RAPPORT

EMI/emiAI : *Intelligent Document-Based Question Answering System for Educational*

Réalisé Par

Anas Oubassou &

EL Maskaoui Oussama

Encadré par

Dr Aniba Ghassane

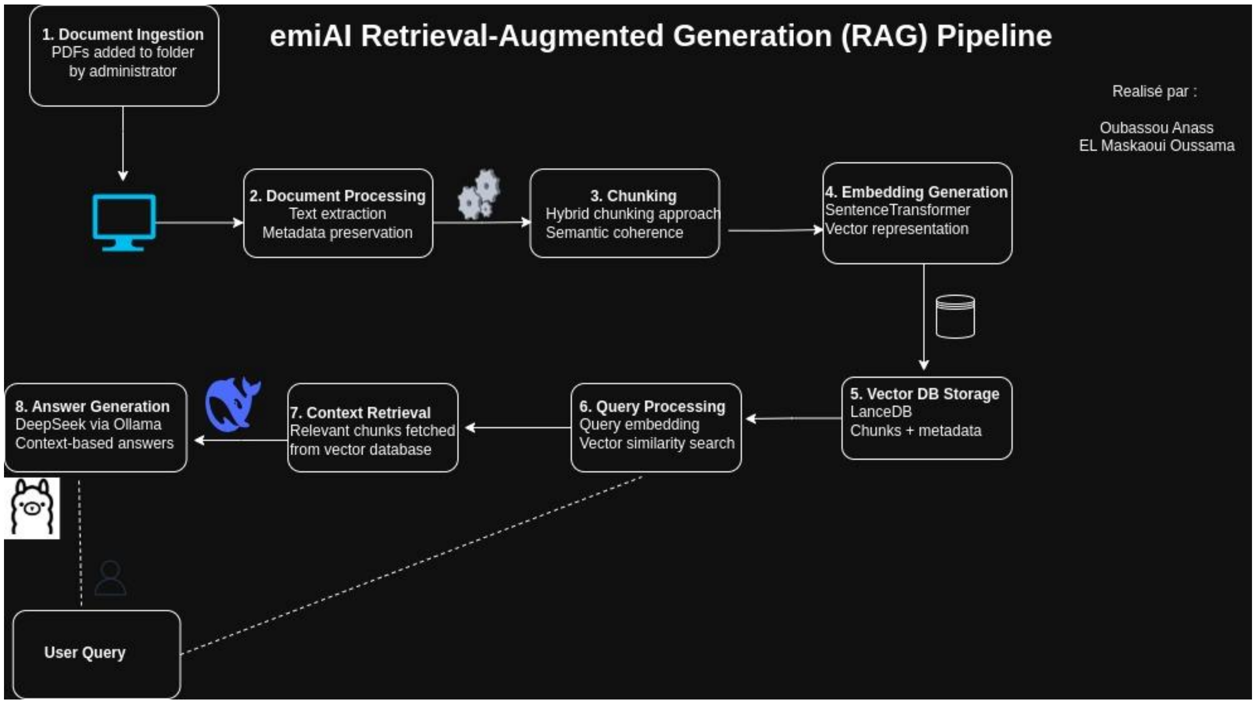
1. Project Purpose

The EMI/emiAI project is a specialized Question Answering (QA) system designed for educational contexts. Its primary purpose is to serve as an intelligent assistant that allows students to query and interact with course documents provided by professors. The system follows these key principles:

- **Document-Grounded Responses** : All answers are strictly sourced from uploaded course PDFs, preventing hallucinations or external information
- **Privacy-Focused**: Uses a local knowledge base derived from PDF files, allowing offline usage and maintaining data privacy
- **Instructor-Controlled**: Only administrators (professors/TAs) can upload documents, ensuring students interact with authorized materials
- **Contextual Understanding**: Employs Retrieval Augmented Generation (RAG) to find relevant document sections and generate coherent, accurate answers
- **Accessibility**: Transforms static PDF documents into an interactive knowledge source through a simple chat interface

The system bridges the gap between traditional course materials and modern interactive learning tools, making document information more accessible without requiring students to manually search through pages of text.

2. RAG Pipeline Explanation



The RAG (Retrieval Augmented Generation) pipeline in emiAI consists of the following sequential steps:

1. **Document Ingestion**:
   - PDF files are placed in a designated folder by the administrator
   - The system reads and processes these files to extract their text content
2. **Document Processing**:
   - Each PDF is converted to text while preserving document structure
   - The system extracts metadata such as page numbers and section headings

3. **Chunking**:
    - Documents are divided into smaller, semantically meaningful chunks
    - A hybrid chunking approach ensures chunks maintain coherence while staying within token limits
4. **Embedding Generation**:
    - Each chunk is converted to a vector representation (embedding) using SentenceTransformer
    - These numerical representations capture the semantic meaning of each chunk
5. **Vector Database Storage**:
    - Chunks and their embeddings are stored in a LanceDB vector database
    - Metadata like filename, page numbers, and headings are preserved alongside each chunk
6. **Query Processing**:
    - When a user asks a question, their query is converted to the same embedding space
    - The system performs a vector similarity search to find relevant chunks
7. **Context Retrieval**:
    - The most semantically similar chunks are retrieved from the database
    - These chunks serve as context for answering the user's question

8. **Answer Generation**:

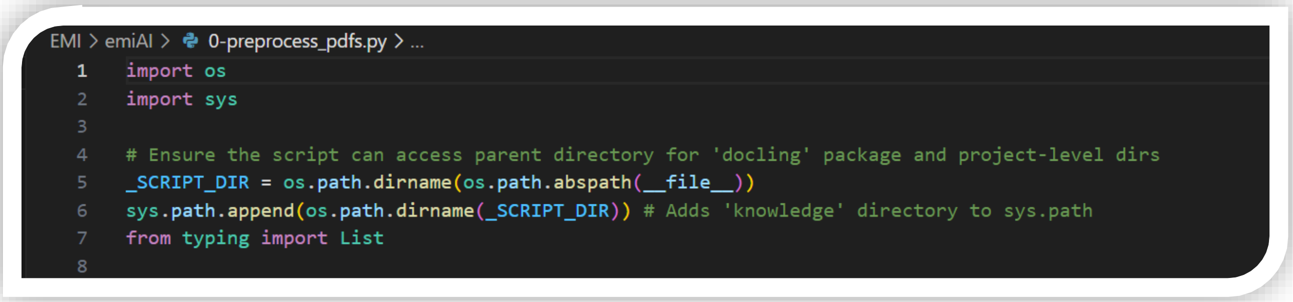
   - The retrieved context and user question are sent to a language model (DeepSeek via Ollama)
   - The model generates an answer based strictly on the provided context
   - Source information (filename, page numbers) is displayed alongside the answer

This pipeline ensures answers are grounded in the actual course materials, providing accurate, contextual responses while preventing hallucinations.

3. Code Analysis

❖ File: 0-preprocess_pdfs.py

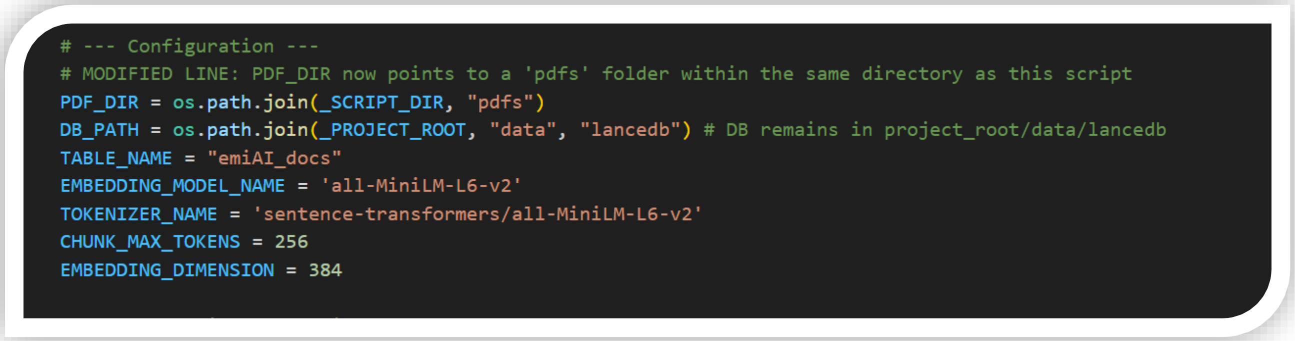
Section 1: Imports and Path Setup



```python
import os
import sys

# Ensure the script can access parent directory for 'docling' package and project-level dirs
_SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
sys.path.append(os.path.dirname(_SCRIPT_DIR)) # Adds 'knowledge' directory to sys.path
from typing import List

```

➔ This section handles the necessary imports and sets up the Python path to ensure the script can access the required packages. The docling package is a critical dependency for document processing.

