Contents

1 Introduction **1**

1.1 Project Motivation 1

1.2 Problem Statement 1

1.3 Project Goals 1

2 Background: Concepts and Tools **3**

2.1 What is RAG (Retrieval-Augmented Generation)? 3

2.1.1 Why RAG for legal/tax tasks? 3

2.2 Key Components Used 3

2.2.1 LangChain 3

2.2.2 FAISS 4

2.2.3 Ollama (local LLM hosting) 4

2.2.4 Streamlit 4

2.2.5 SQLite + bcrypt 4

3 System Design and Architecture **5**

3.1 High-level Architecture 5

3.2 Data Flow — Step by Step 5

3.3 Why chunking, overlap, and top-k matter? 6

3.3.1 Chunk Size 6

3.3.2 Chunk Overlap 6

3.3.3 Top-k Passages 6

4 Implementation **7**

4.1 Project layout (recommended) 7

4.1.1 Key Python modules 7

4.2 Important code excerpts (explanations) 8

4.2.1 Saving uploads and hashing (stability for caching) 8

4.2.2 Chunking using RecursiveCharacterTextSplitter 8

4.2.3 Building/Loading FAISS index 8
4.2.4 Creating the QA chain 9
4.3 Security: credentials and logs 9

5 Use-Cases: Legal and Tax Advisory Examples **10**
5.1 Example 1: Tax query for a layman 10
5.2 Example 2: Company policy question 10

6 Testing, Performance, and Limitations **11**
6.1 Functional Testing 11
6.2 Performance Considerations 11
6.3 Known Limitations 11

7 Deployment: Running Locally on a Wi-Fi Network **13**
7.1 Prerequisites (Host machine) 13
7.2 Host setup commands (example) 13
7.3 Starting services 14
7.4 Security notes for LAN deployment 14

8 Admin Features and User Management **15**
8.1 Overview 15
8.2 Login Interface 15
8.3 Admin Interface 16
8.4 Client Interface 16
8.5 Admin Responsibilities 16
8.6 How Persistence Works 17

9 Extending the System: Adding Users and Changing Passwords **18**
9.1 Via Admin UI 18
9.2 Programmatically (script) 18

10 Source Code and Repository **19**
10.1 Repository structure 19
10.2 Placeholder GitHub link 19

11 Ethics, Privacy and Legal Considerations **20**
11.1 Why local-first matters 20
11.2 Limitations and responsibilities 20

12 Conclusion and Future Work **21**
12.1 Conclusion 21
12.2 Future work 21

List of Figures

3.1 High-level system architecture. 5

8.1 Login Interface — Secure login for Admin and Client roles. 15
8.2 Admin Interface — Upload, Index, and Manage Legal Knowledge. 16
8.3 Client Interface — Ask questions and get AI-generated legal responses. . . 16

Abstract

This internship report describes the design, implementation, and deployment of a local, privacy-focused Retrieval-Augmented Generation (RAG) based AI Legal Assistant. The system enables users to ask natural-language legal and tax questions about documents (e.g., laws, policies, manuals) and receive answers grounded in the uploaded sources. It uses LangChain for orchestration, FAISS for efficient vector search, Ollama-hosted local models for embeddings and text generation, and a Streamlit-based web UI with separate admin/client interfaces. The report explains the full data flow (upload $\rightarrow$ chunking $\rightarrow$ embedding $\rightarrow$ indexing $\rightarrow$ retrieval $\rightarrow$ generation), security and persistence choices (SQLite for credentials and logs, hashed passwords), and local Wi-Fi deployment steps for use within a secure local network.

Chapter 1

Introduction

1.1 Project Motivation

Accessing legal knowledge is difficult for many non-experts. Laws, tax codes, and corporate policies are long and written in legalese — this creates barriers for citizens, small business owners, and even junior lawyers. A searchable, conversational assistant that answers questions using the exact language of the law (and cites the source) would accelerate understanding and reduce errors.

