MERN Document Search and OCR System

(A Full-Stack Web Application for Intelligent Document Processing and Text-Based Search using Optical Character Recognition)

A PROJECT REPORT

Submitted by

Armaan Atri

in partial fulfilment for the award of the degree of Bachelors of Engineering

Computer Science Engineering

IN

Artificial Intelligence and Machine Learning

Chandigarh University

November 2025

Project Title

Document Search and OCR using MERN Stack

Project Description

The **MERN Document Search and OCR System** is a full-stack web application designed to facilitate intelligent document management through automated Optical Character Recognition (OCR) and full-text search functionalities. The system enables users to upload, process, and search documents in real time, making it a powerful tool for organizations, researchers, and individuals handling large volumes of digital files.

This project leverages the **MERN stack**—**MongoDB, Express.js, React.js, and Node.js**—to provide a seamless and efficient document processing environment. The core objective of the system is to simplify the way users interact with digital documents by automatically extracting text from uploaded files, indexing it for efficient searching, and presenting it through an intuitive and responsive user interface.

System Overview

The system allows users to upload various document types including PDFs, images, and text files. Upon upload, the backend processes these files using OCR (Optical Character Recognition) to extract readable text. Each document undergoes multiple stages — queuing, processing, and indexing — which are tracked and updated in real time using **WebSocket communication** (Socket.IO). Users can then search across all processed documents, view specific pages, and manage uploaded files through a centralized dashboard.

Key Features

1. **Document Upload and Management:**
   Users can upload multiple document formats (PDF, JPG, PNG, TXT) which are stored securely and linked with metadata such as upload time, file size, and processing status.

2. **OCR Processing:**
   The system integrates OCR functionality to automatically extract text content from uploaded documents. Although the current implementation uses a mock OCR process for demonstration, it can be extended to use **Tesseract.js** or cloud-based OCR APIs in a production environment.

3. **Real-Time Job Tracking:**
   The use of **Socket.IO** enables live updates of document processing status, progress percentage, and notifications on completion or failure.

4. **Full-Text Search:**
   Users can perform search queries across all processed documents. The search results display matched text excerpts, allowing users to directly open and review relevant content.

5. **Document Viewing and Navigation:**
   Each processed document can be viewed through the frontend, showing extracted text page by page for easier reading and verification.

6. **Queue Management:**
   The system employs an in-memory queue mechanism to manage multiple OCR jobs efficiently. For large-scale deployments, this can be replaced with **Redis-based** or **message queue systems** for distributed task handling.

Technology Stack

- **Frontend:** React 18 with Vite, Tailwind CSS, React Router, Socket.IO Client, Axios
- **Backend:** Node.js with Express, MongoDB with Mongoose, Socket.IO, Multer, Helmet, and CORS
- **Database:** MongoDB (local or cloud instance) for storing document metadata, job status, and extracted text
- **Communication Protocol:** WebSockets for real-time progress tracking
- **Development Tools:** Nodemon, ESLint, Vite, and Postman for API testing

Objectives

- To develop a user-friendly system for document upload, processing, and retrieval.
- To integrate OCR technology for automatic text extraction from documents.
- To implement real-time job monitoring and notifications using WebSockets.
- To provide a full-text search feature for easy information retrieval.
- To ensure scalability, security, and performance for handling large document volumes.

Outcome

The successful implementation of this project results in a **robust and interactive web-based document management system** capable of handling complex OCR and search operations with efficiency and accuracy. It demonstrates how the MERN stack can be used to build scalable, real-time web applications and provides a foundation for future enhancements such as cloud storage integration, advanced OCR accuracy using machine learning, and document summarization.

Hardware/Software Requirements

The development and deployment of the **MERN Document Search and OCR System** require both hardware and software resources to ensure efficient performance and smooth operation. The following section outlines the **minimum and recommended configurations** for the project's environment setup.

1. Hardware Requirements

Component	Minimum Configuration	Recommended Configuration
Processor (CPU)	Intel Core i3 (10th Gen) or equivalent	Intel Core i5/i7 (12th Gen) or AMD Ryzen 5/7
RAM	4 GB	8 GB or higher
Storage	10 GB free disk space	20 GB or higher (SSD preferred)
Display	1366 × 768 resolution	1920 × 1080 (Full HD)
Network	Stable internet connection	High-speed broadband connection
GPU (Optional)	Integrated graphics	Dedicated GPU (for OCR acceleration if required)

Explanation:

The system's backend processes OCR and text extraction tasks that can be computationally intensive depending on file size and OCR model used. Higher RAM and faster processors enhance real-time processing and server response times.

2. Software Requirements

Category	Software / Tool	Purpose
Operating System	Windows 10 / 11, Linux (Ubuntu 20.04+), or macOS	Development and deployment environment
Backend Runtime	Node.js (v18 or higher)	Server-side JavaScript runtime environment
Frontend Framework	React.js (v18) with Vite	Building dynamic and modular frontend UI
Database	MongoDB (v6.0 or higher)	NoSQL database for storing document metadata and text
Web Framework	Express.js	REST API and routing management
Real-time Communication	Socket.IO	Real-time progress updates between client and server
OCR Library	Tesseract.js (optional for actual OCR)	Text extraction from images and documents
Package Manager	npm or yarn	Managing dependencies and build scripts
Styling Framework	Tailwind CSS	Frontend styling and responsive layout design
Version Control	Git and GitHub	Source code management and version tracking
Security Middleware	Helmet.js and CORS	API security and cross-origin request handling
Development Tools	Visual Studio Code / WebStorm	Integrated development environment
API Testing Tool	Postman	Testing and validating backend APIs
Browser	Google Chrome / Edge / Firefox	Running and testing frontend application