Section 2: Configuration Setup

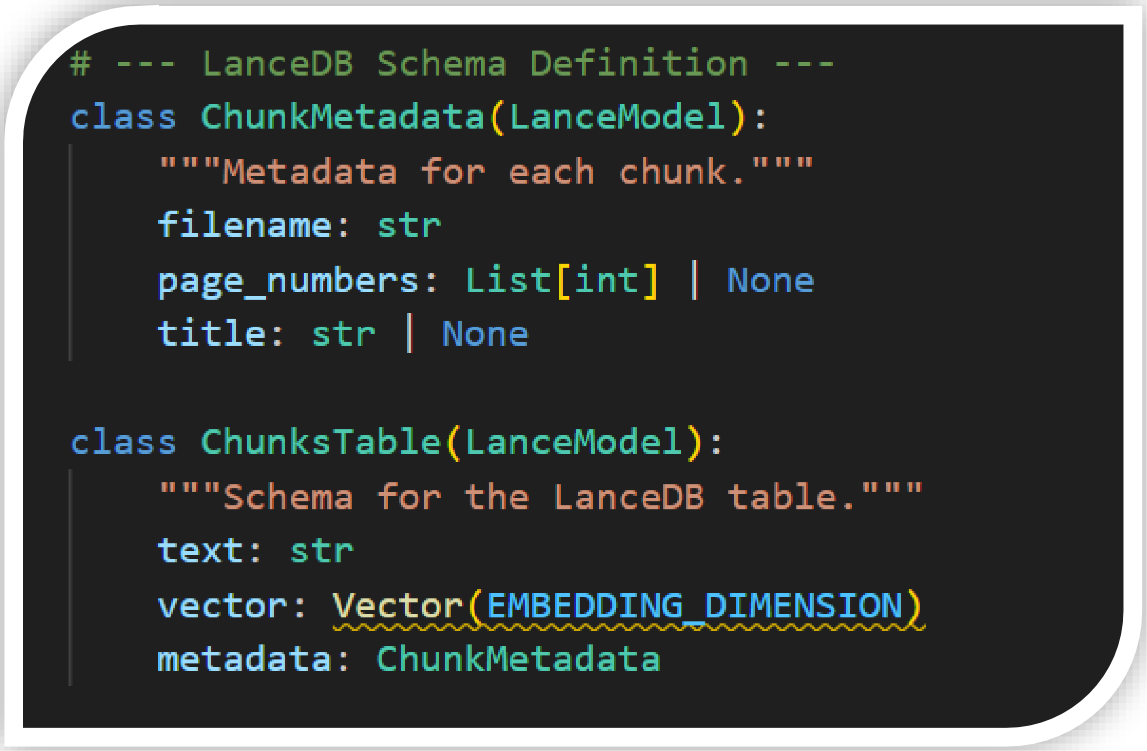

```python
# --- Configuration ---
# MODIFIED LINE: PDF_DIR now points to a 'pdfs' folder within the same directory as this script
PDF_DIR = os.path.join(_SCRIPT_DIR, "pdfs")
DB_PATH = os.path.join(_PROJECT_ROOT, "data", "lancedb") # DB remains in project_root/data/lancedb
TABLE_NAME = "emiAI_docs"
EMBEDDING_MODEL_NAME = 'all-MiniLM-L6-v2'
TOKENIZER_NAME = 'sentence-transformers/all-MiniLM-L6-v2'
CHUNK_MAX_TOKENS = 256
EMBEDDING_DIMENSION = 384
```

→

This section defines the critical configuration parameters for the preprocessing script, including:

- PDF directory location where documents are stored
- Database path for LanceDB vector storage
- Table name for document chunks
- Embedding model configuration (using SentenceTransformer)
- Chunking parameters (maximum token count per chunk)

Section 3: Database Schema Definition



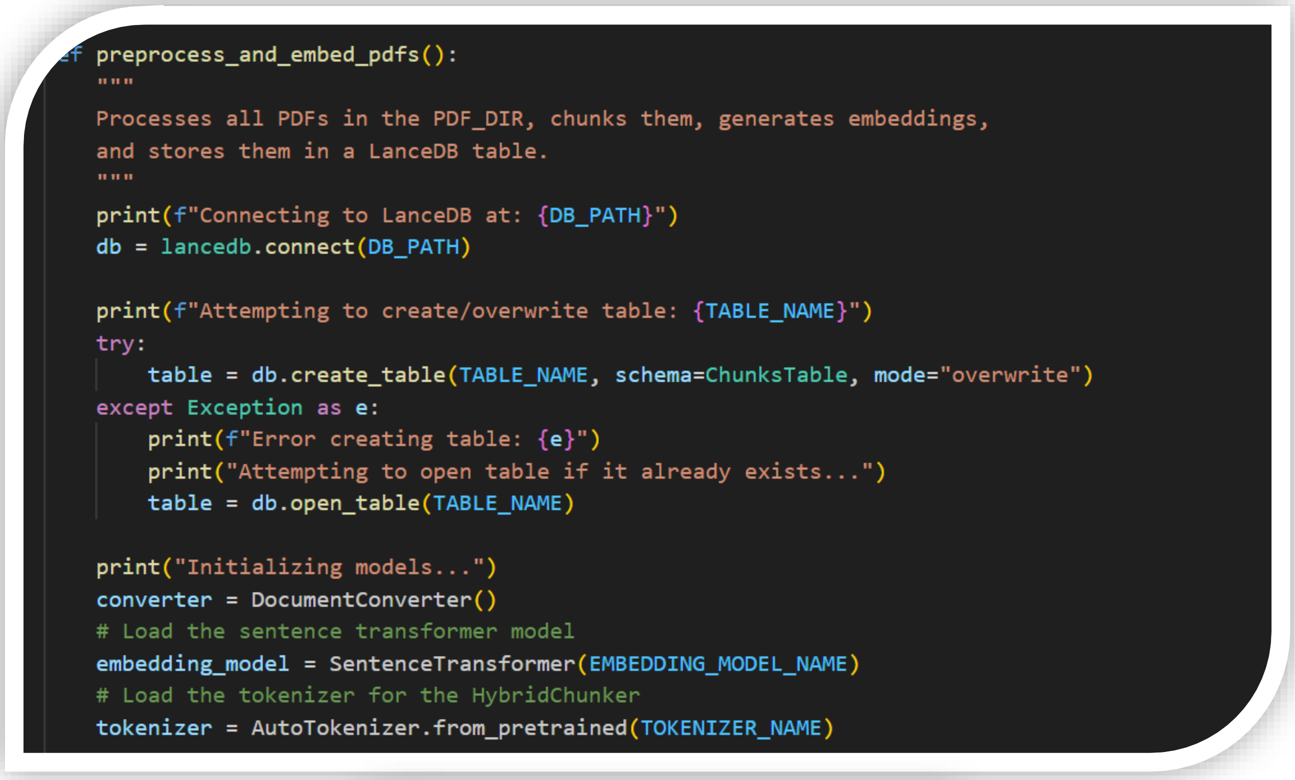
```python
# --- LanceDB Schema Definition ---
class ChunkMetadata(LanceModel):
    """Metadata for each chunk."""
    filename: str
    page_numbers: List[int] | None
    title: str | None

class ChunksTable(LanceModel):
    """Schema for the LanceDB table."""
    text: str
    vector: Vector(EMBEDDING_DIMENSION)
    metadata: ChunkMetadata
```

➔

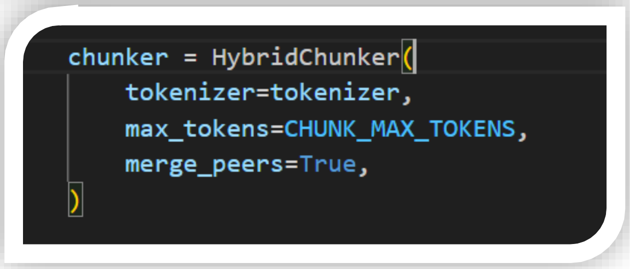
This section defines the database schema using Pydantic models for LanceDB. It specifies:

- **ChunkMetadata** : Stores information about the source document (filename, page numbers, and title)
- **ChunksTable** : The main table schema with text content, vector embeddings, and associated metadata

Section 4: Document Processing Function



```python
ef preprocess_and_embed_pdfs():
    """
    Processes all PDFs in the PDF_DIR, chunks them, generates embeddings,
    and stores them in a LanceDB table.
    """
    print(f"Connecting to LanceDB at: {DB_PATH}")
    db = lancedb.connect(DB_PATH)

    print(f"Attempting to create/overwrite table: {TABLE_NAME}")
    try:
        table = db.create_table(TABLE_NAME, schema=ChunksTable, mode="overwrite")
    except Exception as e:
        print(f"Error creating table: {e}")
        print("Attempting to open table if it already exists...")
        table = db.open_table(TABLE_NAME)

    print("Initializing models...")
    converter = DocumentConverter()
    # Load the sentence transformer model
    embedding_model = SentenceTransformer(EMBEDDING_MODEL_NAME)
    # Load the tokenizer for the HybridChunker
    tokenizer = AutoTokenizer.from_pretrained(TOKENIZER_NAME)
```