1.2 Problem Statement

Design a local, secure system that:

- Allows an administrator to upload legal documents (PDFs).
- Converts documents into a knowledge base that can be queried in natural language.
- Answers questions by retrieving relevant passages and generating concise, grounded responses.
- Runs fully offline (no cloud API calls), keeping data private.
- Supports multiple clients on a local network, and logs queries for auditing.

1.3 Project Goals

1. Implement a RAG pipeline: chunking, embeddings, FAISS index, retriever, and LLM generation.
2. Provide an admin interface for uploads and client interface for queries using Streamlit.

3. Store user credentials and logs in SQLite; use bcrypt to hash passwords.

4. Provide documentation and a deployable local workflow over Wi-Fi.

Chapter 2

Background: Concepts and Tools

This chapter explains core concepts in plain language.

2.1 What is RAG (Retrieval-Augmented Generation)?

Retrieval-Augmented Generation (RAG) is a pattern that improves LLM answers by retrieving relevant texts from a knowledge base and giving them to the model as context. Instead of relying solely on the LLM's memory, RAG grounds answers in source documents, enabling factual consistency and citations.

2.1.1 Why RAG for legal/tax tasks?

Legal answers must reference authoritative documents. RAG helps by:

- Finding the exact law clause or paragraph.
- Letting the LLM summarize/explain the retrieved paragraph in clearer language.
- Reducing hallucinations because the model uses concrete source text.

2.2 Key Components Used

2.2.1 LangChain

LangChain is a Python toolkit for building LLM-powered applications. It provides building blocks: document loaders, text splitters, retrievers, chains, and memory — these let us connect embeddings + vector DB + LLMs in a reliable pipeline.

2.2.2 FAISS

FAISS (Facebook AI Similarity Search) is a fast vector index library for nearest-neighbor search of high-dimensional embeddings. We store numeric document embeddings in FAISS so we can quickly retrieve the most semantically relevant chunks.

2.2.3 Ollama (local LLM hosting)

Ollama provides a local model server; you can "pull" models and run them locally. We use two model types locally:

- **Embedding model** (e.g., 'nomic-embed-text') to convert text chunks into vector embeddings.
- **LLM** (e.g., 'llama3', 'mistral') for generating answers given retrieved context.

Because models run locally with Ollama, no document text leaves the LAN.

2.2.4 Streamlit

Streamlit is a Python library for quickly building interactive web UIs. We used it for both the Admin upload interface and the Client Q&A chat interface.

2.2.5 SQLite + bcrypt

SQLite is a light, file-based SQL DB used to store users and logs. bcrypt is used to hash passwords before storing to protect credentials.

Chapter 3

System Design and Architecture

This chapter describes the system architecture, data flow and components.

3.1 High-level Architecture

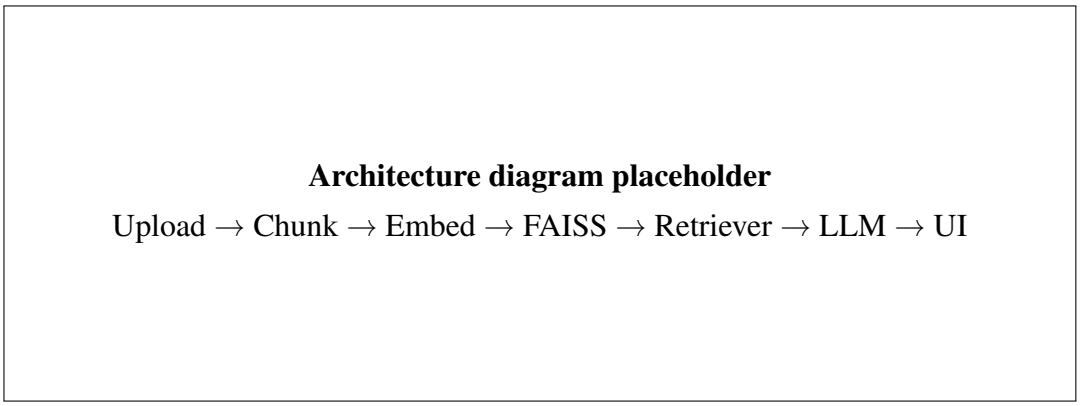


Figure 3.1: High-level system architecture.

