Handout: Building an LLM-Powered RAG Chatbot - Part 3

Project Structure & Code Walkthrough

Welcome to the hands-on part of our session! In this section, we'll dive into the actual code that powers our LLM-Powered RAG Chatbot. We'll explore the project's structure and examine key code snippets to understand how each component contributes to the overall system.

Our Goal: To understand the "how" behind building an LLM-powered chatbot, focusing on the important components, their roles, and how they interact.

1. Project Structure: The Blueprint of Our Application

A well-organized project is easier to understand, maintain, and extend. Our RAG chatbot follows a modular structure, separating concerns into logical files and folders.

```
agent-base/
├── config.yaml              # Centralized configuration for the entire application
├── prep-data.py             # Script to prepare and ingest data into the vector database
├── app.py                   # Streamlit web application for the RAG chatbot UI
└── src/                     # Source code for modular components (our "toolkit")
    ├── __init__.py          # Makes 'src' a Python package (empty file)
    ├── config_loader.py     # Handles loading and parsing of config.yaml
    ├── document_loader.py   # Manages loading documents from various sources (PDF, CSV, Web, Text)
    ├── text_splitter.py     # Encapsulates logic for splitting documents into chunks
    ├── embedding_model.py   # Initializes and provides the Ollama embedding model
    ├── llm_model.py         # Initializes and provides the Ollama LLM for generation
    ├── rag_chain.py         # Builds and orchestrates the LangChain RAG pipeline
    └── vector_store.py      # Manages ChromaDB connection and document operations
└── data/                    # Directory to store your raw source documents
    ├── pdfs/                # Example: Place your PDF files here
    ├── csvs/                # Example: Place your CSV files here
    └── texts/               # Example: Place your plain text files here
└── chroma_db/               # Directory where ChromaDB will persist its data (created by prep-data.py)
```

Why this structure?

- **Clarity:** Each file has a clear, single responsibility.
- **Maintainability:** Changes to one part (e.g., how PDFs are loaded) don't necessarily affect others.
- **Extensibility:** Easy to add new features (e.g., new data sources, different LLMs) by modifying specific modules.
- **Reusability:** Functions within src/ can be reused in other projects.

2. Code Walkthrough: Diving into the Details

Let's explore the core components and their corresponding code snippets.

A. Configuration: The Brain's Settings

Our chatbot needs to know which LLM to use, where to find data, and how to store information. We manage all these settings in one place: config.yaml.

Code File: config.yaml

```
# config.yaml
# ... (other sections)
ollama:
  host: "http://localhost:11434"
  llm_model: "llama3.2:latest"
  embedding_model: "mxbai-embed-large:latest"

data_ingestion:
  document_sources:
    - type: "pdf"
      path: "./data/pdfs/"
  chunking:
    chunk_size: 1000
    chunk_overlap: 200
  vector_store:
    type: "chromadb"
    persist_directory: "./chroma_db"
    collection_name: "rag_chatbot_collection"
# ... (other sections)
```

Concept: This YAML file centralizes all configurable parameters. It's human-readable and allows us to change settings (like swapping LLMs or adjusting chunk sizes) without touching the Python code.

- **Code File:** src/config_loader.py

```
# src/config_loader.py
import yaml
import os

def load_config(config_path: str = "config.yaml") -> dict:
    """Loads and parses the YAML configuration file."""
    if not os.path.exists(config_path):
```

```
        raise FileNotFoundError(f"Configuration file not found at: {config_path}")
    try:
        with open(config_path, 'r') as file:
            config = yaml.safe_load(file)
        return config
    except Exception as e:
        raise Exception(f"Error loading config: {e}")
```

Concept: This simple function reads the config.yaml file and converts its content into a Python dictionary, making all our settings accessible throughout the application.

- **Rationale:** Decouples configuration from code, making the application flexible and easy to update without redeploying code.
- **Other Options:** Could use .env files for environment variables (good for secrets) or Python dictionaries directly (less flexible for external changes).

B. Data Ingestion: Building Our Knowledge Base

This is the process of taking raw documents and transforming them into a searchable format for our chatbot. The prep-data.py script orchestrates this.

- **Code File:** prep-data.py

```
# prep-data.py (Simplified structure)
import os, shutil, sys
from src.config_loader import load_config
from src.document_loader import load_documents_from_sources
from src.text_splitter import get_text_splitter
from src.embedding_model import get_ollama_embeddings
from src.vector_store import get_chroma_vector_store, add_documents_to_vector_store

def prepare_data(clear_existing_db: bool = True):
    # 1. Load Configuration (from config.yaml)
    config = load_config()
    # ... extract relevant configs ...

    # 2. (Optional) Clear existing ChromaDB for a full refresh
    if clear_existing_db and
os.path.exists(config['data_ingestion']['vector_store']['persist_directory']):
        shutil.rmtree(config['data_ingestion']['vector_store']['persist_directory'])

    # 3. Load Documents (using src/document_loader.py)
    documents =
load_documents_from_sources(config['data_ingestion']['document_sources'])
```

```
    # 4. Split Documents into Chunks (using src/text_splitter.py)
    text_splitter = get_text_splitter(
        chunk_size=config['data_ingestion']['chunking']['chunk_size'],
        chunk_overlap=config['data_ingestion']['chunking']['chunk_overlap']
    )
    chunks = text_splitter.split_documents(documents)

    # 5. Initialize Embedding Model (using src/embedding_model.py)
    embeddings = get_ollama_embeddings(
        model_name=config['ollama']['embedding_model'],
        base_url=config['ollama']['host']
    )

    # 6. Initialize/Load ChromaDB and Add Documents (using src/vector_store.py)
    vector_db = get_chroma_vector_store(

persist_directory=config['data_ingestion']['vector_store']['persist_directory'],
        collection_name=config['data_ingestion']['vector_store']['collection_name'],
        embedding_function=embeddings
    )
    add_documents_to_vector_store(vector_db, chunks)

    print("--- Data Preparation Complete ---")

if __name__ == "__main__":
    prepare_data(clear_existing_db=True) # Runs a full refresh by default
```

Concept: prep-data.py acts as the main script for our "librarian." It reads the configuration, then systematically calls functions from our src/ toolkit to load, split, embed, and store our documents. The clear_existing_db=True ensures we always start with a fresh, up-to-date knowledge base.