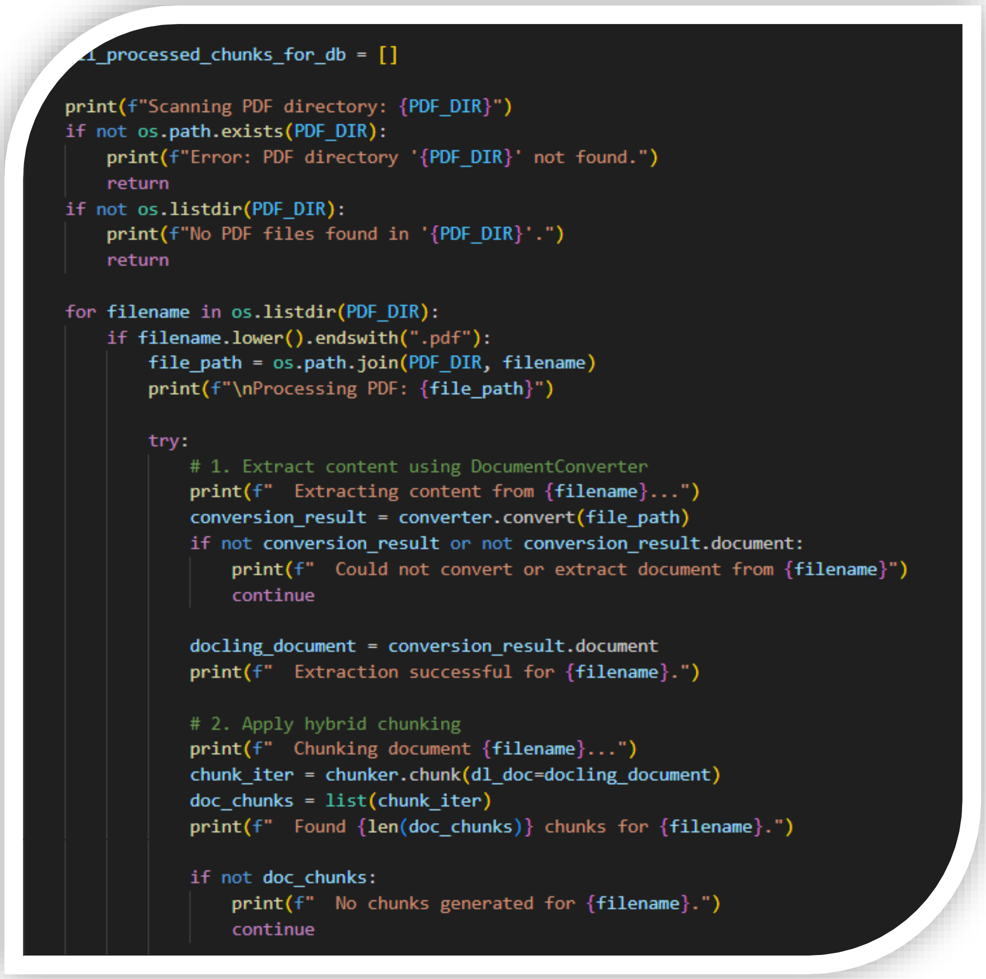
```python
chunker = HybridChunker(
    tokenizer=tokenizer,
    max_tokens=CHUNK_MAX_TOKENS,
    merge_peers=True,
)
```

→

This section initializes the database connection and required models:

- Connects to LanceDB and creates/opens the table
- Initializes the document converter for PDF processing
- Loads the sentence transformer for embedding generation
- Sets up the hybrid chunker with the specified parameters

Section 5: Document Processing Loop



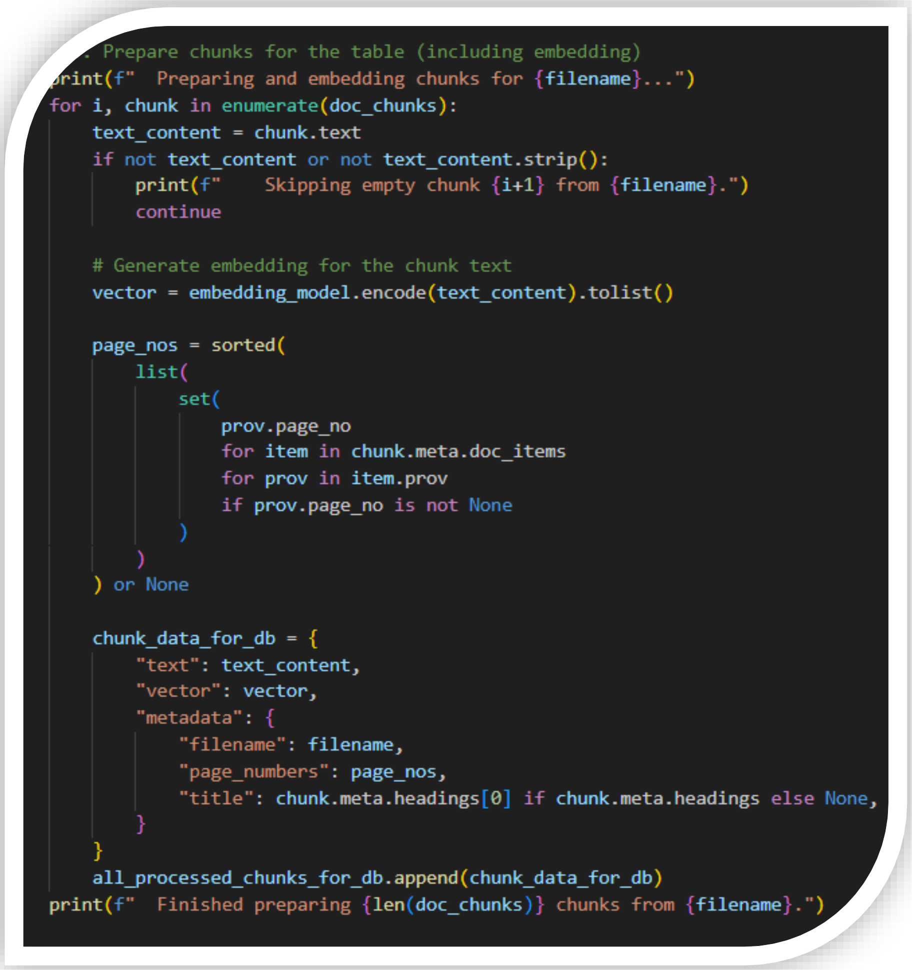
```
l_processed_chunks_for_db = []

print(f"Scanning PDF directory: {PDF_DIR}")
if not os.path.exists(PDF_DIR):
    print(f"Error: PDF directory '{PDF_DIR}' not found.")
    return
if not os.listdir(PDF_DIR):
    print(f"No PDF files found in '{PDF_DIR}'.")
    return

for filename in os.listdir(PDF_DIR):
    if filename.lower().endswith(".pdf"):
        file_path = os.path.join(PDF_DIR, filename)
        print(f"\nProcessing PDF: {file_path}")

        try:
            # 1. Extract content using DocumentConverter
            print(f"  Extracting content from {filename}...")
            conversion_result = converter.convert(file_path)
            if not conversion_result or not conversion_result.document:
                print(f"  Could not convert or extract document from {filename}")
                continue

            docling_document = conversion_result.document
            print(f"  Extraction successful for {filename}.")

            # 2. Apply hybrid chunking
            print(f"  Chunking document {filename}...")
            chunk_iter = chunker.chunk(dl_doc=docling_document)
            doc_chunks = list(chunk_iter)
            print(f"  Found {len(doc_chunks)} chunks for {filename}.")

            if not doc_chunks:
                print(f"  No chunks generated for {filename}.")
                continue
```

→

This section handles the document processing loop:

- Checks if the PDF directory exists and contains files
- Iterates through each PDF file in the directory
- Extracts content using the DocumentConverter
- Applies hybrid chunking to break documents into manageable pieces
- Provides detailed logging for monitoring the process

Section 6: Chunk Embedding and Metadata Extraction



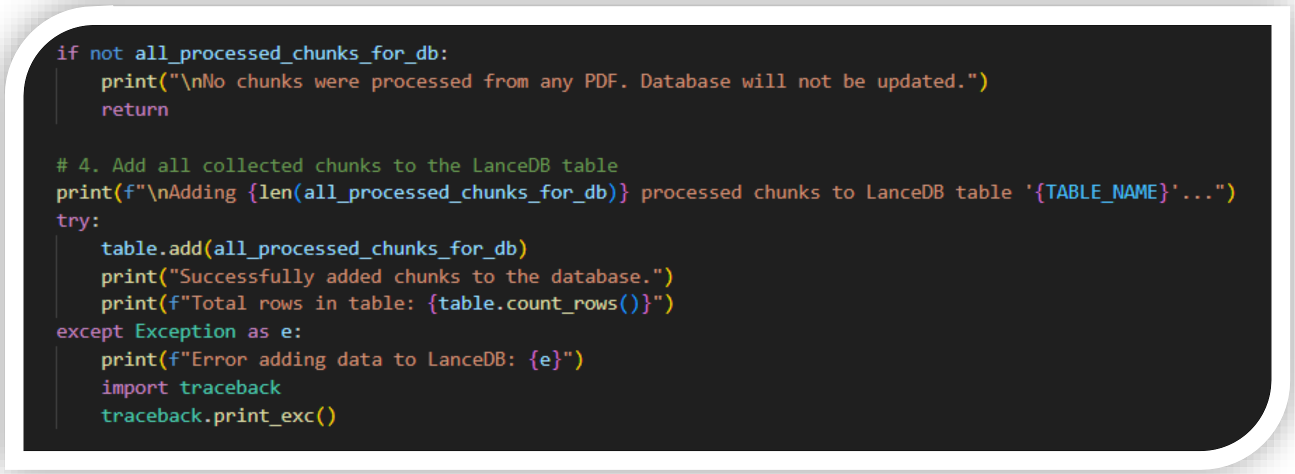
```python
. Prepare chunks for the table (including embedding)
print(f"  Preparing and embedding chunks for {filename}...")
for i, chunk in enumerate(doc_chunks):
    text_content = chunk.text
    if not text_content or not text_content.strip():
        print(f"    Skipping empty chunk {i+1} from {filename}.")
        continue

    # Generate embedding for the chunk text
    vector = embedding_model.encode(text_content).tolist()

    page_nos = sorted(
        list(
            set(
                prov.page_no
                for item in chunk.meta.doc_items
                for prov in item.prov
                if prov.page_no is not None
            )
        )
    ) or None

    chunk_data_for_db = {
        "text": text_content,
        "vector": vector,
        "metadata": {
            "filename": filename,
            "page_numbers": page_nos,
            "title": chunk.meta.headings[0] if chunk.meta.headings else None,
        }
    }
    all_processed_chunks_for_db.append(chunk_data_for_db)
print(f"  Finished preparing {len(doc_chunks)} chunks from {filename}.")
```

