic chunks	Should return array of chunked strings with proper boundaries	Output matched expected chunk sizes	Pass

Test Type	Test Case / Feature	Predicted Outcome	Actual Outcome	Status
Integration Testing	Upload → Process → Q&A workflow	Upload document → Embed → Ask question → Get AI response	Entire flow worked correctly and responses matched document content	Pass
Integration Testing	Login → Access protected route (/api/history)	Authenticated users should access route, others denied	Protected route properly secured	Pass
Mocking External Services	Mock Gemini API responses during offline tests	Should return dummy responses mimicking Gemini output	Mocked responses successfully replaced real API calls	Pass

Test Type	Test Case / Feature	Predicted Outcome	Actual Outcome	Status
Mocking External Services	Mock MongoDB connection during backend test	Should simulate DB read/write without affecting real data	Mocked database worked as expected	Pass

Setup Environmental Settings for both staging and production

1. Introduction

The focus of Week 11 was on system validation through detailed test case creation and execution.
At this stage, the SmartDocQ system was functionally complete — both backend APIs and frontend modules had been integrated and deployed.
The main goal was to ensure that the system performs correctly, consistently, and is stable after multiple updates.

2. Objectives

- To design and document test cases for validating each system function.
- To perform UI Testing and confirm that the interface works as expected.
- To conduct Regression Testing after recent code modifications.
- To update all test cases in the Software Requirements Specification (SRS).
- To record and prepare a video demonstration of the working project.

3. Test Case Design

Test cases were created based on the modules defined in SRS.
Each test case includes:

- Test ID
- Test Objective
- Input
- Expected Output
- Actual Result
- Status (Pass/Fail)

The test cases covered:

- Authentication Module – Login, Signup, and Logout
- Chat/Query Module – Sending user queries and receiving backend responses
- Database Operations – Data insertion, retrieval, and validation

- Navigation – Page routing and transitions between components

4. UI Testing

The UI testing was performed manually using browser-based inspection tools and test scripts. The objective was to ensure that the application's visual elements and user flows behave as expected.

Test ID	Description	Expected Result	Actual Result	Status
UI-01	Verify login form fields load properly	All fields visible	Working as expected	Pass
UI-02	Check responsiveness across devices	Layout adjusts correctly	Verified on desktop and mobile	Pass
UI-03	Validate button click navigation	Buttons redirect correctly	Working fine	Pass
UI-04	Check error message on invalid input	Error message shown	Working fine	Pass

5. Regression Testing

After implementing minor fixes and optimizations in Week 10, regression testing was conducted to ensure that new changes did not affect existing functionality.
Modules tested again:

- User authentication
- Chat and API response handling
- Navigation and route guards
- Database connectivity

Result: All previously working functionalities continued to perform correctly, confirming system stability.

6. Updating SRS

The Software Requirements Specification (SRS) document was updated to include:

- New test case tables
- Revised module flow diagrams
- Changes in API endpoints (if any)
- Updated screenshots of the latest UI build