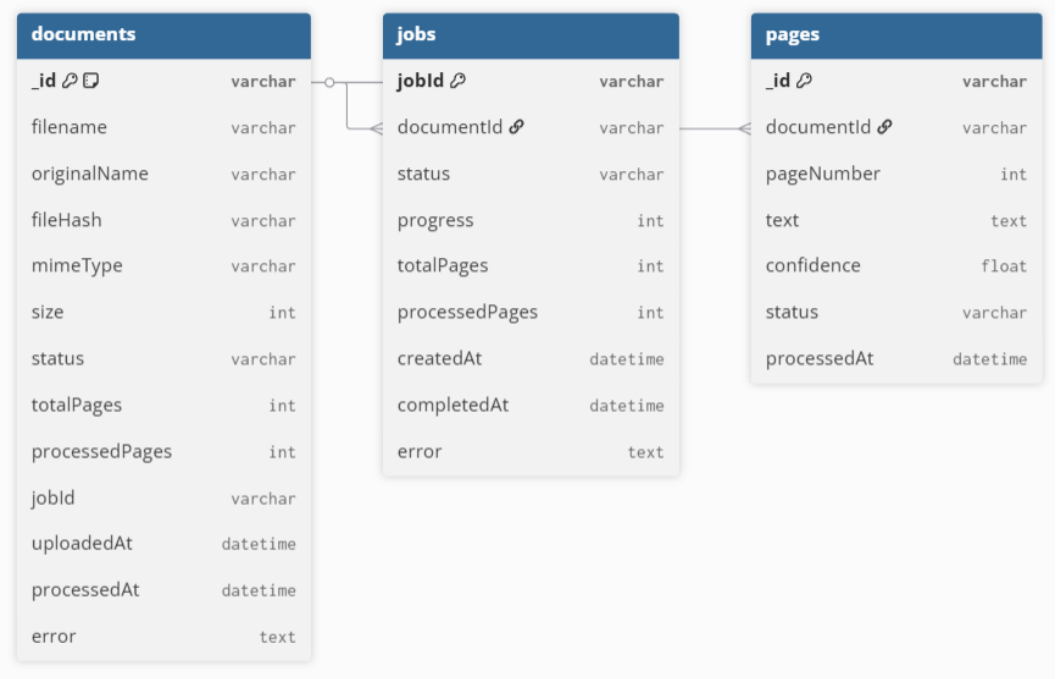
3. Additional Dependencies

- **Multer:** For handling file uploads in the backend.
- **Axios:** For frontend API calls.
- **Vite:** For fast frontend bundling and hot module reloading.
- **Nodemon:** For automatic backend server restarts during development.
- **dotenv:** For managing environment variables securely.
- **Rate Limiter:** For API request control and security.

4. System Setup Requirements

- **MongoDB Connection:** A local or cloud MongoDB instance (e.g., MongoDB Atlas).
- **Environment Variables:** Proper configuration of .env files in both frontend and backend for URLs, ports, and database connection strings.
- **Node Modules Installation:** All dependencies installed using npm install.
- **Uploads Directory:** Creation of uploads/ folder in backend for storing uploaded files

ER Diagram



The **Entity–Relationship (ER) Diagram** for the *MERN Document Search and OCR System* illustrates the logical structure of the database and the relationships among the key entities: **Document**, **Job**, and **Page**.

The system is designed around the document processing workflow — from file upload, OCR extraction, and page-wise text storage to real-time job tracking. The ER diagram represents how data is interconnected to support these operations efficiently.

1. Document Entity

The **Document** entity is the **central component** of the system. It stores metadata and processing details about each uploaded document.
Each record in this entity represents a single document file uploaded by the user.

Attributes:

- filename – The internal file name stored on the server.
- originalName – The original file name as uploaded by the user.
- fileHash – A unique hash value to prevent duplicate uploads.
- mimeType – The type of the uploaded file (e.g., PDF, JPG, PNG).
- size – The file size in bytes.
- status – Indicates the processing state (pending, processing, completed, failed).
- totalPages – Total number of pages in the document.
- processedPages – Number of pages successfully processed.
- jobId – A reference to the job handling OCR processing.
- uploadedAt and processedAt – Timestamps for upload and completion.
- error – Stores any processing error message.

Each document can have one **Job** associated with it and multiple **Pages** resulting from OCR text extraction.

2. Job Entity

The **Job** entity represents the **OCR processing task** assigned to each uploaded document. It is responsible for tracking the progress, completion status, and errors during text extraction.

Attributes:

- jobId – Unique identifier for each OCR job.
- documentId – Foreign key linking the job to its document.
- status – Indicates the state of the job (waiting, active, completed, failed).
- progress – Numeric representation of job progress (in percentage).
- totalPages and processedPages – Tracks how many pages were successfully processed.
- createdAt and completedAt – Timestamps for job creation and completion.
- error – Stores any errors encountered during processing.

Relationship:
Each **Document** has **one Job** associated with it (1 : 1 relationship).
This ensures that every uploaded document has its own unique processing task.

3. Page Entity

The **Page** entity represents the **individual pages** of a processed document. Each page stores the extracted text, OCR confidence level, and status.

Attributes:

- pageNumber – Sequential page number within the document.
- text – The OCR-extracted text content.
- confidence – OCR confidence score (accuracy estimate).
- status – Indicates processing status for the page (completed, failed, etc.).
- processedAt – Timestamp indicating when the page was processed.

Relationship:
Each **Document** can have multiple **Pages** (1 : N relationship).
This structure allows for efficient storage and retrieval of page-level text data.

4. Relationships Summary

Relationship	Type	Description
Document → Job	1 : 1	Each document is assigned exactly one OCR processing job.
Document → Page	1 : N	A single document can have multiple pages extracted and stored individually.
Job → Page	Indirect	A job processes multiple pages indirectly through its linked document.

5. Overall System Flow (via Entities)

1. **User Uploads Document** → A new record is created in the **Document** collection.
2. **OCR Job Created** → A **Job** entity is initiated and linked to the corresponding document.
3. **Page Extraction** → OCR processing divides the document into **Pages**, creating multiple records in the **Page** collection.
4. **Real-Time Tracking** → The job's progress updates are emitted to the frontend via **Socket.IO**.
5. **Search and Retrieval** → Extracted text stored in the **Page** collection is indexed and can be searched through the **Search API**.

Database Schema

The database schema of the **MERN Document Search and OCR System** defines the structure of data stored in the **MongoDB** database.
MongoDB, being a NoSQL database, stores data in the form of **collections** and **documents** rather than traditional relational tables and rows.
Each document is represented in **JSON-like format**, which allows flexible and dynamic data storage while maintaining relationships through object references.

The system consists of **three main collections** — **Documents**, **Jobs**, and **Pages** — that collectively manage the lifecycle of document uploads, OCR processing, and text extraction.

1. Document Collection

This collection stores metadata and status information for each uploaded document.