→

This section handles embedding generation and metadata extraction:

- Creates vector embeddings for each chunk using SentenceTransformer
- Extracts page numbers by collecting unique page references
- Organizes chunk data with text, vector, and metadata
- Adds processed chunks to a collection for database insertion

Section 7: Database Storage

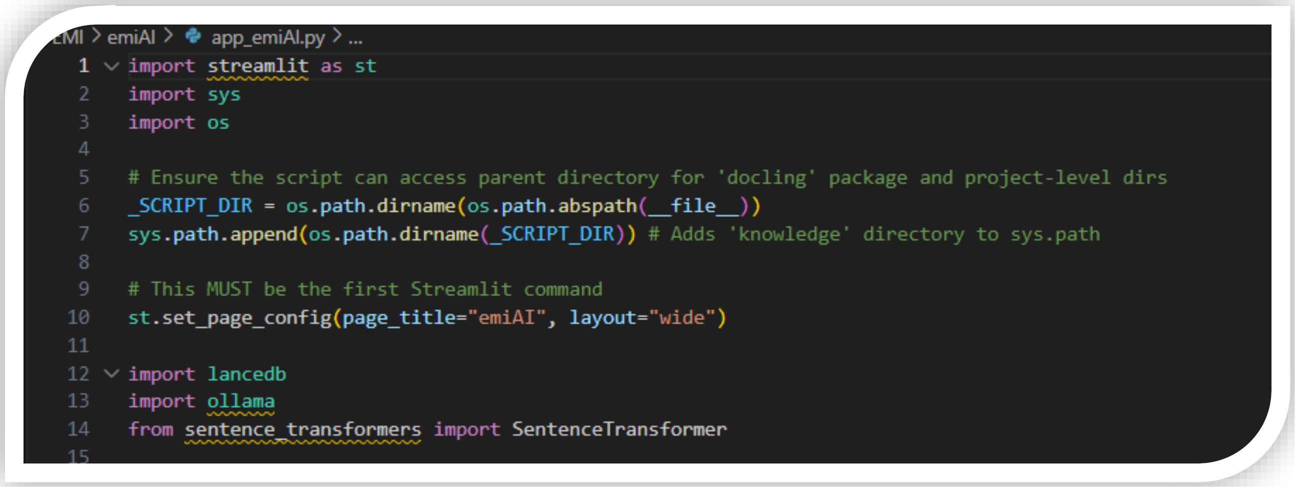

```
if not all_processed_chunks_for_db:
    print("\nNo chunks were processed from any PDF. Database will not be updated.")
    return

# 4. Add all collected chunks to the LanceDB table
print(f"\nAdding {len(all_processed_chunks_for_db)} processed chunks to LanceDB table '{TABLE_NAME}'...")
try:
    table.add(all_processed_chunks_for_db)
    print("Successfully added chunks to the database.")
    print(f"Total rows in table: {table.count_rows()}")
except Exception as e:
    print(f"Error adding data to LanceDB: {e}")
    import traceback
    traceback.print_exc()
```

This section handles the database storage:

- Checks if any chunks were processed
- Bulk adds all processed chunks to the LanceDB table
- Reports success or failure with detailed error information
- Provides a count of total rows in the table for verification

❖ **File: app_emiAI.py**

Section 1: Imports and Setup



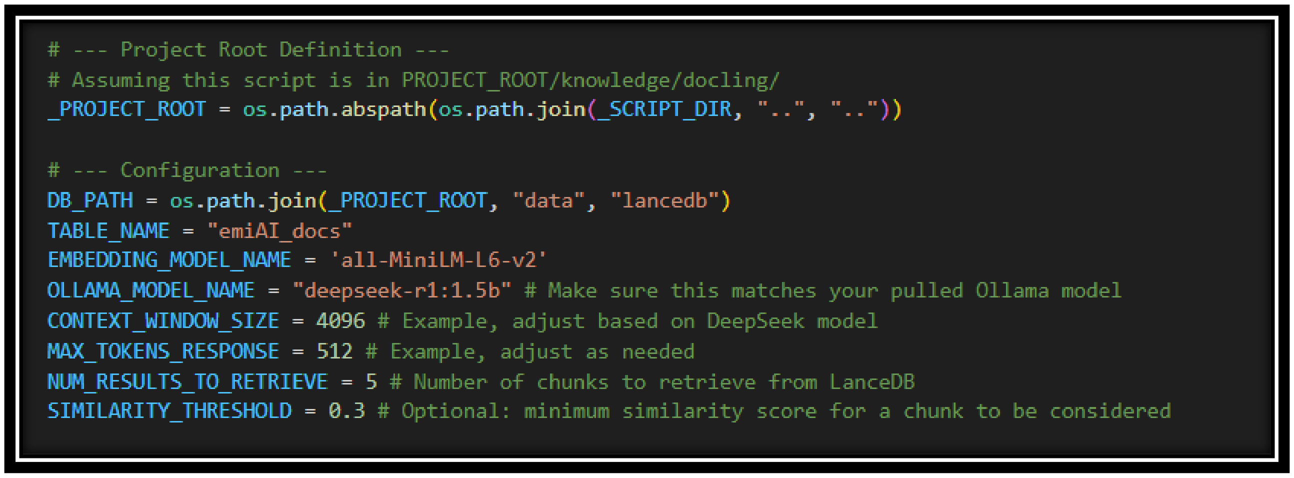
EMI > emiAI > app_emiAI.py > ...

```python
import streamlit as st
import sys
import os

# Ensure the script can access parent directory for 'docling' package and project-level dirs
_SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))
sys.path.append(os.path.dirname(_SCRIPT_DIR)) # Adds 'knowledge' directory to sys.path

# This MUST be the first Streamlit command
st.set_page_config(page_title="emiAI", layout="wide")

import lancedb
import ollama
from sentence_transformers import SentenceTransformer

```

This section sets up the Streamlit application and imports necessary packages:

- Imports Streamlit for the web interface
- Sets up Python path for accessing required packages
- Configures the Streamlit page layout
- Imports LanceDB for vector database access
- Imports Ollama for local language model interaction
- Imports SentenceTransformer for generating query embeddings

Section 2: Configuration Settings

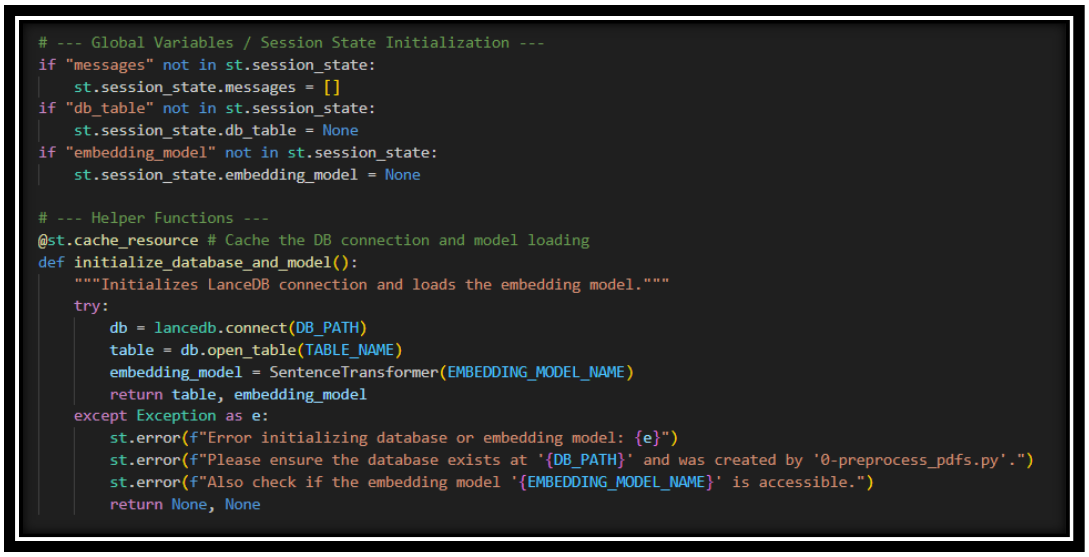


```
# --- Project Root Definition ---
# Assuming this script is in PROJECT_ROOT/knowledge/docling/
_PROJECT_ROOT = os.path.abspath(os.path.join(_SCRIPT_DIR, "..", ".."))

# --- Configuration ---
DB_PATH = os.path.join(_PROJECT_ROOT, "data", "lancedb")
TABLE_NAME = "emiAI_docs"
EMBEDDING_MODEL_NAME = 'all-MiniLM-L6-v2'
OLLAMA_MODEL_NAME = "deepseek-r1:1.5b" # Make sure this matches your pulled Ollama model
CONTEXT_WINDOW_SIZE = 4096 # Example, adjust based on DeepSeek model
MAX_TOKENS_RESPONSE = 512 # Example, adjust as needed
NUM_RESULTS_TO_RETRIEVE = 5 # Number of chunks to retrieve from LanceDB
SIMILARITY_THRESHOLD = 0.3 # Optional: minimum similarity score for a chunk to be considered
```

This section defines the application configuration:

- Sets paths for database access (matching the preprocessing script)
- Specifies the embedding model for query encoding
- Configures the local LLM model via Ollama
- Sets parameters for context window size and response length
- Defines retrieval parameters (number of chunks and similarity threshold)