7. Solution Demo Video

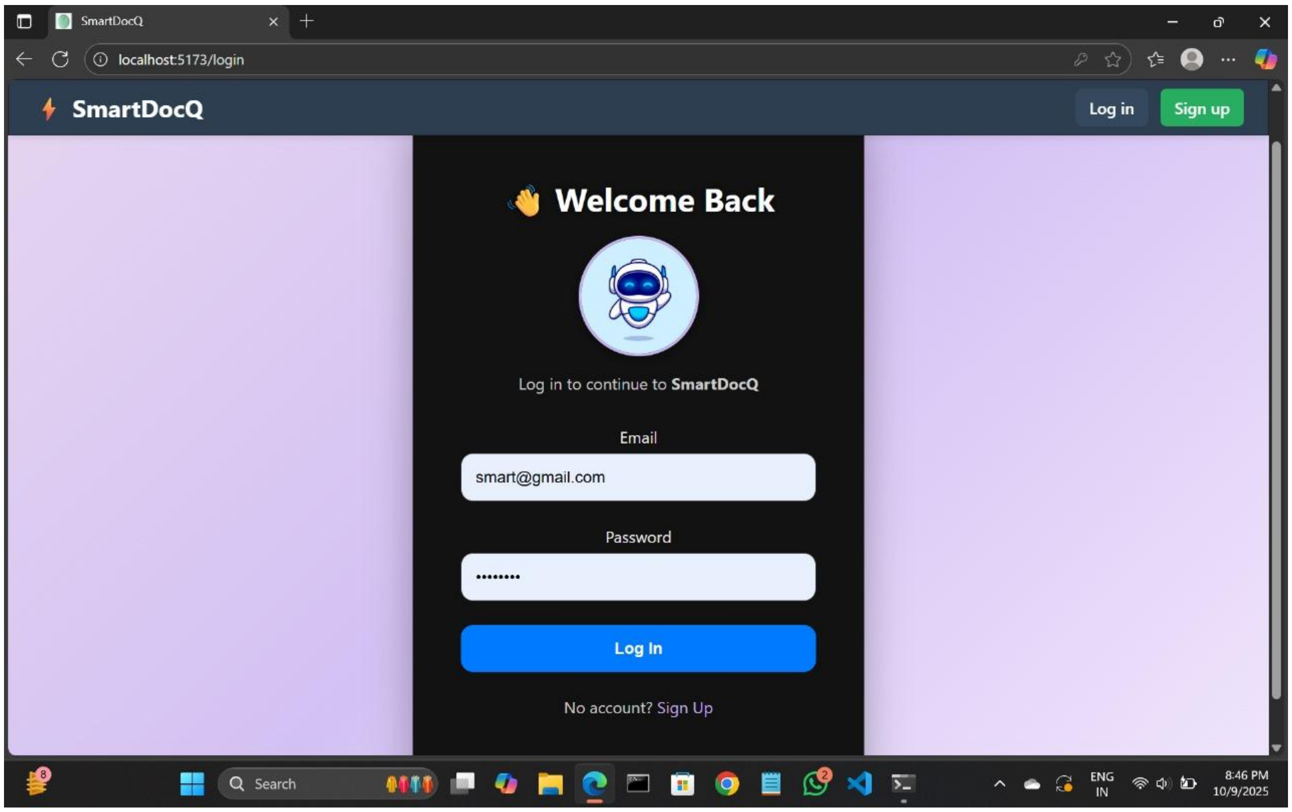
A final solution demo video was recorded showcasing:

1. User login and signup process.
2. Navigation through different pages.
3. Sending a query/message through the chat interface.
4. Backend response displayed on screen.
5. MongoDB database reflecting the user activity.