Schema Definition (Document.js)

```
{
  filename: String,
  originalName: String,
  fileHash: String,
  mimeType: String,
  size: Number,
  status: String,        // 'pending', 'processing', 'completed', 'failed'
  totalPages: Number,
  processedPages: Number,
  jobId: String,
  uploadedAt: Date,
  processedAt: Date,
  error: String
}
```

Explanation

- filename — The internal name assigned to the uploaded file on the server.
- originalName — The name of the file as uploaded by the user.
- fileHash — A unique hash value generated to detect duplicate files.
- mimeType — Specifies the document format (e.g., application/pdf, image/jpeg).
- size — Indicates the size of the file in bytes.
- status — Denotes the current processing stage of the document.
- totalPages and processedPages — Track the total and processed page count.
- jobId — References the Job collection to identify which job processed this document.

- uploadedAt and processedAt — Store timestamps of upload and processing completion.
- error — Logs any error encountered during OCR or upload.

2. Job Collection

This collection tracks OCR processing tasks assigned to documents.
Each job corresponds to one document and stores its progress and completion status.

Schema Definition (Job.js)

```
{
 jobId: String,
 documentId: ObjectId,  // Reference to Document
 status: String,        // 'waiting', 'active', 'completed', 'failed'
 progress: Number,
 totalPages: Number,
 processedPages: Number,
 createdAt: Date,
 completedAt: Date,
 error: String
}
```

Explanation

- jobId — Unique identifier for each job.
- documentId — References the associated document being processed.
- status — Represents the current state of processing.
- progress — Indicates the percentage of processing completed.
- totalPages and processedPages — Provide progress tracking at the page level.
- createdAt and completedAt — Store timestamps for job creation and completion.
- error — Describes any error encountered during the OCR process.

3. Page Collection

The Page collection stores text extracted from each individual page of a processed document.
It links back to the document via the documentId field and may contain OCR confidence scores.

Schema Definition (Page.js)

```
{
 documentId: ObjectId,   // Reference to Document
 pageNumber: Number,
 text: String,
```

```
  confidence: Number,
  status: String,        // 'pending', 'processed', 'failed'
  processedAt: Date
}
```

Explanation

- documentId — References the parent document.
- pageNumber — Identifies the page number within the document.
- text — Stores the OCR-extracted text content.
- confidence — Represents the confidence level of OCR accuracy.
- status — Indicates processing completion for each page.
- processedAt — Stores the timestamp of page processing.

4. Relationships Between Collections

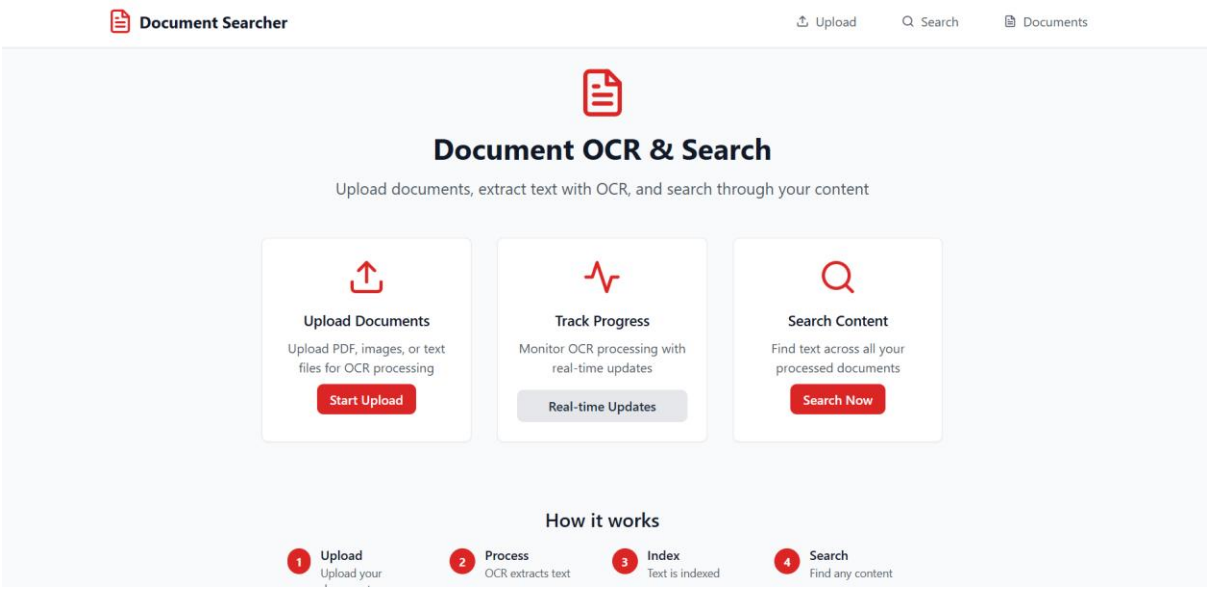
Collection	Relationship	Description
Document → Job	1 : 1	Each document has one corresponding job for OCR processing.
Document → Page	1 : N	A single document can contain multiple pages of extracted text.
Job → Page	Indirect (via Document)	Each job processes multiple pages linked through the document reference.

