Technical Documentation: ChatBot

1. Introduction

This document provides a formal technical overview of the Local PDF Q&A Chatbot architecture. The system is a client-side Retrieval-Augmented Generation (RAG) application that utilizes a Python proxy server to interface with a local Ollama instance. It analyzes specific issues and bugs identified during code analysis.

---

2. System Overview

The application allows users to upload PDF or DOCX documents, extract text client-side, and perform semantic searches against that text to generate answers using a local Large Language Model (LLM). It consists of a frontend interface for document processing and chat, and a lightweight backend proxy to handle CORS and API forwarding to Ollama.

---

3. Technical Architecture

3.1 Technology Stack

- **Frontend Runtime:** Browser (HTML5/ES6)
- **Styling:** Bootstrap 5.3.2 & Custom CSS
- **PDF Engine:** PDF.js (v3.11.174)
- **DOCX Engine:** Mammoth.js
- **Backend Proxy:** Python `http.server` / `socketserver`
- **AI Backend:** Ollama (Local LLM API)

3.2 File Structure Summary

- **`index.html`**: Main user interface containing the chat window, file upload controls, and settings configuration.
- **`script.js`**: Core logic for text extraction, chunking, vector embedding generation, cosine similarity search, and chat state management.
- **`server.py`**: A Python HTTP proxy that forwards requests from the browser to the local Ollama instance (default `127.0.0.1:11434`) to bypass browser CORS restrictions.
- **`test_parsing.js`**: A unit test script verifying the JSON stream parsing logic used for handling LLM responses.
- **`pdf.min.js` / `pdf.worker.min.js`**: Core libraries responsible for parsing and rendering PDF binary data.

---

4. Backend API Flow

4.1 The Python Proxy The browser cannot directly access the Ollama API due to Cross-Origin Resource Sharing (CORS) restrictions. The `server.py` script acts as a middleware proxy.

Process:

1. **Interception:** The server listens on port 3001. Requests starting with `/api/` are intercepted.
2. **Forwarding:** The server constructs a new request to `OLLAMA_URL` (defaulting to `http://127.0.0.1:11434`).
3. **Header Spoofing:** It injects `Origin` and `Referer` headers to match the Ollama host, bypassing strict CORS checks on the LLM side.
4. **Streaming:** The proxy supports streaming responses, reading chunks of 1024 bytes and writing them immediately to the client `wfile`.

---

5. Technical Workflow (RAG Pipeline)

5.1 Document Ingestion

- **Detection:** The system detects file types (`application/pdf` or `vnd.openxmlformats...`).
- **Extraction:**
    - **PDF:** Iterates through all pages using `pdfjsLib`, concatenating text items.
    - **DOCX:** Uses `mammoth.extractRawText` to pull raw string data.
- **Chunking:** The text is split into chunks of 200 words via whitespace splitting. No sliding window is currently implemented.

5.2 Vector Embedding Upon extraction, the client immediately iterates through all chunks and calls the `/api/embeddings` endpoint.

- **Model:** Hardcoded to use `nomic-embed-text`.
- **Storage:** Embeddings are stored in the client-side memory array `chunkEmbeddings`.

5.3 Retrieval & Generation When a user asks a question:

1. **Query Embedding:** The user's query is embedded using the same model.
2. **Vector Search:** A Cosine Similarity calculation compares the query vector against all chunk vectors.
3. **Context Selection:** The top 5 scoring chunks are selected.

4. **Prompt Engineering:** A "Strict Mode" prompt is constructed, injecting the retrieved chunks and instructing the model (hardcoded as `llama2`) to answer *only* based on that context.

---

6. Identified Bugs & Issues

Bug 1: Hardcoded AI Models

- **Issue:** The embedding model is hardcoded to `nomic-embed-text` and the chat model is hardcoded to `llama2`.
- **Impact:** If the user has not explicitly pulled these specific models in Ollama, the application will fail silently or return API errors. The user cannot select different models via the UI.

Bug 2: Naive Chunking Strategy

- **Issue:** Text chunking relies on simple whitespace splitting (`text.split(/\s+/)`) with a fixed size of 200.
- **Impact:** This cuts sentences in half and ignores semantic boundaries, potentially degrading RAG performance by breaking context.

Bug 3: Hardcoded Error Messages

- **Issue:** The error message for direct file access instructs the user to open `http://localhost:3001`.
- **Impact:** If the user changes the `PORT` variable in `server.py`, this error message will provide incorrect instructions.

Bug 4: Missing Vectorization Error Handling

- **Issue:** In the embedding loop (`for` loop iterating `chunks`), if a single `embedText` call fails (returns null), it is logged to the console, but the process continues.
- **Impact:** Partial failures result in an incomplete knowledge base without warning the user that specific sections of the document are missing.

---

7. Recommendations

- **Model Selection:** Update `script.js` to fetch available models from `/api/tags` and allow the user to select their preferred embedding and chat models via the settings menu.
- **Improved Chunking:** Implement a sentence-aware chunker or a sliding window approach (e.g., chunk size 500 characters with 50 character overlap) to improve context retrieval.
- **Dynamic Configuration:** Pass the server port to the frontend or use relative paths in error messages to ensure the URL in the instructions matches the actual server configuration.
- **Robust Error Handling:** Implement a retry mechanism for failed embedding requests and provide a UI alert if the document was only partially processed.

---

8. Conclusion

The Local PDF Q&A Chatbot demonstrates a functional baseline for client-side RAG architecture. It successfully offloads processing to the client and uses a proxy to bridge browser-local capabilities with a local AI server. However, the hardcoded dependency on specific Ollama models and naive text processing techniques limit its robustness and flexibility in a production environment.