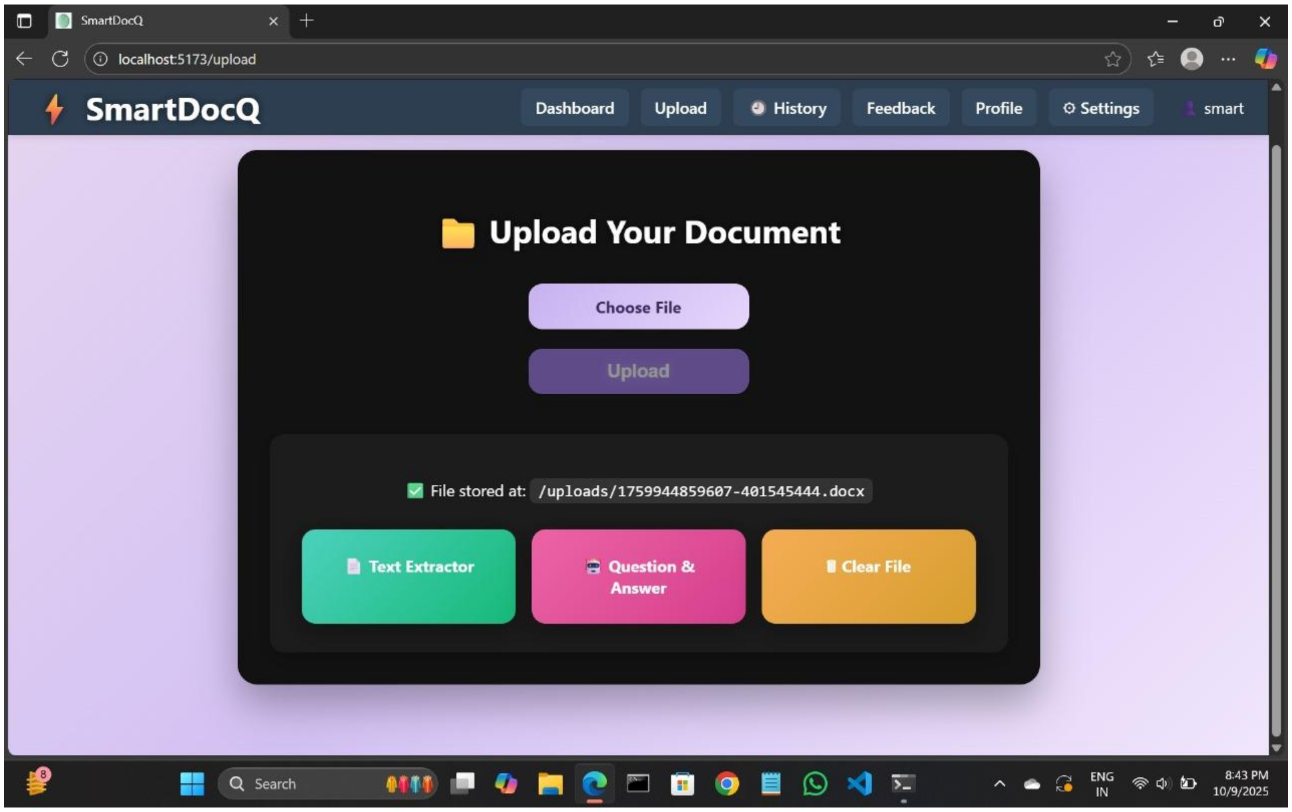
8. Observations

- The UI remained consistent after multiple deployments.
- Test coverage improved as test cases were aligned with all functional requirements.
- Minor layout adjustments were required for small-screen devices.
- No critical defects were found during regression testing.