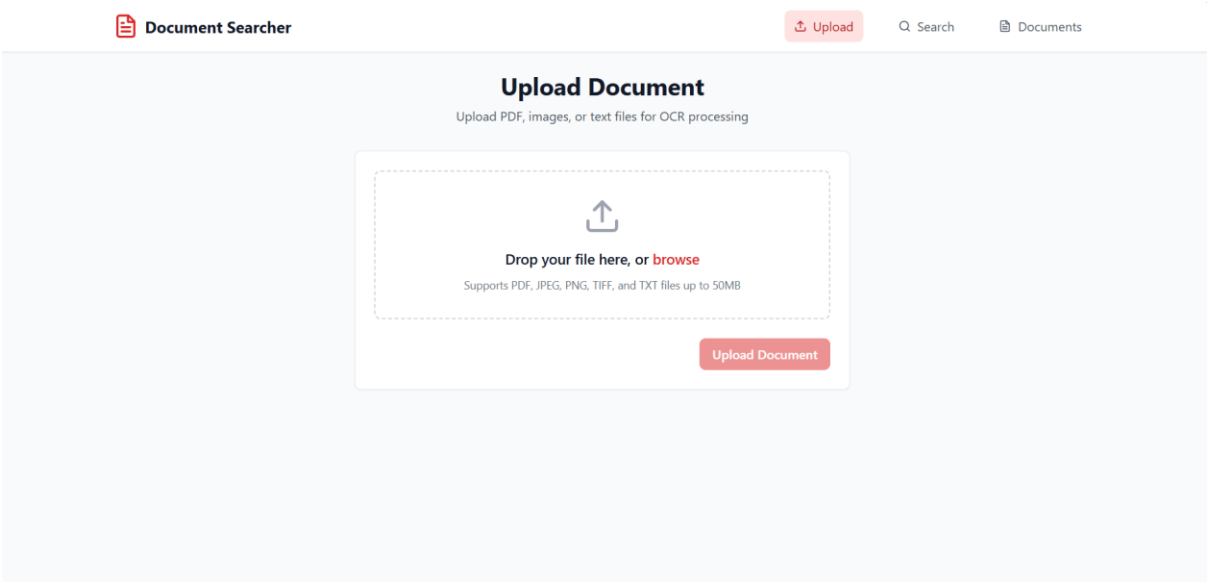
5. Database Design Justification

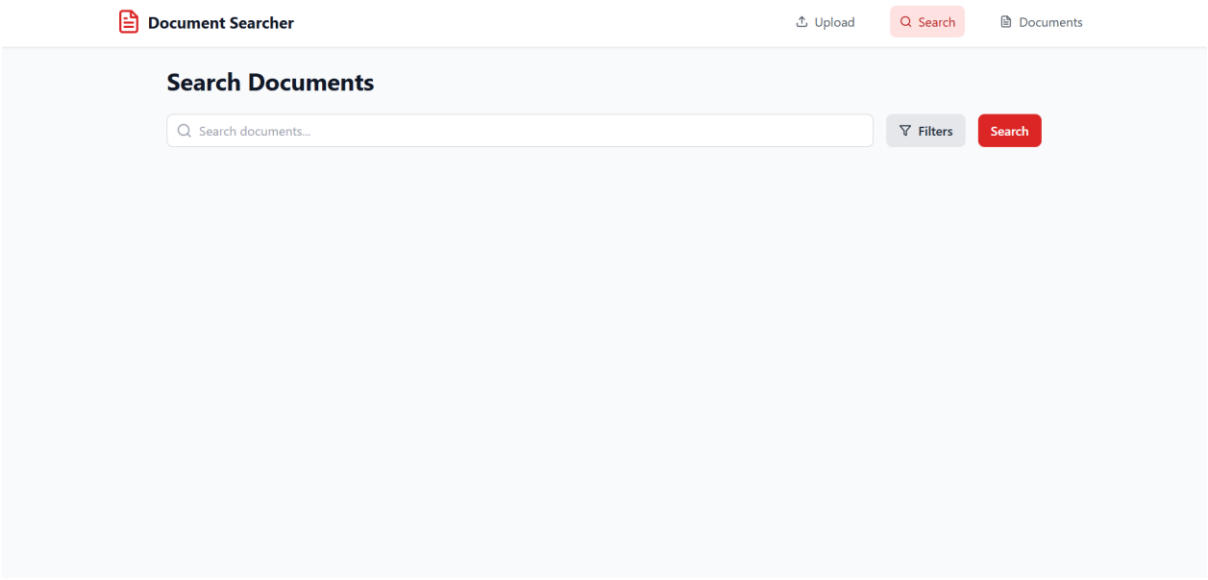
The chosen schema provides:

- **Scalability:** The use of MongoDB allows dynamic document structures suitable for variable OCR outputs.
- **Flexibility:** Fields can be easily extended for additional metadata or integrations (e.g., cloud storage, AI summarization).
- **Efficiency:** Using references between documents ensures minimal redundancy and optimized retrieval.
- **Integrity:** Relationships among documents, jobs, and pages maintain logical consistency throughout processing.