Section 3: Session State and Helper Functions



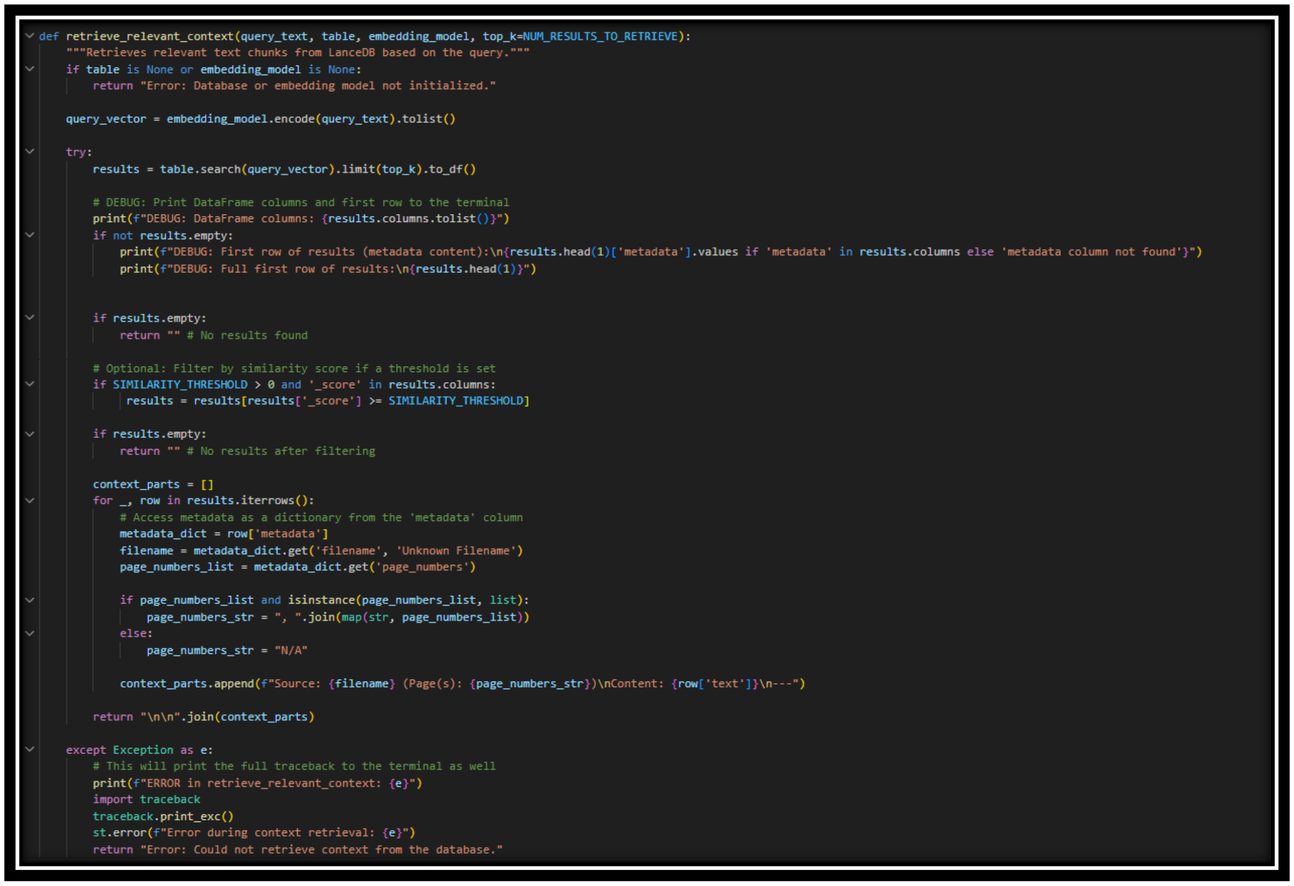
```python
# --- Global Variables / Session State Initialization ---
if "messages" not in st.session_state:
    st.session_state.messages = []
if "db_table" not in st.session_state:
    st.session_state.db_table = None
if "embedding_model" not in st.session_state:
    st.session_state.embedding_model = None

# --- Helper Functions ---
@st.cache_resource # Cache the DB connection and model loading
def initialize_database_and_model():
    """Initializes LanceDB connection and loads the embedding model."""
    try:
        db = lancedb.connect(DB_PATH)
        table = db.open_table(TABLE_NAME)
        embedding_model = SentenceTransformer(EMBEDDING_MODEL_NAME)
        return table, embedding_model
    except Exception as e:
        st.error(f"Error initializing database or embedding model: {e}")
        st.error(f"Please ensure the database exists at '{DB_PATH}' and was created by '0-preprocess_pdfs.py'.")
        st.error(f"Also check if the embedding model '{EMBEDDING_MODEL_NAME}' is accessible.")
        return None, None
```

→

This section initializes session state and defines helper functions:

- Sets up Streamlit session state for message history and resources
- Creates a cached resource function to efficiently load the database and embedding model
- Provides error handling for database connection issues
- Uses clear error messages to guide administrators in troubleshooting

Section 4: Vector Retrieval Function

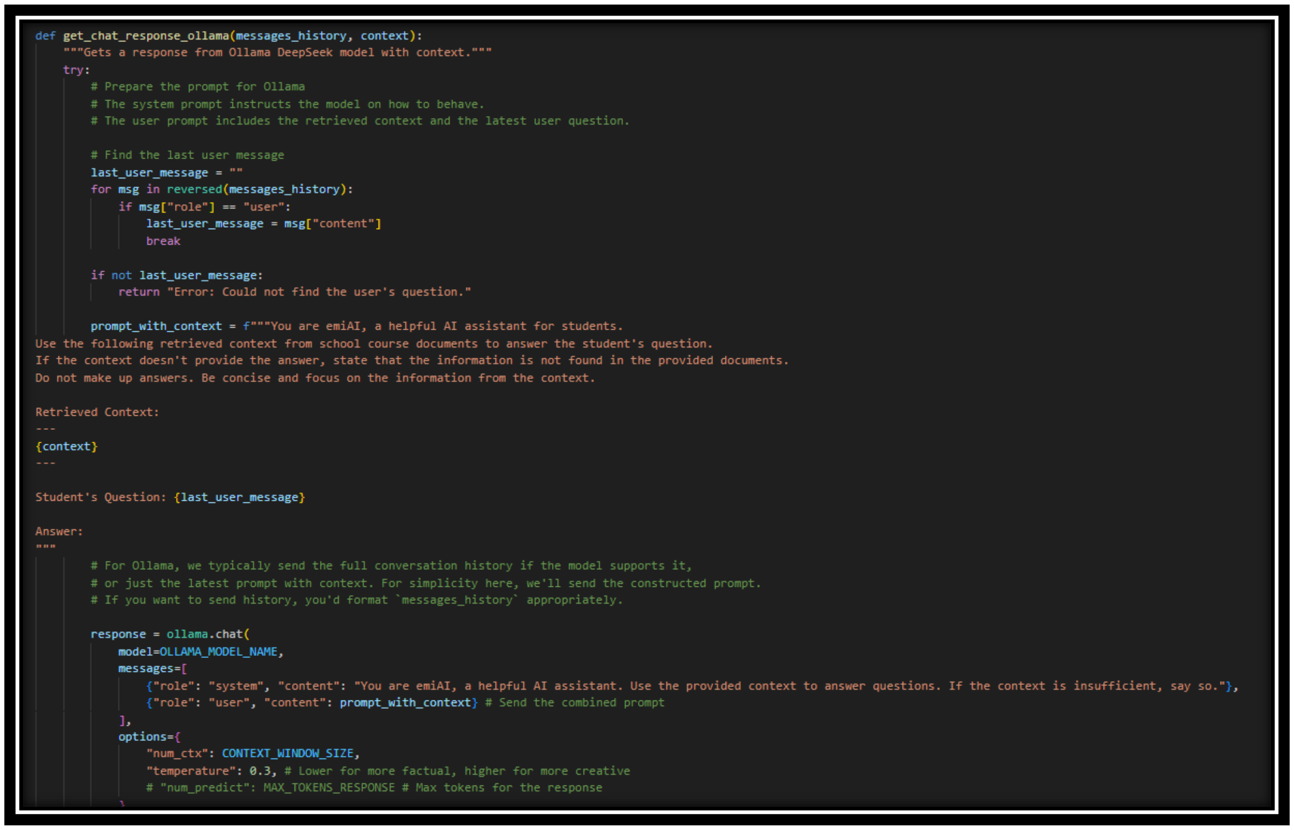


```python
def retrieve_relevant_context(query_text, table, embedding_model, top_k=NUM_RESULTS_TO_RETRIEVE):
    """Retrieves relevant text chunks from LanceDB based on the query."""
    if table is None or embedding_model is None:
        return "Error: Database or embedding model not initialized."

    query_vector = embedding_model.encode(query_text).tolist()

    try:
        results = table.search(query_vector).limit(top_k).to_df()

        # DEBUG: Print DataFrame columns and first row to the terminal
        print(f"DEBUG: DataFrame columns: {results.columns.tolist()}")
        if not results.empty:
            print(f"DEBUG: First row of results (metadata content):\n{results.head(1)['metadata'].values if 'metadata' in results.columns else 'metadata column not found'}")
            print(f"DEBUG: Full first row of results:\n{results.head(1)}")


        if results.empty:
            return "" # No results found

        # Optional: Filter by similarity score if a threshold is set
        if SIMILARITY_THRESHOLD > 0 and '_score' in results.columns:
             results = results[results['_score'] >= SIMILARITY_THRESHOLD]

        if results.empty:
            return "" # No results after filtering

        context_parts = []
        for _, row in results.iterrows():
            # Access metadata as a dictionary from the 'metadata' column
            metadata_dict = row['metadata']
            filename = metadata_dict.get('filename', 'Unknown Filename')
            page_numbers_list = metadata_dict.get('page_numbers')

            if page_numbers_list and isinstance(page_numbers_list, list):
                page_numbers_str = ", ".join(map(str, page_numbers_list))
            else:
                page_numbers_str = "N/A"

            context_parts.append(f"Source: {filename} (Page(s): {page_numbers_str})\nContent: {row['text']}\n---")

        return "\n\n".join(context_parts)

    except Exception as e:
        # This will print the full traceback to the terminal as well
        print(f"ERROR in retrieve_relevant_context: {e}")
        import traceback
        traceback.print_exc()
        st.error(f"Error during context retrieval: {e}")
        return "Error: Could not retrieve context from the database."
```