3.2 Data Flow — Step by Step

1. **Upload (Admin)**: Admin uploads one or more PDFs through Streamlit.
2. **Storage**: Each uploaded file is saved under ‘uploaded$_files/‘andgivenastablehash - basedfilename$.
2. **Chunking**: PDF text is split into overlapping chunks (e.g. 1000 char chunks with 200 overlap). Overlap ensures context is preserved across chunk boundaries.
3. **Embedding**: Each chunk is converted to a numeric embedding vector using a local embedding model (Ollama embedding model).

4. **Indexing**: Embeddings are stored in FAISS. We persist the FAISS index to disk (pickle or FAISS files) for reuse.

5. **Retrieval**: For each user query, the retriever finds top-k nearest chunks by vector similarity.

6. **Generation**: The selected chunks are concatenated into the LLM prompt (with instruction template). The local LLM generates the final answer.

7. **Logging**: The query, user id and metadata are recorded in the SQLite ‘logs‘ table for auditing.

3.3 Why chunking, overlap, and top-k matter?

3.3.1 Chunk Size

Chunk size determines granularity. Small chunks mean more precise retrieval but more index entries and longer retrieval time. Larger chunks keep more context but risk retrieving irrelevant extra text.

3.3.2 Chunk Overlap

Overlap carries context from the end of one chunk into the next, preventing lost context for sentences split across chunks.

3.3.3 Top-k Passages

Top-k controls how many retrieved passages are given to the LLM. Increasing k gives the LLM more material but also risks overloading the prompt and introducing contradictions. Typical k = 3–8.

Chapter 4

Implementation

This chapter provides an implementation walkthrough and explains the important code components (Streamlit + LangChain + FAISS + Ollama + SQLite).

4.1 Project layout (recommended)

```
project/
 app.py                # Streamlit multi-user app (admin/client)
 init_users.py         # helper to precreate admin/client
 uploaded_files/       # saved PDFs
 vectorstores/         # saved FAISS index (combined_index.pkl)
 app_data.db           # SQLite DB for credentials & logs
 requirements.txt
 README.md
```

4.1.1 Key Python modules

- `langchain_community.document_loaders.PyPDFLoader` — extracts text from PDF.
- `RecursiveCharacterTextSplitter` — chunk text into overlapping chunks.
- `OllamaEmbeddings` — remote call to local Ollama embedding endpoint.
- `FAISS` — vectorstore class for storing and retrieving embeddings.
- `ConversationalRetrievalChain` — LangChain chain that handles retrieval + LLM generation.

4.2 Important code excerpts (explanations)

Below are representative snippets and explanations. The full app source is included in the GitHub repository (placeholder).

4.2.1 Saving uploads and hashing (stability for caching)

```
def save_upload_and_hash(uploaded_file):
    file_hash = get_file_hash(uploaded_file)
    os.makedirs("uploaded_files", exist_ok=True)
    save_path = f"uploaded_files/{file_hash}_{uploaded_file.name}"
    with open(save_path, "wb") as f:
        f.write(uploaded_file.getbuffer())
    return save_path, file_hash
```

Explanation: This generates a stable MD5 hash of the file bytes and uses it in the saved filename. If the same PDF is uploaded again, the pre-existing file and index will be reused (caching).

4.2.2 Chunking using RecursiveCharacterTextSplitter

```
splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000, chunk_overlap=200, is_separator_regex=False)
chunks = splitter.split_documents(documents)
```

Explanation: Splits long text into overlapping chunks for embedding and indexing.