MOCKSCREENS

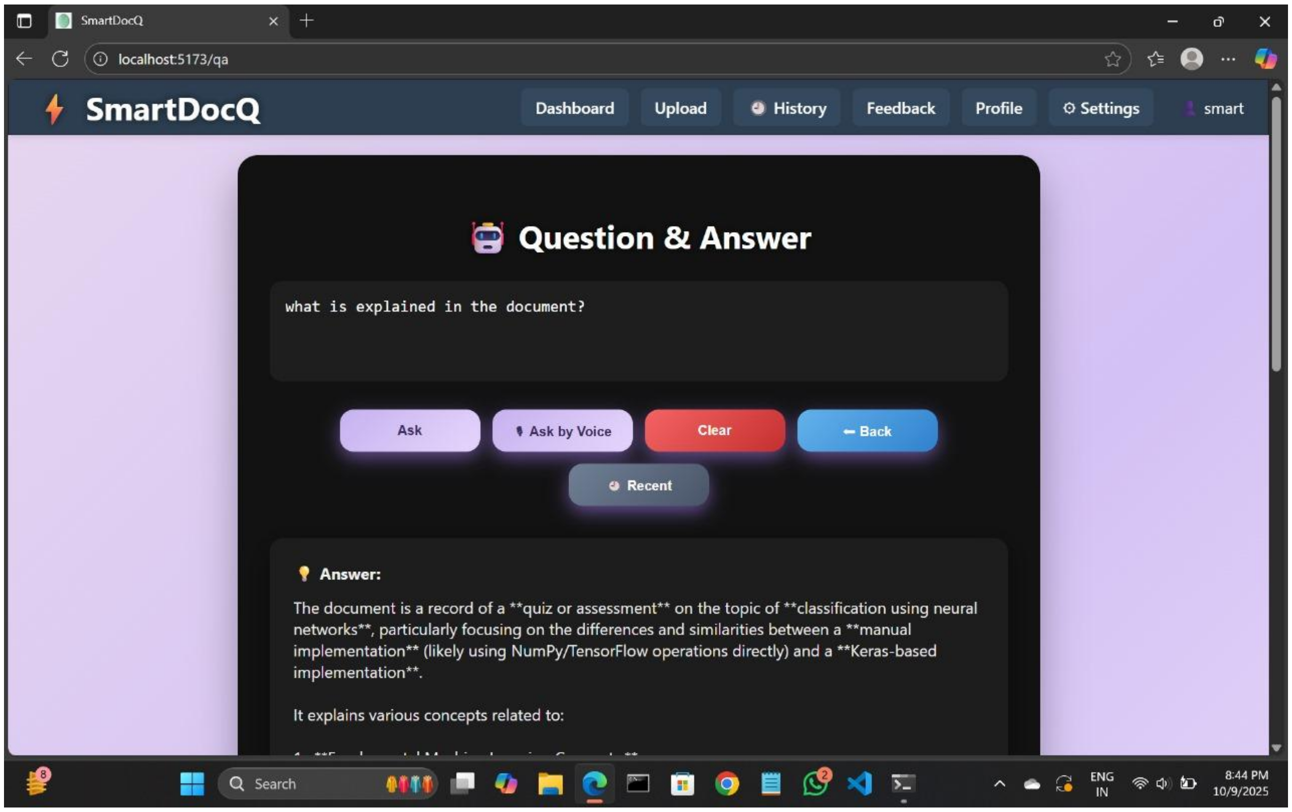


This screen allows users to securely create a new account or log in to access the SmartDocQ platform.



File stored at: /uploads/1759944859607-401545444.docx

This screen allows users to upload documents (PDF, DOCX, or TXT) for text extraction and further processing