Section 5: LLM Response Generation

This section defines the vector retrieval function:

- Encodes the user query into a vector using SentenceTransformer
- Performs a similarity search in LanceDB to find relevant chunks
- Optionally filters results by similarity score threshold
- Formats retrieved chunks with source information (filename and page numbers)
- Provides detailed error handling and debugging information

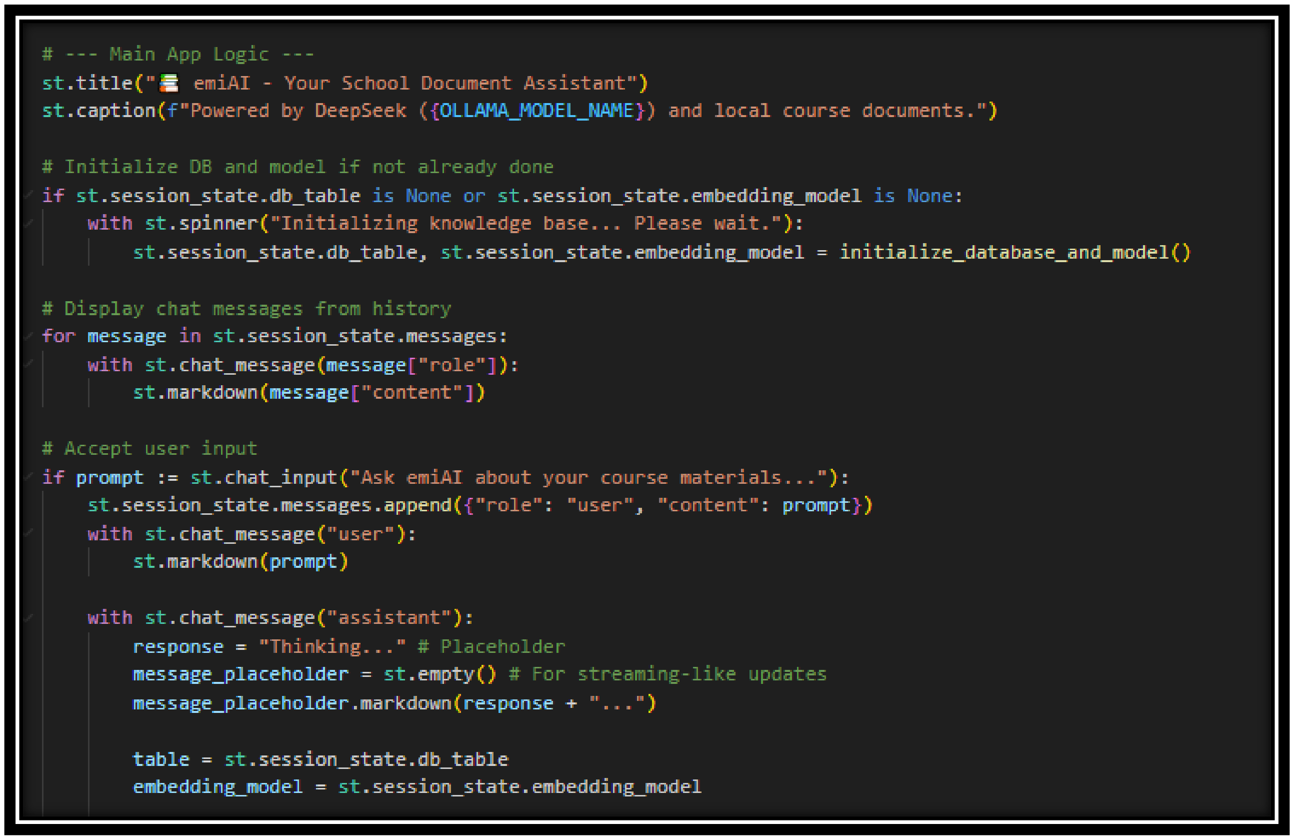


```python
def get_chat_response_ollama(messages_history, context):
    """Gets a response from Ollama DeepSeek model with context."""
    try:
        # Prepare the prompt for Ollama
        # The system prompt instructs the model on how to behave.
        # The user prompt includes the retrieved context and the latest user question.

        # Find the last user message
        last_user_message = ""
        for msg in reversed(messages_history):
            if msg["role"] == "user":
                last_user_message = msg["content"]
                break

        if not last_user_message:
            return "Error: Could not find the user's question."

        prompt_with_context = f"""You are emiAI, a helpful AI assistant for students.
Use the following retrieved context from school course documents to answer the student's question.
If the context doesn't provide the answer, state that the information is not found in the provided documents.
Do not make up answers. Be concise and focus on the information from the context.

Retrieved Context:
---
{context}
---

Student's Question: {last_user_message}

Answer:
"""
        # For Ollama, we typically send the full conversation history if the model supports it,
        # or just the latest prompt with context. For simplicity here, we'll send the constructed prompt.
        # If you want to send history, you'd format `messages_history` appropriately.

        response = ollama.chat(
            model=OLLAMA_MODEL_NAME,
            messages=[
                {"role": "system", "content": "You are emiAI, a helpful AI assistant. Use the provided context to answer questions. If the context is insufficient, say so."},
                {"role": "user", "content": prompt_with_context} # Send the combined prompt
            ],
            options={
                "num_ctx": CONTEXT_WINDOW_SIZE,
                "temperature": 0.3, # Lower for more factual, higher for more creative
                # "num_predict": MAX_TOKENS_RESPONSE # Max tokens for the response
```

This section handles LLM response generation:

- Extracts the latest user question from message history
- Constructs a prompt that includes retrieved context and clear instructions
- Uses Ollama to interact with the local DeepSeek language model
- Sets parameters for context window size and temperature
- Returns the model's response or a helpful error message

Section 6: Streamlit UI and Main Logic

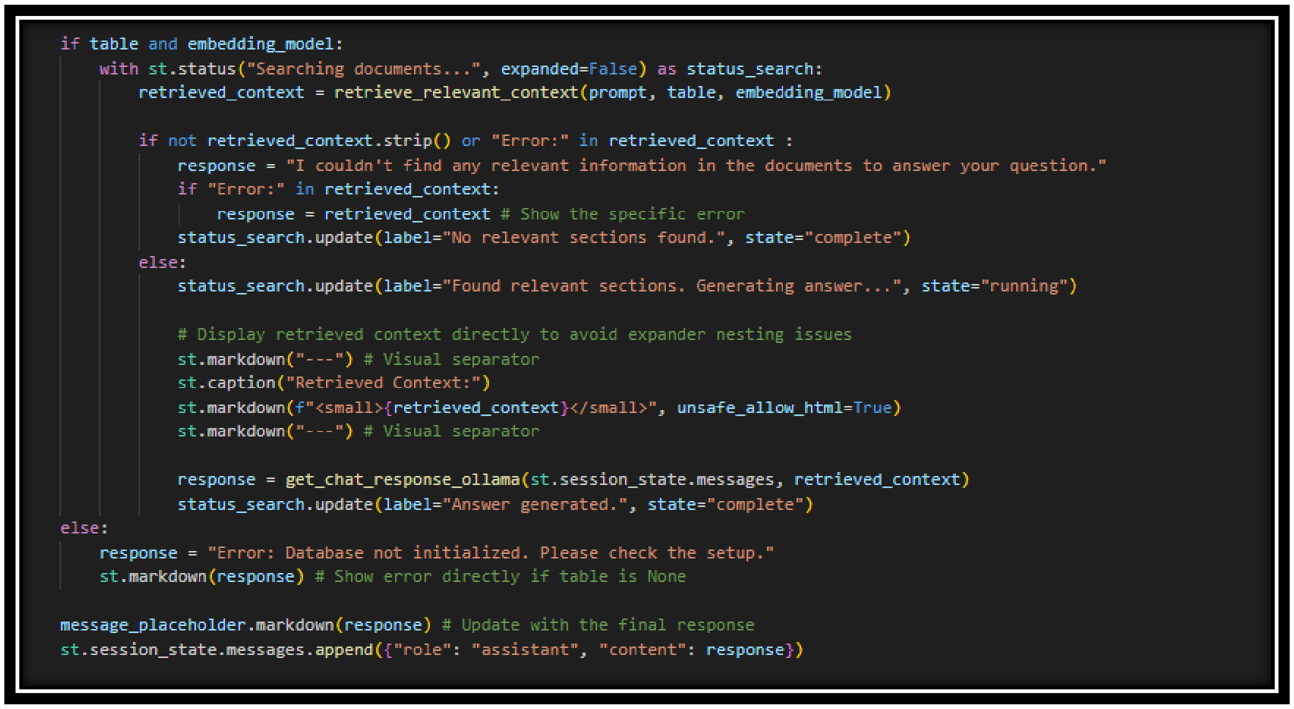


```python
# --- Main App Logic ---
st.title("📚 emiAI - Your School Document Assistant")
st.caption(f"Powered by DeepSeek ({OLLAMA_MODEL_NAME}) and local course documents.")

# Initialize DB and model if not already done
if st.session_state.db_table is None or st.session_state.embedding_model is None:
    with st.spinner("Initializing knowledge base... Please wait."):
        st.session_state.db_table, st.session_state.embedding_model = initialize_database_and_model()

# Display chat messages from history
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# Accept user input
if prompt := st.chat_input("Ask emiAI about your course materials..."):
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    with st.chat_message("assistant"):
        response = "Thinking..." # Placeholder
        message_placeholder = st.empty() # For streaming-like updates
        message_placeholder.markdown(response + "...")

        table = st.session_state.db_table
        embedding_model = st.session_state.embedding_model
```