4.2.3 Building/Loading FAISS index

```
def build_or_load_index(pdf_paths):
    faiss_file = "vectorstores/combined_index.pkl"
    if os.path.exists(faiss_file):
        return pickle.load(open(faiss_file,"rb"))
    # otherwise build...
    embeddings = OllamaEmbeddings(model="nomic-embed-text")
    vectorstore = FAISS.from_documents(all_docs, embeddings)
    pickle.dump(vectorstore, open(faiss_file,"wb"))
    return vectorstore
```

Explanation: We persist the entire vectorstore to disk. Once built, clients can load it immediately without re-embedding.

4.2.4 Creating the QA chain

```
def make_chain(vectorstore, llm_model="llama3"):
    llm = Ollama(model=llm_model)
    return ConversationalRetrievalChain.from_llm(
        llm=llm, retriever=vectorstore.as_retriever(search_kwargs={"k
            ↪ ":5}),
        return_source_documents=True
    )
```

Explanation: Wraps the retriever and LLM into a single callable chain that returns both the answer and the source documents for citation.

4.3 Security: credentials and logs

We use SQLite to store:

- `users` table: username, password hash, role (admin/client).
- `logs` table: username, action, details, timestamp.

Passwords are hashed with bcrypt before storage. Admin-only actions (uploading, clearing data, managing users) are gated by role checks.

Chapter 5

Use-Cases: Legal and Tax Advisory Examples

This section describes realistic questions and how the system will respond.

5.1 Example 1: Tax query for a layman

User question: "Do I need to register for income tax if my freelance income is $2,500 per year?"

System flow:

1. Retriever finds paragraphs from Income Tax Act or local tax guide discussing registration thresholds.
2. LLM generates a short, plain-language answer summarizing the clause and cites the page and paragraph.
3. The answer includes a friendly follow-up: "Would you like to see the exact clause?"

5.2 Example 2: Company policy question

User question: "What's the penalty for violating the confidentiality clause in our employee handbook?"

Result: The system retrieves the confidentiality section and summarizes punitive measures, referencing the handbook page number.

Chapter 6

Testing, Performance, and Limitations

6.1 Functional Testing

- Upload PDFs of known laws and test retrieval of target clauses.
- Confirm stored FAISS index loads between restarts.
- Verify admin-only actions require the admin role.
- Confirm logs record queries and uploads correctly.

6.2 Performance Considerations

- **Indexing time:** Building embeddings for hundreds of pages can take minutes per PDF depending on local CPU/GPU and the embedding model. Persist the index to avoid reprocessing.
- **Query latency:** Retrieving top-k and generating an answer locally takes on the order of 1–10 seconds depending on model size.
- **Memory:** FAISS and LLMs can be memory intensive. Ensure the host machine has adequate RAM.

6.3 Known Limitations

- LLM may sometimes hallucinate even with RAG — always inspect ‘source$_{d}ocuments‘fortraceabili$
- Very long PDFs produce many chunks; manage index size or selectively index only necessary sections.

- Requires Ollama model(s) to be downloaded locally (disk space).

Chapter 7

Deployment: Running Locally on a Wi-Fi Network

This section explains step-by-step local deployment so multiple PCs inside a local area network (LAN) can access the app while all data remains on-premise.

7.1 Prerequisites (Host machine)

- A host PC that will run the Streamlit app and Ollama server. Prefer Linux or Windows with enough RAM.
- Python 3.10+ environment (conda recommended).
- Ollama installed and running (models pulled locally).
- FAISS, LangChain, Streamlit, bcrypt installed in the Python environment.