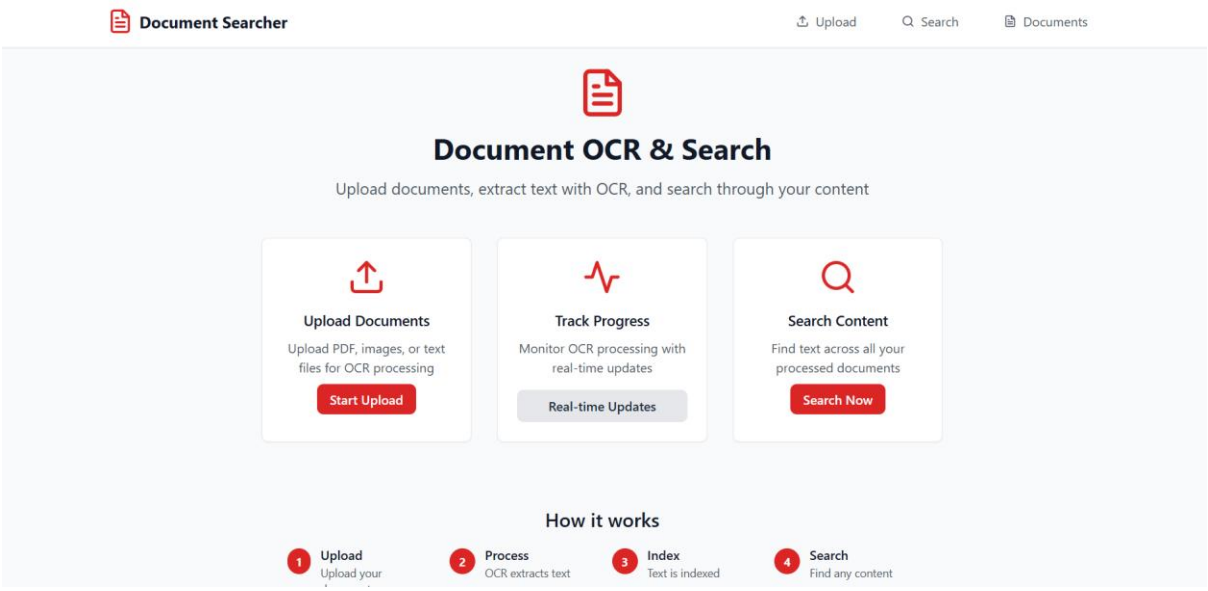
Front-End Screens

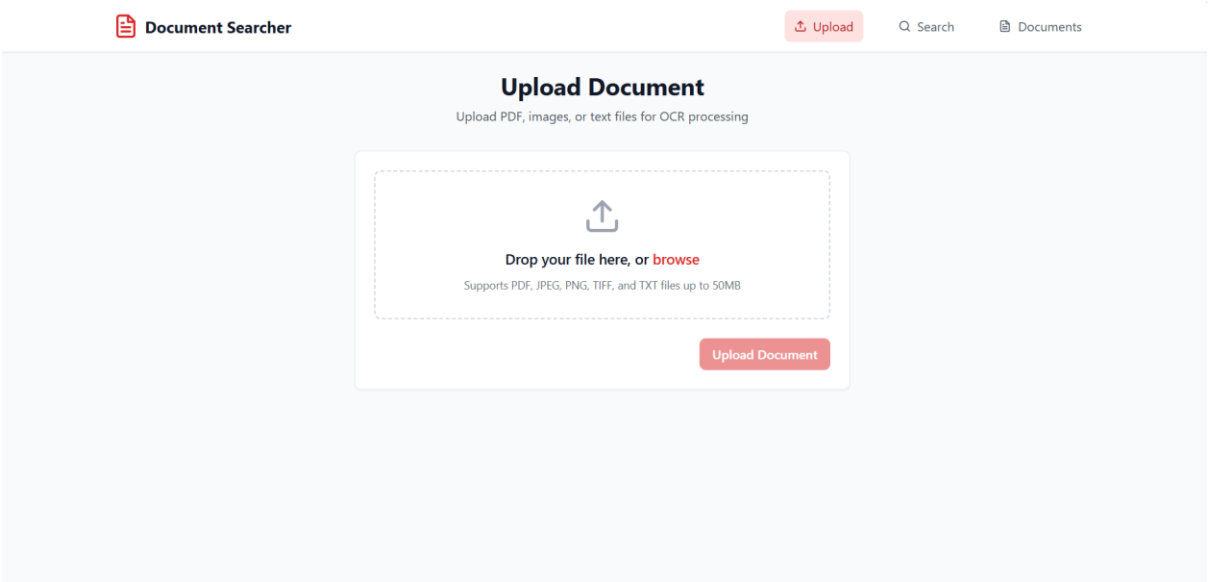


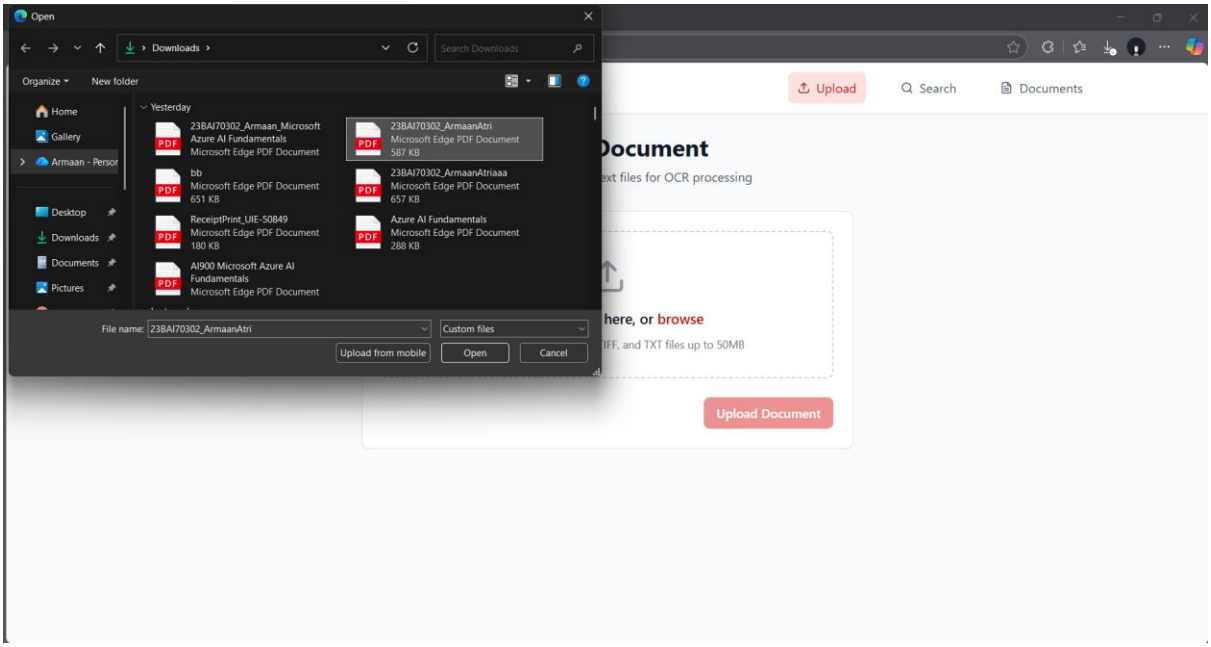


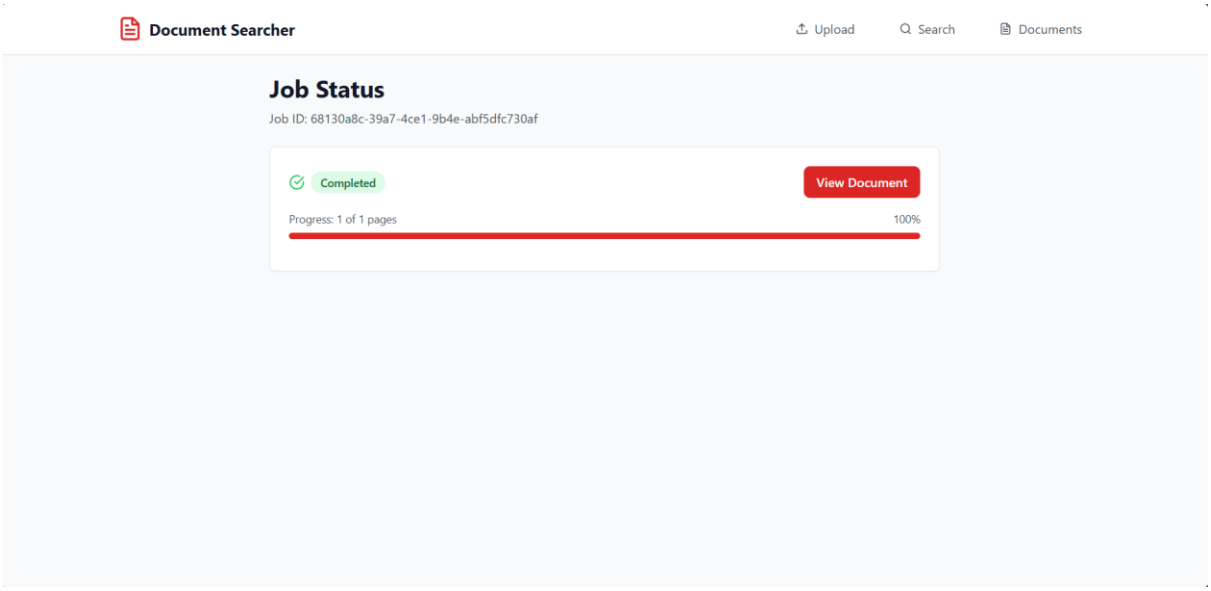


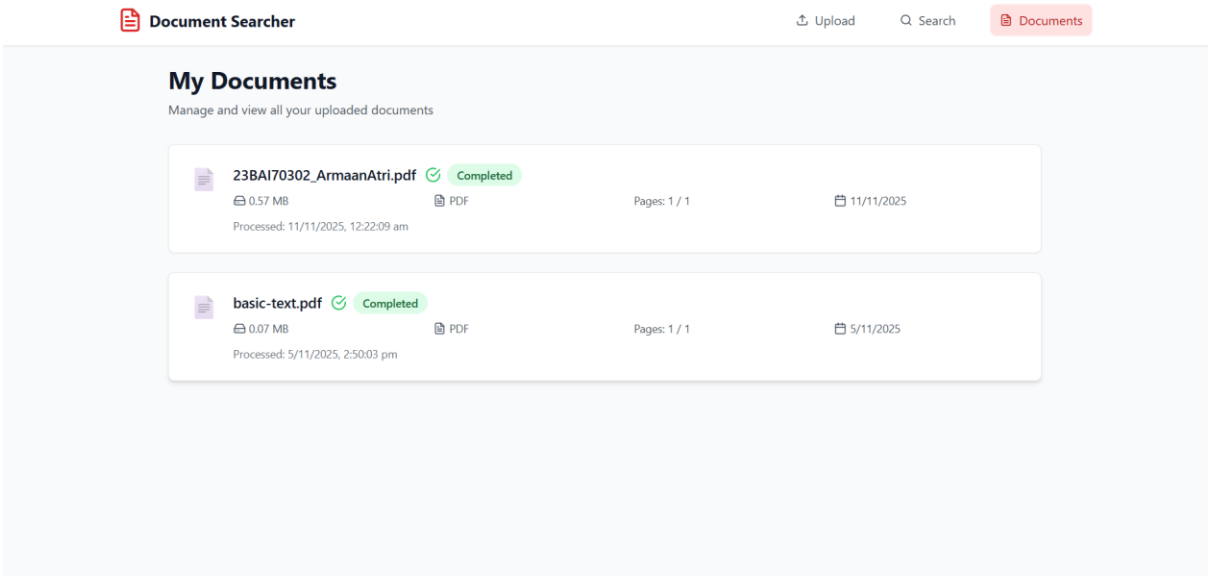
Output Screens and Reports








Document Searcher
Upload
Search
Documents
Job Status
Job ID: 68130a8c-39a7-4ce1-9b4e-abf5dfc730af
Completed
View Document
Progress: 1 of 1 pages
100%


Document Searcher
Upload
Search
Documents
My Documents
Manage and view all your uploaded documents
23BAI70302_ArmaanAtri.pdf
Completed
0.57 MB
PDF
Pages: 1 / 1
11/11/2025
