# Frontend Implementation

For a comprehensive file chatbot/exam assistant, here's what you'll need to implement on the frontend:

1. **User Interface**
   * Clean, intuitive chat interface
   * File upload component with drag-and-drop functionality
   * Document viewer for displaying uploaded files
   * Progress indicators for uploads and processing
   * Message history display with proper formatting
2. **File Handling**
   * Support for multiple file formats (PDF, images, docx, etc.)
   * File preview capabilities
   * Document navigation for multi-page files
   * Image viewing with zoom/pan controls
3. **Interaction Design**
   * Message input area with formatting options
   * Send button and keyboard shortcuts
   * Chat history scrolling with proper loading states
   * Visual indicators for AI thinking/processing
4. **Responsive Design**
   * Mobile-friendly layout
   * Adaptive components for different screen sizes
   * Touch-friendly controls for mobile users
5. **Accessibility Features**
   * Screen reader compatibility
   * Keyboard navigation
   * Proper contrast ratios
   * Alternative text for images
6. **User Experience**
   * Onboarding flow for new users
   * Clear error messages and recovery paths
   * Session persistence (saving state between visits)
   * Loading states and animations

# Backend Implementation

# Tools for Image-Text Mapping in Document Chatbots

## Text Extraction Tools

* **PyMuPDF (fitz)**: Extract text with positional data from PDFs
* **Unstructured.io**: Extract text from multiple document formats
* **pdf2image + Tesseract**: Convert and extract text from scanned documents
* **python-docx**: Extract text from Word documents with layout information

## Image Extraction & Processing

* **PyMuPDF**: Extract images with positional data from PDFs
* **OpenCV**: Process and analyze images
* **Pillow (PIL)**: Handle image data and formats
* **python-docx**: Extract embedded images from Word documents

## Text Analysis & Keyword Extraction

* **spaCy**: NLP library for keyword extraction and entity recognition
* **KeyBERT**: BERT-based keyword extraction
* **YAKE/RAKE**: Unsupervised keyword extraction algorithms
* **LexRank**: Extract key phrases from text

## Image-Text Association

* **CLIP**: Connect images with textual descriptions
* **LayoutParser**: Analyze document layout to connect text and images
* **Donut**: Document understanding transformer
* **LayoutLM**: Pre-trained model for document layout analysis

## Storage Solutions

* **Redis**: Fast key-value store for image-keyword mappings
* **PostgreSQL + pgvector**: Store text, images, and vector embeddings
* **MongoDB**: Document store with GridFS for image storage
* **SQLite**: Lightweight relational database for local implementation

## Query Processing

* **sentence-transformers**: Create embeddings for semantic matching
* **FAISS/Chroma**: Fast similarity search for finding matching keywords
* **NetworkX**: Graph-based reasoning for connecting query terms to document content

These tools can be combined to build your system for mapping keywords to images and retrieving relevant images based on user queries.

**Goal:**

* After the user uploads a file (PDF, DOCX, etc.):
  + Extract **text** and **images**.
  + Preprocess **images** (e.g., resize, encode, or normalize).
  + Create a **key-value store**:
    - **Key** = word/concept from extracted text related to an image.
    - **Value** = image ID (could be a filename, UUID, or database reference).

Later, when the user **asks a question**, your chatbot:

* **Analyzes the question** to find the best matching **key(s)**.
* **Returns the related image** using the stored image ID.

Here’s a simple architecture flow for this:

1. **Upload file**
2. **Extract text and images**  
   (e.g., PyMuPDF (fitz), pdf2image, pdfminer, etc.)
3. **Preprocess images**  
   (e.g., resize, hash, save with unique ID)
4. **Create Key-Value mapping**
   * Use basic NLP (spaCy, NLTK) to extract important words from the text.
   * Associate important keywords/concepts with image IDs.
5. **Store Key-Value pairs**  
   (Maybe in a simple dictionary, JSON file, SQLite, or MongoDB.)
6. **User asks a question**
7. **Analyze question** to find matching **key(s)**
   * Use text similarity (e.g., cosine similarity, embeddings)
8. **Return image(s)** based on matching image IDs.

**Example Structure:**

json

CopyEdit

{

"engine": "pdf\_parser",

"data": [

{

"key": "photosynthesis",

"value": "img\_12345.jpg"

},

{

"key": "chlorophyll",

"value": "img\_67890.jpg"

}

]

}

**Simple Code Sketch:**

python

CopyEdit

import fitz # PyMuPDF

import uuid

import os

# Step 1: Extract images

def extract\_images\_from\_pdf(file\_path, output\_folder):

doc = fitz.open(file\_path)

images = []

for page\_index in range(len(doc)):

for img\_index, img in enumerate(doc[page\_index].get\_images(full=True)):

xref = img[0]

base\_image = doc.extract\_image(xref)

img\_bytes = base\_image["image"]

img\_ext = base\_image["ext"]

img\_id = str(uuid.uuid4())

img\_path = os.path.join(output\_folder, f"{img\_id}.{img\_ext}")

with open(img\_path, "wb") as img\_file:

img\_file.write(img\_bytes)

images.append((img\_id, img\_path))

return images

# Step 2: Build key-value mapping manually or via NLP

key\_value\_store = {

"photosynthesis": "img\_12345.jpg",

"chlorophyll": "img\_67890.jpg"