This section builds the Streamlit UI and implements the main app logic:

- Creates a titled page with descriptive caption
- Initializes the database and embedding model if not already loaded
- Displays the chat history from previous interactions
- Accepts user input through a chat input field
- Prepares the response area with a placeholder message

Section 7: Response Generation Logic

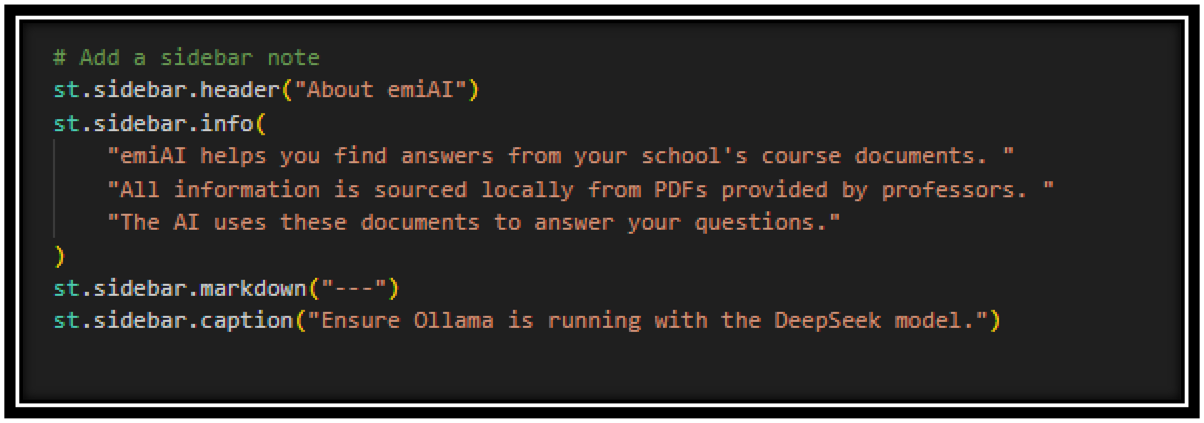


```python
if table and embedding_model:
    with st.status("Searching documents...", expanded=False) as status_search:
        retrieved_context = retrieve_relevant_context(prompt, table, embedding_model)

        if not retrieved_context.strip() or "Error:" in retrieved_context :
            response = "I couldn't find any relevant information in the documents to answer your question."
            if "Error:" in retrieved_context:
                response = retrieved_context # Show the specific error
            status_search.update(label="No relevant sections found.", state="complete")
        else:
            status_search.update(label="Found relevant sections. Generating answer...", state="running")

            # Display retrieved context directly to avoid expander nesting issues
            st.markdown("---") # Visual separator
            st.caption("Retrieved Context:")
            st.markdown(f"<small>{retrieved_context}</small>", unsafe_allow_html=True)
            st.markdown("---") # Visual separator

            response = get_chat_response_ollama(st.session_state.messages, retrieved_context)
            status_search.update(label="Answer generated.", state="complete")
else:
    response = "Error: Database not initialized. Please check the setup."
    st.markdown(response) # Show error directly if table is None

message_placeholder.markdown(response) # Update with the final response
st.session_state.messages.append({"role": "assistant", "content": response})
```

This section implements the response generation logic:

- Retrieves relevant context from the vector database
- Handles cases where no relevant information is found
- Displays the retrieved context for transparency
- Generates a response using the Ollama model and context
- Updates the message placeholder with the final response
- Adds the assistant's response to the message history

Section 8: Sidebar Information



```python
# Add a sidebar note
st.sidebar.header("About emiAI")
st.sidebar.info(
    "emiAI helps you find answers from your school's course documents. "
    "All information is sourced locally from PDFs provided by professors. "
    "The AI uses these documents to answer your questions."
)
st.sidebar.markdown("---")
st.sidebar.caption("Ensure Ollama is running with the DeepSeek model.")
```

This section adds informational elements to the sidebar:

- Provides a brief description of emiAI's purpose
- Explains that information comes from local PDFs
- Reminds users about the Ollama dependency

EMI/emiAI Administrator Guide

This guide is designed for administrators who will be responsible for setting up, maintaining, and operating the EMI/emiAI application. No prior experience with the system is required.

System Overview

EMI/emiAI is a specialized Question Answering system that allows students to ask questions about course materials. The system:

- Processes PDF documents provided by instructors
- Creates a searchable knowledge base from these documents
- Allows students to query this knowledge base through a chat interface
- Answers questions using only information found in the provided documents

Prerequisites

Before setting up EMI/emiAI, ensure you have:

1. **Python Environment**: Python 3.9+ installed on your system
2. **Git**: For cloning the project repository (optional)
3. **Ollama**: For running the local language model
    - Install from Ollama.ai
    - Make sure it's running in the background when using the application

Step 1: Initial Setup

Installing Dependencies

1. Open a terminal or command prompt
2. Navigate to the project directory
3. Create a virtual environment (recommended):

   bash

   ```
   python -m venv .venv
   ```

4. Activate the virtual environment:
   - Windows: .venv\Scripts\activate
   - macOS/Linux: source .venv/bin/activate
5. Install requirements:

   bash

   ```
   pip install -r requirements.txt
   ```

Installing the DeepSeek Model

1. Ensure Ollama is installed and running
2. Open a terminal and run:

   bash

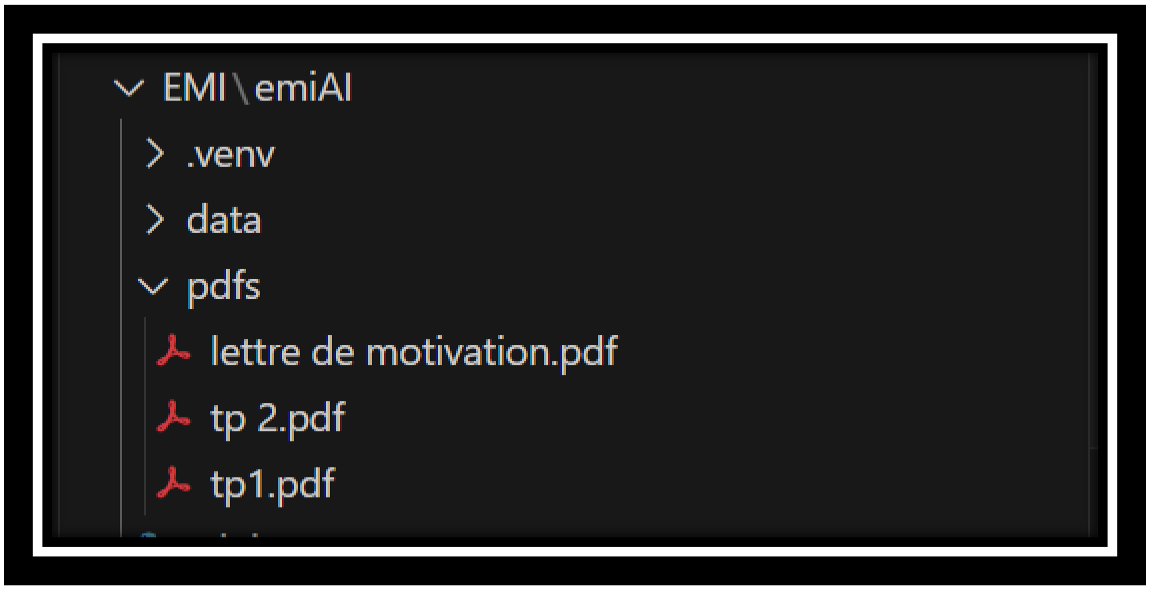
   ```
   ollama pull deepseek-r1:1.5b
   ```
3. Wait for the download to complete (this may take some time depending on your internet connection)

Step 2: Document Management

Adding Course Documents

1. Navigate to the project directory
2. Locate the pdfs folder:

   EMI/emiAI/pdfs/
3. Copy your PDF course materials into this folder



```
EMI\emiAI
  .venv
  data
  pdfs
    lettre de motivation.pdf
    tp 2.pdf
    tp1.pdf
```

exemple :

4. Guidelines for PDFs:
   - ❖ Use descriptive filenames
   - ❖ Ensure PDFs are searchable (not scanned images without OCR)
   - ❖ Recommended size: under 50MB per file for optimal performance