7.2 Host setup commands (example)

```
# create environment (example using conda)
conda create -n local_ai python=3.10 -y
conda activate local_ai

# install dependencies (example pip)
pip install streamlit langchain langchain-community faiss-cpu ollama-
    ↪ client bcrypt
# install other libs as needed (pypdf, sentence-transformers if used, etc
    ↪ .)

# make sure Ollama is installed and models pulled
```

```
# ollama pull nomic-embed-text
# ollama pull llama3
```

7.3 Starting services

1. Start Ollama server (follow Ollama docs). Typically:

   ```
   ollama serve
   ```

2. Start Streamlit app and bind to host IP:

   ```
   streamlit run app.py --server.port 8501 --server.address 0.0.0.0
   ```

   The ‘0.0.0.0‘ address exposes it to other devices on the LAN (Wi-Fi). The host machine’s firewall must allow incoming connections to port 8501.

3. On client machines, open a web browser and navigate to:

   ```
   http://<host-ip>:8501
   ```

   where ‘¡host-ip¿‘ is the local IP of the host machine (e.g., ‘192.168.1.12‘).

7.4 Security notes for LAN deployment

- Because the app is exposed on LAN, ensure the Wi-Fi network is trusted and secured (WPA2/3, network isolation).
- Consider running the host behind a local reverse proxy with HTTPS if required.
- Keep model files and indexes on the host; do not share raw PDFs over the network unless required.

Chapter 8

Admin Features and User Management

8.1 Overview

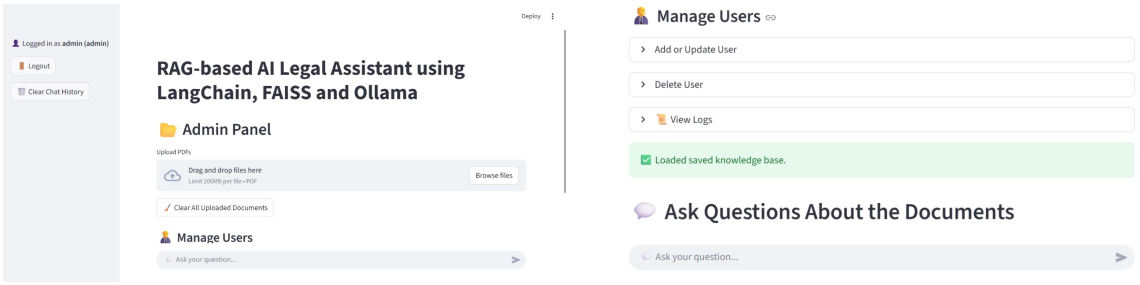
The system is implemented in Python using Streamlit and LangChain. It runs fully offline with Ollama-hosted local models.

8.2 Login Interface

Figure 8.1: Login Interface — Secure login for Admin and Client roles.

This interface allows users to log in with pre-created credentials (admin or client). Passwords are hashed and stored in SQLite.

8.3 Admin Interface



(a) Upload and Indexing (b) User Management and Logs

Figure 8.2: Admin Interface — Upload, Index, and Manage Legal Knowledge.

Admins can upload legal documents, rebuild or clear the FAISS index, manage user credentials, and view logs of all client interactions.

8.4 Client Interface

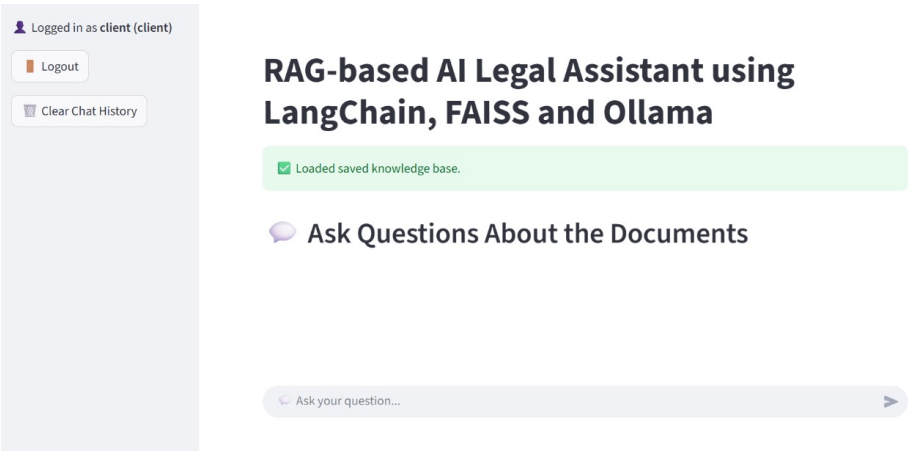


Figure 8.3: Client Interface — Ask questions and get AI-generated legal responses.