Processed: 11/11/2025, 12:22:09 am
basic-text.pdf
Completed
0.07 MB
PDF
Pages: 1 / 1
5/11/2025
Processed: 5/11/2025, 2:50:03 pm

Sample Document for PDF Testing

Introduction

This is a simple document created to test basic PDF functionality. It includes various text formatting options to ensure proper rendering in PDF readers.

Text Formatting Examples

1. **Bold text** is used for emphasis.
2. *Italic text* can be used for titles or subtle emphasis.
3. ~~Strikethrough~~ is used to show deleted text.

Lists

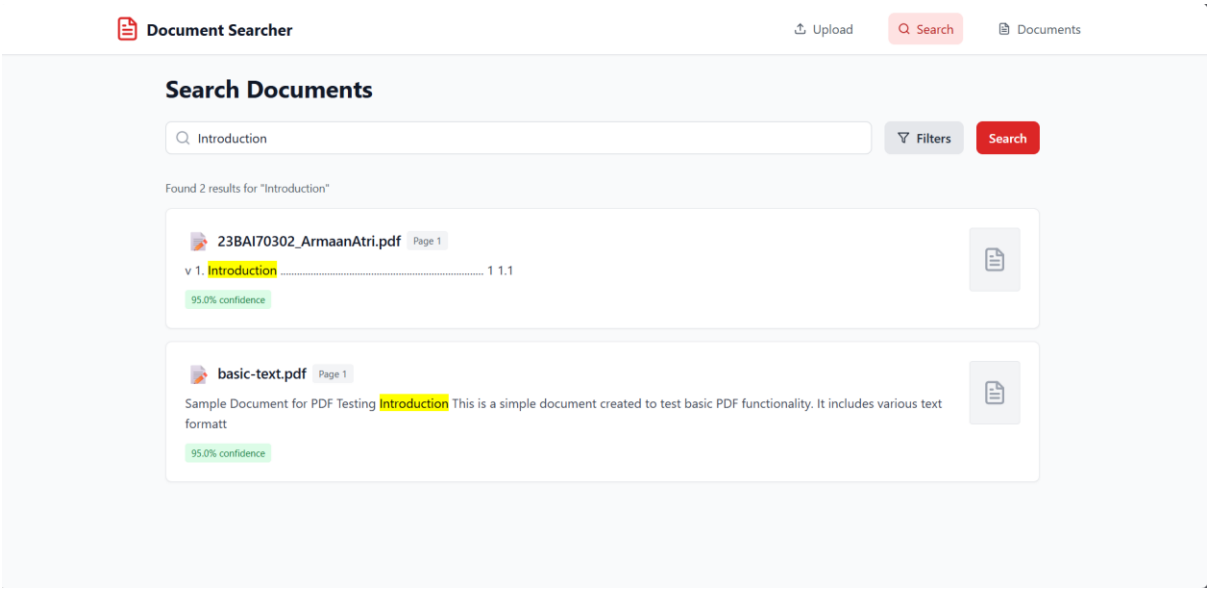
Here's an example of an unordered list:

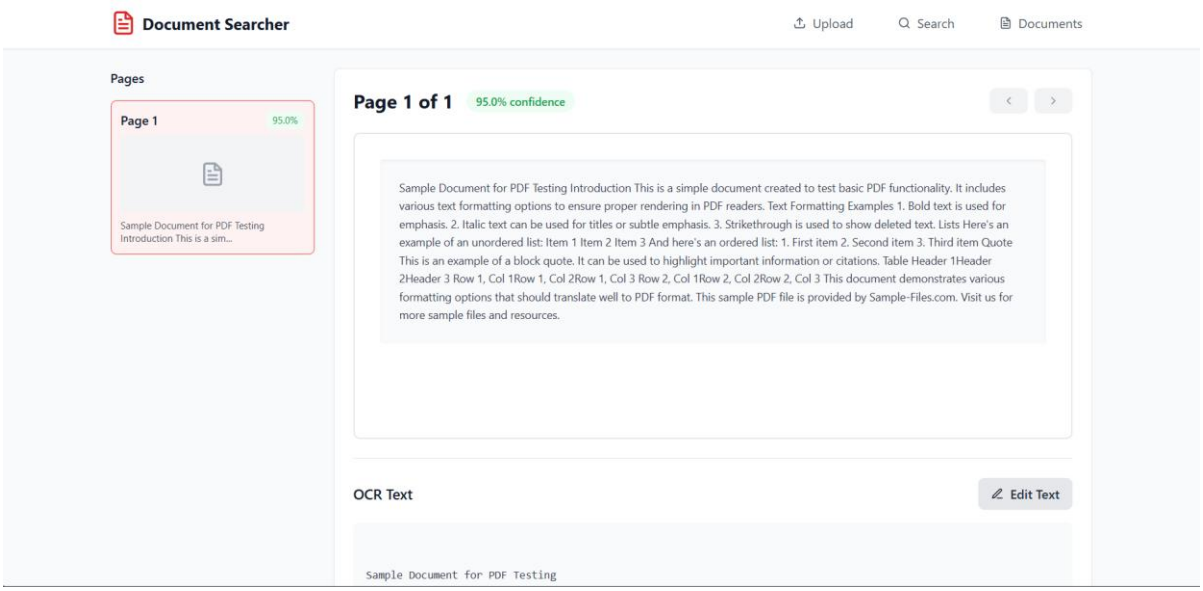
- Item 1
- Item 2
- Item 3

And here's an ordered list:

1. First item
2. Second item
3. Third item

(Sample Content from the PDF that is Uploaded)





Limitation & Future Scope

Limitations

1. **Mock OCR / Accuracy**
    - Current implementation uses a mock OCR or a simple OCR library; accuracy is limited for complex layouts, handwritten text, or low-quality scans.
    - OCR confidence and error-handling are basic and may produce noisy or incomplete text.
2. **In-memory Queue**
    - The queue used for processing jobs is in-memory (single-process). It lacks persistence, fault tolerance, and horizontal scalability.
    - If the server crashes, queued jobs are lost.
3. **Local File Storage**
    - Files are stored locally in an uploads/ folder. This is not resilient, not suitable for multi-instance deployments, and risks disk saturation.
    - No automated backup or lifecycle management (e.g., archival, TTL).
4. **Search Scalability & Relevance**
    - Basic text search (MongoDB text indexes or simple queries) may not scale well for large corpora, and relevance ranking is limited.
    - No advanced features like fuzzy search, highlighting, ranking by relevance, or phrase search tuning.
5. **Authentication & Authorization**
    - Minimal or no user authentication, role management, or per-user document access control is included.
    - Lack of audit trails for document actions.

6. **Performance on Large Documents**
    - Memory and CPU spikes when processing large multi-page PDFs or many concurrent uploads.
    - No worker isolation or GPU acceleration for OCR where applicable.
7. **Limited File Type Support & Preprocessing**
    - May not support complex document formats (scanned PDFs with rotated pages, multi-column layouts, or embedded images).
    - Insufficient preprocessing (deskewing, denoising, resolution adjustments).
8. **Monitoring, Logging & Error Recovery**
    - Basic logging only; lacks structured observability (metrics, centralized logs, dashboards) and automated retries for transient failures.
9. **Privacy & Compliance**
    - No built-in mechanisms for data encryption at rest beyond OS-level protections, compliance logging, or PII redaction workflows.
    - Not designed with GDPR/other regulatory workflows in mind (data deletion requests, consent tracking).
10. **Testing & CI/CD**
    - Limited automated tests for OCR quality, integration, end-to-end flows, or performance regression.
    - No production-ready CI/CD pipelines or container orchestration.

Future Scope

1. **Replace In-Memory Queue with Redis / Bull / RabbitMQ**
    - Add persistent, distributed job queue for reliability, retries, delayed jobs, and horizontal scaling.
    - Implementation note: use bullmq or bee-queue with Redis; separate worker processes handle OCR.
2. **Integrate Robust OCR (Tesseract / Cloud OCR)**
    - Integrate Tesseract.js for client-side/lightweight or server-side Tesseract, or cloud OCR APIs (Google Cloud Vision, AWS Textract, Azure OCR) for higher accuracy.
    - Add preprocessing (deskew, binarization, DPI normalization) to improve OCR results.
3. **Move File Storage to Cloud (S3/MinIO)**
    - Use S3-compatible storage with lifecycle rules, versioning, and scalable capacity.
    - Store file references in MongoDB; stream files to/from workers.
4. **Full-Text Search Engine (ElasticSearch / OpenSearch)**
    - Replace or augment Mongo text search with ElasticSearch/OpenSearch for scalable, fast, and feature-rich search (highlighting, fuzzy search, relevancy tuning, aggregations).
    - Implementation note: index page-level documents and document metadata.

5. **Authentication, Authorization & RBAC**
    - Implement user accounts, JWT-based auth, role-based access control (admin, uploader, viewer), and per-document permissions.
    - Add audit logs for critical actions (upload, delete, reprocess, share).
6. **Advanced NLP & Document Understanding**
    - Add named-entity recognition (NER), keyword extraction, document classification, and summarization.
    - Use open-source models or cloud NLP APIs to extract structured information (dates, invoices, names).
7. **Document Versioning & Change History**
    - Support reprocessing with version history, diffs of extracted text, and ability to revert to previous OCR outputs.
8. **Batch Processing & Bulk Uploads**
    - Add robust batch import, scheduling, and bulk reprocessing features with progress dashboards.
9. **Document Annotation & Collaboration**
    - Allow users to annotate OCR text, correct OCR errors, highlight text, and share annotations among users.
10. **Multi-language & Handwriting Support**
    - Support multiple languages and handwriting recognition models for broader use-cases.