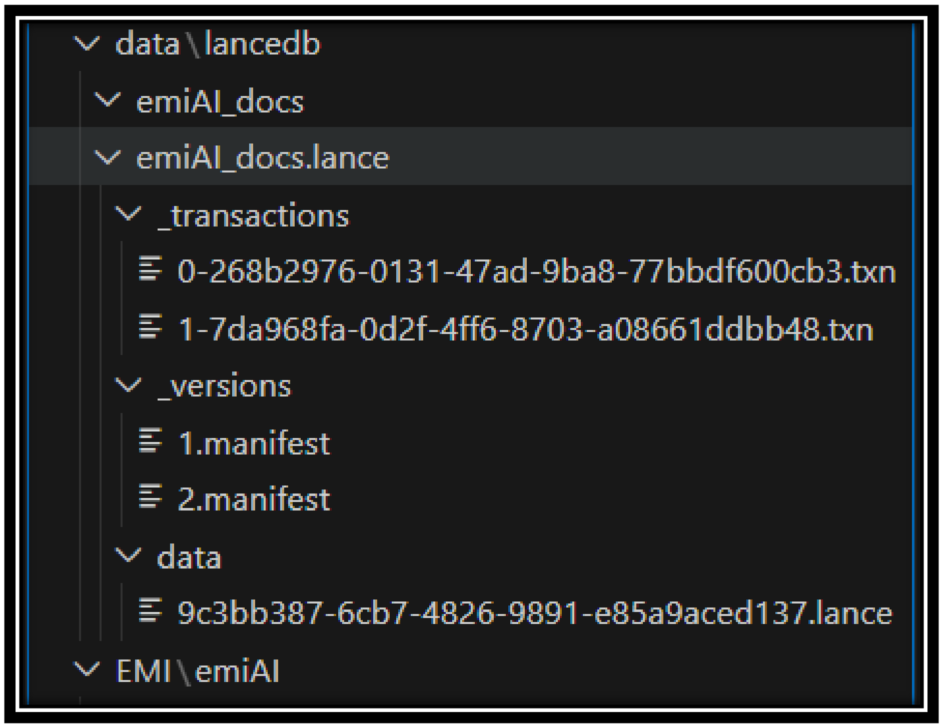
Processing Documents

After adding or updating PDFs, you must process them:

1. Ensure your virtual environment is activated
2. Navigate to the project directory
3. Run the preprocessing script:

   bash

   python 0-preprocess_pdfs.py
4. The script will:
   - ❖ Extract text from all PDFs in the pdfs folder
   - ❖ Chunk the content into manageable pieces
   - ❖ Generate embeddings for these chunks
   - ❖ Store everything in a LanceDB database



```
data\lancedb
  emiAI_docs
  emiAI_docs.lance
    _transactions
      0-268b2976-0131-47ad-9ba8-77bbdf600cb3.txn
      1-7da968fa-0d2f-4ff6-8703-a08661ddbb48.txn
    _versions
      1.manifest
      2.manifest
    data
      9c3bb387-6cb7-4826-9891-e85a9aced137.lance
EMI\emiAI
```

5. Watch the console output for:
    - Processing progress
    - Any errors or warnings
    - Confirmation of successful database update

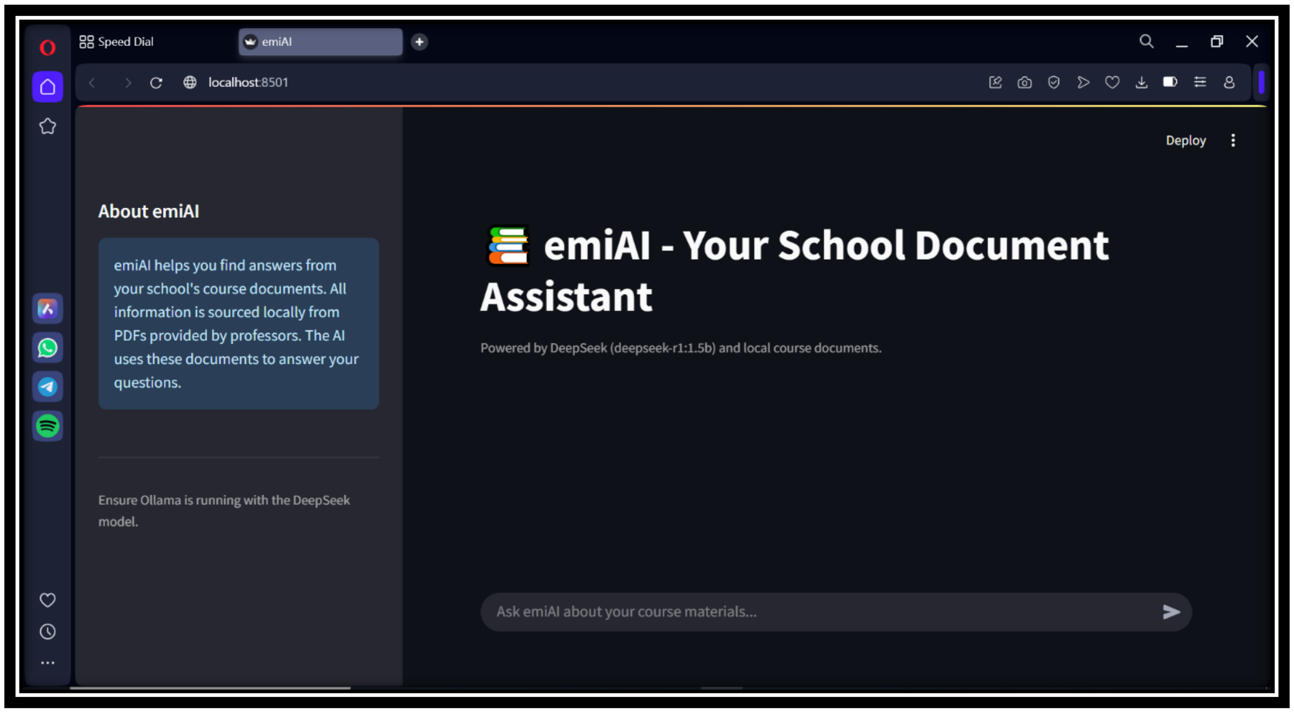
Step 3: Running the Application

Starting the Streamlit App

1. Ensure Ollama is running in the background
2. Navigate to the project directory
3. Activate your virtual environment if not already active
4. Start the Streamlit application:

   bash

   ```
   streamlit run app_emiAI.py
   ```
5. The application will:
    - Start a local web server
    - Open a browser window automatically
    - Display the emiAI interface



About emiAI

emiAI helps you find answers from your school's course documents. All information is sourced locally from PDFs provided by professors. The AI uses these documents to answer your questions.

Ensure Ollama is running with the DeepSeek model.

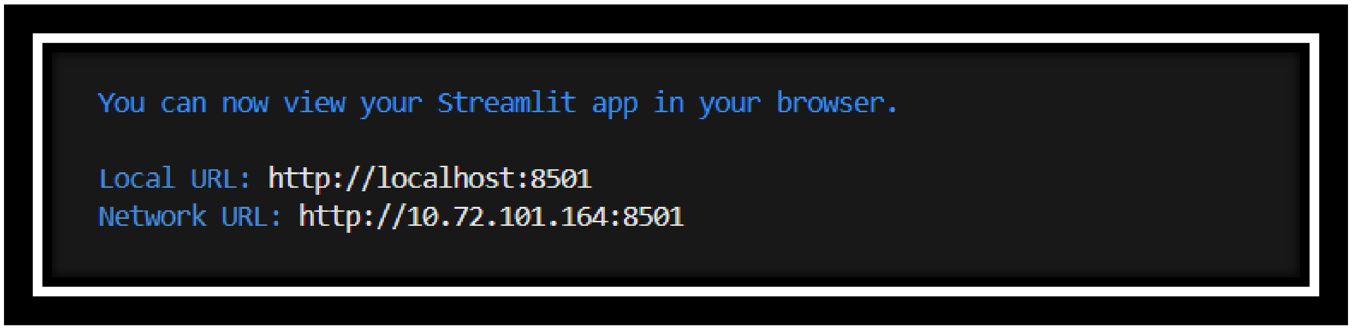
emiAI - Your School Document Assistant

Powered by DeepSeek (deepseek-r1:1.5b) and local course documents.

Ask emiAI about your course materials...

Accessing the Application

- Local access: http://localhost:8501
- Network access (same network): See the terminal output for the specific URL



```
You can now view your Streamlit app in your browser.

Local URL: http://localhost:8501
Network URL: http://10.72.101.164:8501
```

- To make the application available externally, consider using Streamlit sharing or a reverse proxy

Common Issues and Solutions

Problem	**Possible Cause**	**Solution**
Application won't start	Streamlit not installed	Check requirements installation
"Database not initialized" error	Missing or corrupted database	Run preprocessing script
No responses generated	Ollama not running	Start Ollama in background
Incorrect model name	Update OLLAMA_MODEL_NAME in app_emiAI.py	
Slow responses	Large document collection	Consider reducing number of documents
Poor answer quality	Documents not properly chunked	Adjust chunking parameters in preprocessing script
		Try reducing SIMILARITY_THRESHOLD

Step 5: Customization

Adjusting Parameters

You can customize the application by editing these parameters in app_emiAI.py:

- ❖ NUM_RESULTS_TO_RETRIEVE: Number of document chunks to retrieve (more chunks = more context but slower)
- ❖ SIMILARITY_THRESHOLD: Minimum similarity score for a chunk to be considered (lower = more results but less relevant)
- ❖ CONTEXT_WINDOW_SIZE: Size of the context window for the language model (adjust based on model capabilities)

Advanced Customization

For advanced users, consider modifying:

- 0-preprocess_pdfs.py:
  - ❖ CHUNK_MAX_TOKENS: Adjust chunk size for different document types
  - ❖ EMBEDDING_MODEL_NAME: Use a different embedding model for different languages or domains

Ecole Mohammadia d'Ingénieurs
المدرسة المحمدية للمهندسين

جامعة محمد الخامس بالرباط
Université Mohammed V de Rabat

Security Considerations

- **Data Privacy**: All data stays local to your machine
- **Access Control**: Consider implementing user authentication if deploying for wider access
- **Network Security**: By default, the app is accessible on your local network

Getting Help

If you encounter issues not covered in this guide:

1. Check the system logs for error messages
2. Refer to the documentation for individual components:
   - Streamlit Documentation
   - Ollama Documentation
   - LanceDB Documentation
3. Search online communities for similar issues