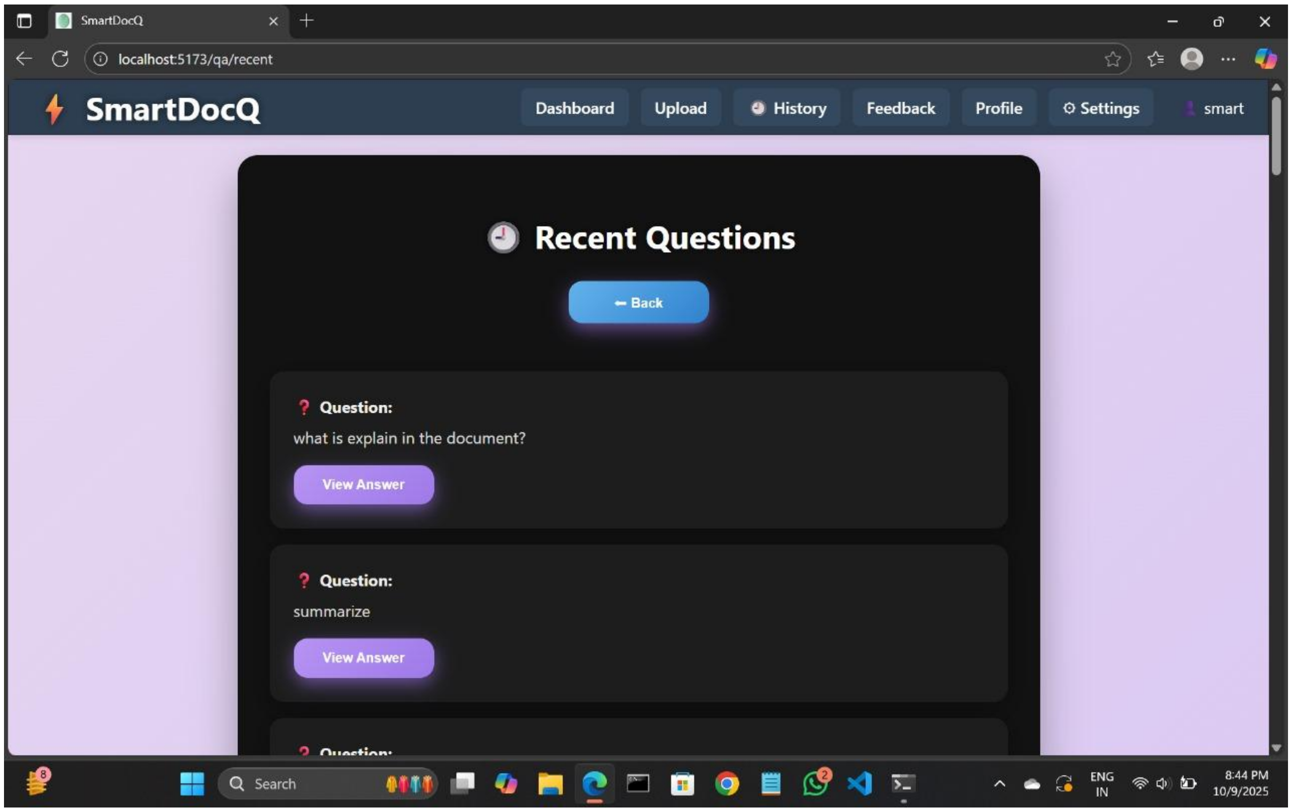


what is explained in the document?

The document is a record of a **quiz or assessment** on the topic of **classification using neural networks**, particularly focusing on the differences and similarities between a **manual implementation** (likely using NumPy/TensorFlow operations directly) and a **Keras-based implementation**.

It explains various concepts related to:

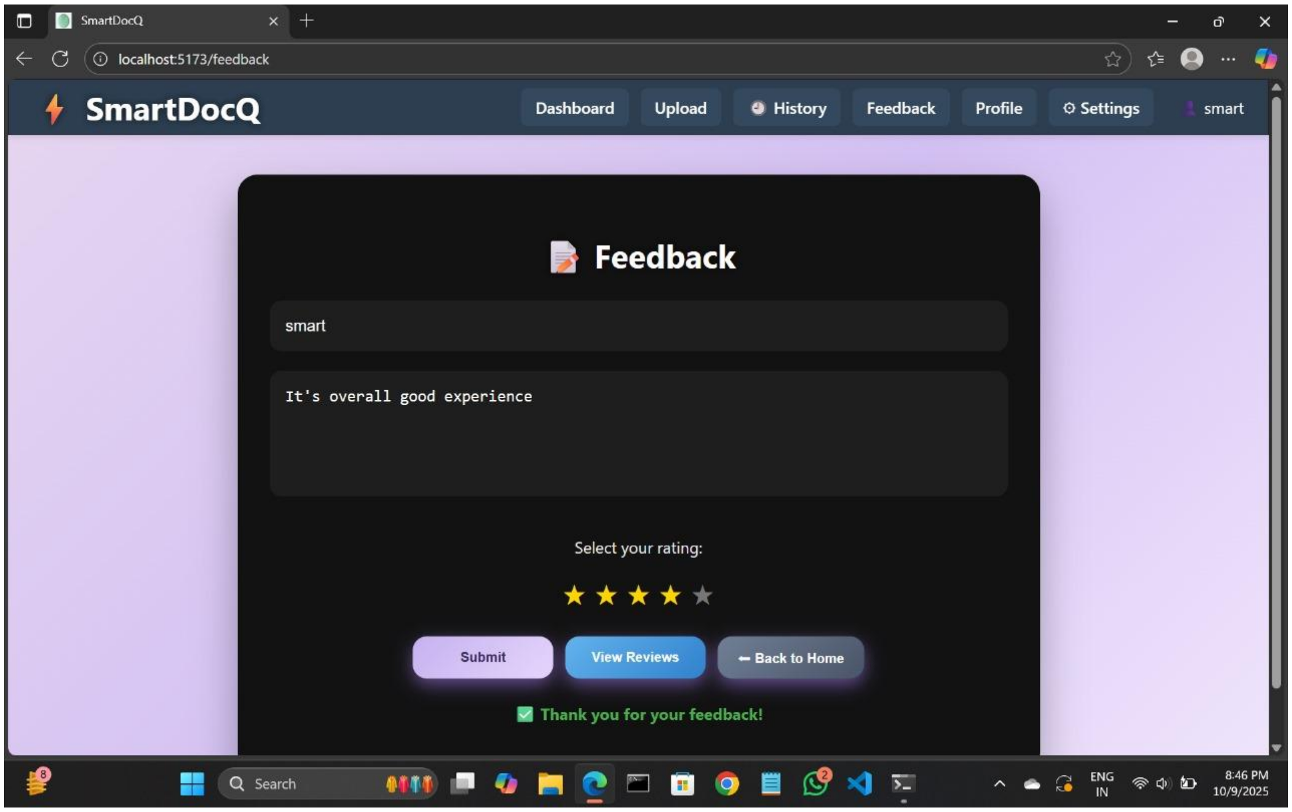
This screen allows users to ask questions related to their uploaded documents and receive AI-powered answers using the integrated chatbot.