GitHub URL

Armaan1804/Capstone-Project

The Project Structure as Follows:

```
mern-document-search/
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   │   ├── Upload.jsx
│   │   │   ├── JobStatus.jsx
│   │   │   ├── Search.jsx
│   │   │   ├── DocumentViewer.jsx
│   │   │   └── Documents.jsx
│   │   ├── hooks/
│   │   │   └── useWebSocket.js
│   │   ├── utils/
│   │   │   └── api.js
│   │   ├── App.jsx
│   │   └── main.jsx
│   ├── package.json
│   └── vite.config.js
└── backend/
    ├── src/
    │   ├── models/
    │   │   ├── Document.js
    │   │   ├── Job.js
    │   │   └── Page.js
    │   ├── routes/
    │   │   ├── upload.js
    │   │   ├── jobs.js
    │   │   ├── documents.js
    │   │   ├── search.js
    │   │   └── pages.js
    │   ├── middleware/
    │   │   └── upload.js
    │   ├── utils/
    │   │   ├── database.js
    │   │   ├── fileHash.js
    │   │   ├── queue.js
    │   │   └── simpleQueue.js
    │   └── server.js
    └── package.json
```

References

1. React.js – *React Official Documentation*. (n.d.). Meta Platforms, Inc.
   Retrieved from: https://react.dev/

2. Express.js – *Fast, unopinionated, minimalist web framework for Node.js*. (n.d.).
   Retrieved from: https://expressjs.com/

3. MongoDB – *The Developer Data Platform*. (n.d.). MongoDB, Inc.
   Retrieved from: https://www.mongodb.com/docs/

4. Node.js – *Node.js Documentation*. (n.d.). OpenJS Foundation.
   Retrieved from: https://nodejs.org/en/docs

5. Tesseract.js – *Pure JavaScript OCR for 100+ languages*. (n.d.).
   Retrieved from: https://tesseract.projectnaptha.com/

6. Socket.IO – *Bidirectional and Low-latency Communication for Every Platform*. (n.d.).
   Retrieved from: https://socket.io/docs/

7. Tailwind CSS – *A Utility-First CSS Framework*. (n.d.).
   Retrieved from: https://tailwindcss.com/docs

8. Mongoose – *Elegant MongoDB Object Modeling for Node.js*. (n.d.).
   Retrieved from: https://mongoosejs.com/docs/

9. Multer – *Node.js Middleware for Handling Multipart/Form-Data*. (n.d.).
   Retrieved from: https://github.com/expressjs/multer

10. Axios – *Promise-based HTTP Client for the Browser and Node.js*. (n.d.).
    Retrieved from: https://axios-http.com/

11. Helmet.js – *Secure Express Apps by Setting HTTP Headers*. (n.d.).
    Retrieved from: https://helmetjs.github.io/

12. Vite – *Next Generation Frontend Tooling*. (n.d.).
    Retrieved from: https://vitejs.dev/

13. Cloud OCR APIs – *Google Cloud Vision API Documentation*. (n.d.).
    Retrieved from: https://cloud.google.com/vision/docs/ocr

14. GitHub – *Version Control Platform for Source Code Management*. (n.d.).
    Retrieved from: https://github.com/

15. Visual Studio Code – *Code Editing. Redefined.* (n.d.). Microsoft.
    Retrieved from: https://code.visualstudio.com/

16. OpenAI. (2024). *Understanding OCR Systems and AI Text Extraction*. Technical overview.
    Retrieved from: https://openai.com/research

17. MDN Web Docs – *JavaScript Reference and Web APIs*. Mozilla Developer Network.
    Retrieved from: https://developer.mozilla.org/

18. NPM Registry – *Node Package Manager (NPM) Library Repository*. (n.d.).
    Retrieved from: https://www.npmjs.com/

19. TutorialsPoint – *MERN Stack Development Tutorial*. (2023).
    Retrieved from: https://www.tutorialspoint.com/mern_stack/index.htm

Bibliography

1. Sharma, R. (2023). *Building a Document OCR System using Node.js and Tesseract.js*. Medium. Retrieved from: https://medium.com/

2. Kumar, A. (2023). *Implementing Real-Time Communication using Socket.IO in MERN Applications*. GeeksforGeeks. Retrieved from: https://www.geeksforgeeks.org/

3. Singh, P. (2024). *How to Integrate MongoDB with Express.js for Data Handling*. Dev.to. Retrieved from: https://dev.to/

4. ByteXL Learning Portal. (2024). *Capstone Project Ideas – Full Stack Development Unit 3*. Retrieved from: https://bytexl.com/

5. FreeCodeCamp. (2023). *Learn MERN Stack Development – Complete Tutorial*. YouTube. Retrieved from: https://www.youtube.com/@freecodecamp

6. Traversy, B. (2023). *MERN Stack Crash Course – Build a Full Stack App*. YouTube. Traversy Media Channel. Retrieved from: https://www.youtube.com/@TraversyMedia

7. MongoDB University. (2023). *Data Modeling in NoSQL Databases (M001)*. MongoDB University Course. Retrieved from: https://university.mongodb.com/

8. OpenAI Research Blog. (2024). *AI-Assisted Text Extraction and Document Understanding*. Retrieved from: https://openai.com/research

9. The Net Ninja. (2023). *React & Node.js – Full Stack MERN Tutorials*. YouTube. Retrieved from: https://www.youtube.com/@NetNinja

10. DigitalOcean Tutorials. (2023). *Deploying MERN Applications to Cloud Servers*. Retrieved from: https://www.digitalocean.com/community/tutorials

11. Microsoft Learn. (2024). *Modern Web Application Architecture and API Design*. Retrieved from: https://learn.microsoft.com/

12. GeeksforGeeks. (2023). *Understanding OCR – Optical Character Recognition with Tesseract*. Retrieved from: https://www.geeksforgeeks.org/optical-character-recognition-ocr-with-tesseract/

13. TutorialsPoint. (2023). *WebSocket Communication and Real-Time Updates in JavaScript*. Retrieved from: https://www.tutorialspoint.com/websocket_programming/

14. Stack Overflow. (2024). *Common Solutions for File Uploading and Node.js Multer Configuration*. Retrieved from: https://stackoverflow.com/

15. Mozilla Developer Network (MDN). (2024). *Web APIs and Asynchronous JavaScript Programming*. Retrieved from: https://developer.mozilla.org/