Clients interact with the assistant using a chat-style Q&A interface. Answers are derived from uploaded documents and cite specific source paragraphs for transparency.

8.5 Admin Responsibilities

- Upload legal documents (PDFs) via the Admin UI.

- Build or rebuild the FAISS index (the app caches a `combined_index.pkl` file for quick loading).
- Clear all uploaded documents and indexes (useful to replace the old knowledge base).
- Manage users: create, delete, or update credentials (stored securely in the database).
- View logs and audit queries.

8.6 How Persistence Works

- Uploaded files are stored permanently under the folder `uploaded_files/` until an admin clears them.
- The FAISS index is saved under `vectorstores/combined_index.pkl`, so the next startup doesn't require re-indexing.
- The SQLite database (`app_data.db`) permanently stores credentials and user logs.
- Logging enables the admin to audit who performed which actions and when.

Chapter 9

Extending the System: Adding Users and Changing Passwords

9.1 Via Admin UI

We added a simple admin "Manage Users" interface:

- Add or update a user: specify username, password and role.
- Delete a user (admin account protected from deletion).

9.2 Programmatically (script)

Alternatively, you can run a Python script to add users:

```
# init_users.py (example)
from your_app_module import add_user
add_user("newuser","password123","client")
```

This uses the same bcrypt hashing function to securely store the password.

Chapter 10

Source Code and Repository

10.1 Repository structure

- `app.py` — the main Streamlit app.
- `requirements.txt` — pinned package versions.
- `README.md` — instructions: how to install, run Ollama, pull models, run Streamlit, and how to use the admin interface.
- `init_users.py` — optional helper to create default accounts.
- `LICENSE` — if you plan to share.
- `docs/` — screenshots, sample transcripts, testing notes.

10.2 Placeholder GitHub link

`https://github.com/daniyalmehmood2244-lab/rag-local-legal-assistant`

Chapter 11

Ethics, Privacy and Legal Considerations

11.1 Why local-first matters

Legal documents and client queries can contain sensitive information. A local deployment means documents and queries never leave your controlled network — this reduces exposure and is preferable for legal practice.

11.2 Limitations and responsibilities

- The system provides **informational** answers and cannot replace qualified legal advice.
- Always show retrieved source passages and remind users to consult a lawyer for final decisions.
- Maintain regular software updates and security patches for the host machine.

Chapter 12

Conclusion and Future Work

12.1 Conclusion

This project demonstrates a practical, privacy-preserving implementation of RAG for legal and tax advisory use. By combining LangChain orchestration, FAISS indexing, Ollama local models, and Streamlit for UI, we produced a system that:

- Answers questions grounded in uploaded source documents.
- Runs entirely on local infrastructure.
- Provides admin/client separation with persistent indices and logs.

12.2 Future work

- Add per-document metadata filters (search by jurisdiction, year).
- Support multi-lingual documents and embeddings.
- Add role-based granular permissions.
- Add more robust UI (pagination of sources, improved citation styling).
- Integrate differential privacy or audit controls for higher security.

Bibliography

[1] Lewis, Patrick, et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” (2020)

[2] Johnson, Jeff, Matthijs Douze, and Hervé Jégou. “Billion-scale similarity search with GPUs.” (2019)

[3] LangChain documentation. https://python.langchain.com

[4] Ollama documentation. https://ollama.com