❓ Question:
what is explain in the document?

❓ Question:
summarize

This screen displays the user’s recently asked questions along with their answers for quick reference.



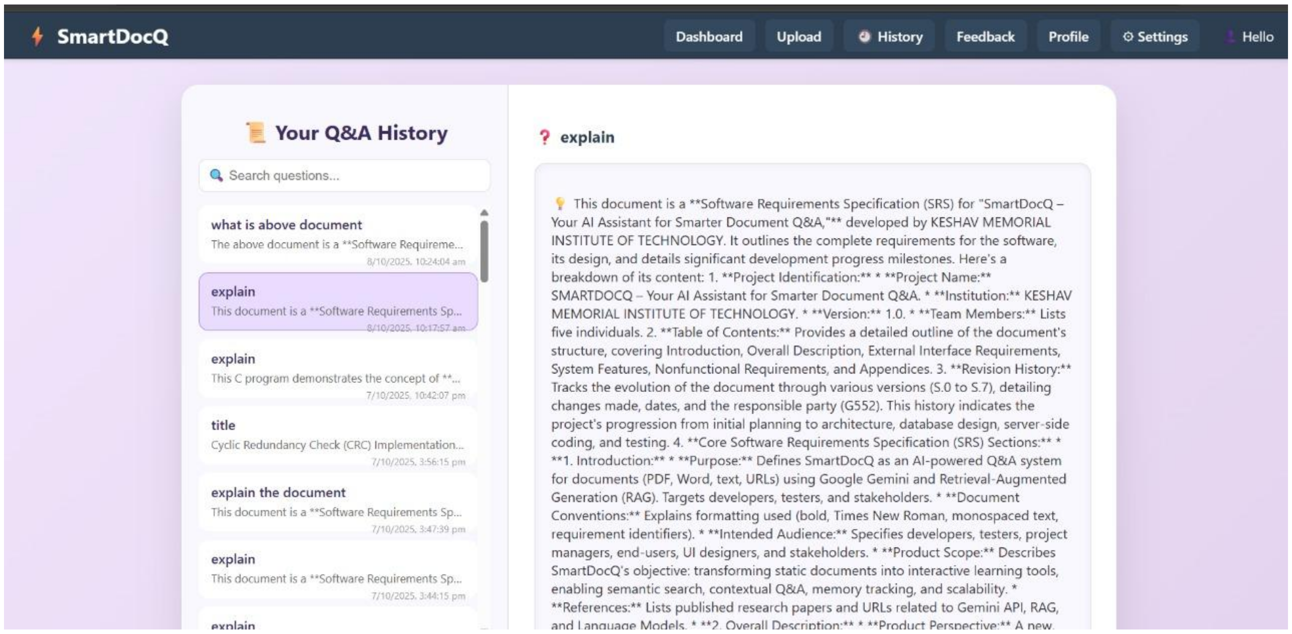
smart

It's overall good experience

★★★★☆

✅ Thank you for your feedback!

This screen enables users to share their feedback or suggestions to help improve the app’s features and user experience.



Your Q&A History

Search questions...

what is above document
The above document is a **Software Requireme...
8/10/2025, 10:24:04 am

explain
This document is a **Software Requirements Sp...
8/10/2025, 10:17:57 am

explain
This C program demonstrates the concept of **...
7/10/2025, 10:42:07 pm

title
Cyclic Redundancy Check (CRC) Implementation...
7/10/2025, 3:56:15 pm

explain the document
This document is a **Software Requirements Sp...
7/10/2025, 3:47:39 pm