}

# Step 3: When user asks a question

def find\_image\_for\_question(question, key\_value\_store):

for key in key\_value\_store:

if key in question.lower():

return key\_value\_store[key]

return None

# Example

user\_question = "Can you show me the diagram of photosynthesis?"

result = find\_image\_for\_question(user\_question, key\_value\_store)

print(result) # => "img\_12345.jpg"

For your backend needs:

1. **API Integration**
   * Integration with Moonshot AI's Kimi VL A3B Thinking API
   * Request/response handling
   * Error handling and retries
   * Authentication and API key management
2. **File Processing Pipeline**
   * File upload handling and validation
   * Document parsing and text extraction
   * Image extraction and preprocessing
   * Chunking for large documents to fit context windows
3. **Server Architecture**
   * RESTful API design or GraphQL
   * Stateless design for scalability
   * Middleware for authentication, logging, etc.
   * Rate limiting and request queuing
4. **Data Storage**
   * Database for user accounts and sessions
   * File storage system (local or cloud-based)
   * Conversation history storage
   * Caching layer for performance
5. **Security Implementation**
   * User authentication and authorization
   * Data encryption (in transit and at rest)
   * Input validation and sanitization
   * Protection against common vulnerabilities
6. **Performance Optimization**
   * Response caching
   * Asynchronous processing for heavy tasks
   * Database query optimization
   * Content delivery network integration
7. **Deployment Infrastructure**
   * Containerization (Docker)
   * CI/CD pipeline
   * Hosting environment setup
   * Monitoring and alerting

**🧩 Tech Stack**

* **Frontend**: React (Vite, TypeScript)
* **Backend**: Node.js ,mongodb.
* **AI Integration**: llava
* **Supported File Types**: All files

exam-assistant/

├── frontend/ # Frontend (React)

│ ├── src/

│ │ ├── App.tsx

│ │ ├── main.tsx

│ │ ├── FileUploader.tsx

│ │ ├── ChatBox.tsx

│ │ └── api.ts # Axios instance

│ └── index.html

├── backend/ # Backend (Node.js)

│ ├── app.py

│ ├── # Handles file uploads

│ └── # Handles AI API calls

├── uploads/ # Folder for uploaded files

├── .env

└── README.md

**🎨 Frontend (React)**

**Key Features**:

* Chat Interface for user queries
* Drag-and-drop file upload (text, image, PDF)
* File viewer (image display, PDF rendering)
* Fully responsive, mobile-friendly design

**Libraries & Tools**:

* React, Vite, TypeScript, Tailwind CSS
* react-dropzone, react-pdf, axios, react-markdown, framer-motion

**🔧 Backend (Node.js 1+ MongoDB)**

**Key Features**:

* REST API for uploads, chat, and file retrieval
* Extract text from PDFs, resize images
* Connects to external AI API for responses

**Libraries**:

* Express.js, SQLite, Knex.js
* Multer (upload), pdf-parse, sharp, axios, dotenv, cors

**🗃 SQLite Database Schema**

CREATE TABLE users (

id INTEGER PRIMARY KEY,

username TEXT,

email TEXT

);

CREATE TABLE chats (

id INTEGER PRIMARY KEY,

user\_id INTEGER,

message TEXT,

timestamp DATETIME

);

CREATE TABLE files (

id INTEGER PRIMARY KEY,

user\_id INTEGER,

file\_path TEXT,

file\_type TEXT,

upload\_date DATETIME

);

**📁 File Handling**

* **Text**: Stored and retrieved as plain text
* **Images**: Processed with sharp, displayed in UI
* **PDFs**: Parsed with pdf-parse, rendered with react-pdf

**⚙️ Development Workflow**

1. **Setup**:
   * npm create vite@latest for frontend
   * npm init -y for backend
2. **Frontend**:
   * Build chat interface and file upload
   * Connect using Axios
3. **Backend**:
   * Set up API routes and SQLite
   * Integrate AI API
4. **Testing**:
   * Jest (frontend)
   * Supertest (backend)
5. **Deployment**:
   * Frontend: **Vercel** or **Netlify**
   * Backend: **Render** or **Heroku**
   * Optional: Use **Docker** for containerization

**🔌 Example API Endpoints**

* POST /api/upload: Upload files
* GET /api/files/:userId: Get user files
* POST /api/chat: Send chat message, get AI reply
* GET /api/chats/:userId: View chat history

**📝 Notes**

* **AI Integration**: llava
* **Scaling**: Start with mongoDB
* **Security**: Add JWT for authentication + input validation before production

**online file chatbot (exam assistant)** using **RAG architecture**:

**🌐 Final Architecture (Using kimi-vl-a3b-thinking)**

[Frontend]

↓

[FastAPI Backend]

↓

[Parse Files (text+images)]

↓

[Vector Search on embedded chunks]

↓

[Top-K chunks + image]

↓

[LLM Call → kimi-vl-a3b-thinking API]

↓

[Answer → Display to user]