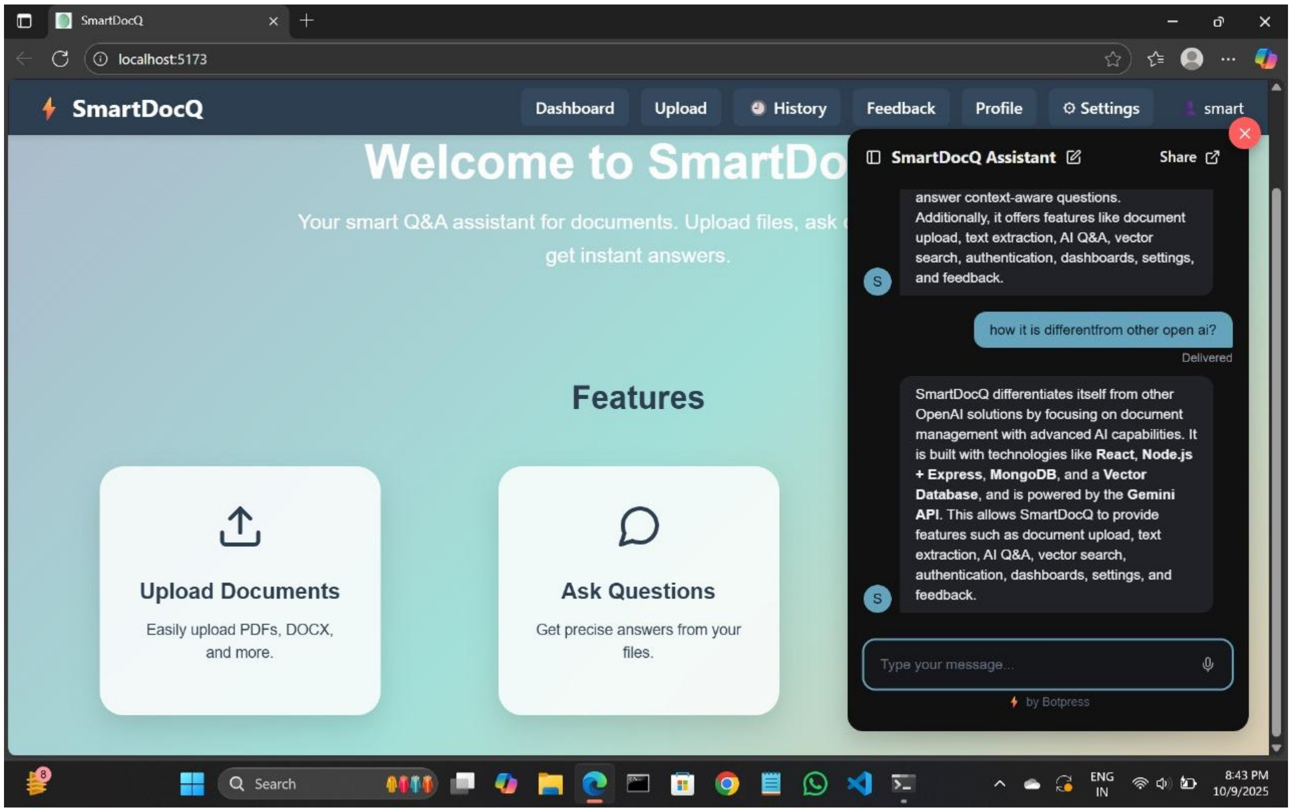
explain
This document is a **Software Requirements Sp...
7/10/2025, 3:44:15 pm

explain

explain

This document is a **Software Requirements Specification (SRS) for "SmartDocQ – Your AI Assistant for Smarter Document Q&A,"** developed by KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY. It outlines the complete requirements for the software, its design, and details significant development progress milestones. Here's a breakdown of its content: 1. **Project Identification:** * **Project Name:** SMARTDOCQ – Your AI Assistant for Smarter Document Q&A. * **Institution:** KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY. * **Version:** 1.0. * **Team Members:** Lists five individuals. 2. **Table of Contents:** Provides a detailed outline of the document's structure, covering Introduction, Overall Description, External Interface Requirements, System Features, Nonfunctional Requirements, and Appendices. 3. **Revision History:** Tracks the evolution of the document through various versions (S.0 to S.7), detailing changes made, dates, and the responsible party (G552). This history indicates the project's progression from initial planning to architecture, database design, server-side coding, and testing. 4. **Core Software Requirements Specification (SRS) Sections:** * **1. Introduction:** * **Purpose:** Defines SmartDocQ as an AI-powered Q&A system for documents (PDF, Word, text, URLs) using Google Gemini and Retrieval-Augmented Generation (RAG). Targets developers, testers, and stakeholders. * **Document Conventions:** Explains formatting used (bold, Times New Roman, monospaced text, requirement identifiers). * **Intended Audience:** Specifies developers, testers, project managers, end-users, UI designers, and stakeholders. * **Product Scope:** Describes SmartDocQ's objective: transforming static documents into interactive learning tools, enabling semantic search, contextual Q&A, memory tracking, and scalability. * **References:** Lists published research papers and URLs related to Gemini API, RAG, and Language Models. * **2. Overall Description:** * **Product Perspective:** A new

A detailed summary screen displaying the user’s past transactions and activities, allowing easy tracking, filtering, and review of financial history in a clear, organized layout.



answer context-aware questions. Additionally, it offers features like document upload, text extraction, AI Q&A, vector search, authentication, dashboards, settings, and feedback.

how it is differentfrom other open ai?

Delivered

SmartDocQ differentiates itself from other OpenAI solutions by focusing on document management with advanced AI capabilities. It is built with technologies like **React**, **Node.js + Express**, **MongoDB**, and a **Vector Database**, and is powered by the **Gemini API**. This allows SmartDocQ to provide features such as document upload, text extraction, AI Q&A, vector search, authentication, dashboards, settings, and feedback.

This screen allows users to interact with an AI-powered chatbot that answers queries and provides assistance related to uploaded documents or general questions.

References

The following published research papers, technical documentation, and online standards have been referenced during the development of SmartDocQ – Your AI Assistant for Smarter Document Q&A:

1**) Leveraging Gemini API**

Published by International Research Journal of Modernization in Engineering Technology and Science (IRJMETS)

URL:https://www.irjmets.com/uploadedfiles/paper//issue_5_may_2024/55847/final/fin_irjmets1715768590.pdf

2)**Lewis, Patrick, et al.**

Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.

Proceedings of NeurIPS 2020.

URL: https://arxiv.org/abs/2005.11401

3)**Brown, Tom, et al.**

Language Models are Few-Shot Learners.

NeurIPS 2020.

URL: https://arxiv.org/abs/2005.14165

Conclusion

The SmartDocQ project successfully demonstrates how artificial intelligence can be integrated with document management to create an intelligent, user-friendly question-answering platform.
By allowing users to upload documents, extract text, and ask contextual questions, SmartDocQ simplifies the process of information retrieval and document comprehension.

The system uses advanced language models (like Google Gemini) to provide accurate and context-aware answers, while maintaining a clean and modern interface built with React.js and a reliable MongoDB–Express–Node.js backend.

- Through this project, we achieved:
- Efficient document parsing for PDFs and DOCX files
- Instant Q&A responses using AI integration
- Secure data storage and retrieval in MongoDB
- An intuitive, responsive frontend interface
- User authentication and personalized experience

Overall, SmartDocQ serves as a practical example of how AI can enhance productivity in handling textual data. It bridges the gap between traditional document storage and smart information access, paving the way for more intelligent enterprise knowledge systems.

Feature Enhancements / Future Scope

Mobile Application

Develop an Android/iOS version of SmartDocQ for on-the-go document access and question answering.

Multi-File Knowledge Integration

Enable querying across multiple uploaded documents simultaneously, creating a unified knowledge base for users.

Enhanced AI Models

Integrate more advanced AI models or fine-tune custom models to improve accuracy and handle domain-specific documents (e.g., legal, medical, academic).

Cloud Storage Integration

Connect with Google Drive, Dropbox, or OneDrive for seamless document upload and synchronization.

Collaborative Q&A Dashboard

Allow multiple users to share documents and collaborate on queries in real-time.

Improved Security

Add encryption and role-based authentication to protect sensitive document data.

Multilingual Support

Extend Q&A capabilities to support multiple languages, enabling global accessibility.

Analytics and Insights

Introduce analytics to show frequently asked questions, most accessed documents, and user activity trends.