- **Rationale:** This script isolates the data preparation process from the main application, allowing for independent updates of the knowledge base. The clear_existing_db flag provides a simple "full refresh" mechanism for the PoC.
- **Other Options:** For production, you might implement more sophisticated incremental updates (detecting changes, adding/updating/deleting specific documents) rather than a full wipe and reload.

- **Code File:** src/document_loader.py

```
# src/document_loader.py (Snippet)
from langchain_community.document_loaders import PyPDFLoader, CSVLoader,
```

```
WebBaseLoader, DirectoryLoader, TextLoader
from langchain_core.documents import Document

def load_documents_from_sources(sources_config: list) -> list[Document]:
    """Loads documents from various configured sources."""
    all_documents = []
    for source in sources_config:
        source_type = source.get('type')
        source_path = source.get('path')
        source_urls = source.get('urls')

        if source_type == "pdf" and source_path:
            # Logic to load PDF files/directories
            loader = PyPDFLoader(source_path) if os.path.isfile(source_path) else
DirectoryLoader(source_path, glob="*.pdf", loader_cls=PyPDFLoader)
            all_documents.extend(loader.load())
        elif source_type == "csv" and source_path:
            # Logic to load CSV files
            loader = CSVLoader(file_path=source_path)
            all_documents.extend(loader.load())
        elif source_type == "website" and source_urls:
            # Logic to load web pages
            loader = WebBaseLoader(web_paths=source_urls)
            all_documents.extend(loader.load())
        # ... and so on for other types
    return all_documents
```

Concept: This module is responsible for reading data from different formats (PDFs, CSVs, websites, plain text). It uses LangChain's specialized loaders to convert raw data into Document objects, which contain the text content and useful metadata (like source file path or URL).

- **Rationale:** Abstracts away the complexity of parsing different file formats. Each loader knows how to extract text from its specific type.
- **Other Options:** LangChain offers hundreds of loaders for various data sources (Notion, Slack, databases, Google Drive, etc.). You could also write custom loaders for highly specific or proprietary formats.

- **Code File:** src/text_splitter.py

```
# src/text_splitter.py (Snippet)
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

```
from langchain_core.documents import Document

def get_text_splitter(chunk_size: int, chunk_overlap: int) ->
RecursiveCharacterTextSplitter:
    """Returns a configured RecursiveCharacterTextSplitter."""
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
    )
    return text_splitter
```

Concept: Large documents are problematic for LLMs (context window limits). This module breaks documents into smaller, manageable chunks. chunk_size defines the maximum size of each piece, and chunk_overlap ensures some context is carried over between adjacent chunks, preventing loss of meaning.

- **Rationale:** Optimizes the data for LLM consumption. Smaller chunks lead to more precise retrieval, while overlap helps maintain context across chunk boundaries.
- **Other Options:** LangChain has other text splitters (e.g., CharacterTextSplitter, MarkdownTextSplitter, HTMLHeaderTextSplitter) for different document structures. Advanced strategies involve "Parent Document Retriever" for retrieving larger contexts based on smaller chunks.

- **Code File:** src/embedding_model.py

```
# src/embedding_model.py (Snippet)
from langchain_community.embeddings import OllamaEmbeddings
from typing import Union

def get_ollama_embeddings(model_name: str, base_url: str = "http://localhost:11434") -
> OllamaEmbeddings:
    """Initializes and returns an OllamaEmbeddings instance."""
    try:
        embeddings = OllamaEmbeddings(model=model_name, base_url=base_url)
        # A small test to ensure connectivity
        _ = embeddings.embed_query("test") # This line attempts to get an embedding
for a dummy text
        return embeddings
    except Exception as e:
        print(f"Error initializing OllamaEmbeddings: {e}")
        print("Please ensure Ollama is running and the embedding model is pulled.")
        raise
```

Concept: Embeddings are numerical representations (vectors) of text that capture its semantic meaning. Texts with similar meanings will have embeddings that are "close" to each other in a high-dimensional space. This module uses Ollama to generate these embeddings locally, which is crucial for semantic search.

- **Rationale:** Transforms human-readable text into a machine-understandable format for vector database operations. Using Ollama allows local, private embedding generation.
- **Other Options:** You could use cloud-based embedding models (e.g., OpenAI, Cohere, Google's embeddings) or other open-source models (e.g., from Hugging Face) if you're not restricted to local execution.

- **Code File:** src/vector_store.py

```
# src/vector_store.py (Snippet)
import os
from langchain_community.vectorstores import Chroma
from langchain_core.documents import Document
from typing import Any, List

def get_chroma_vector_store(
    persist_directory: str,
    collection_name: str,
    embedding_function: Any
) -> Chroma:
    """Initializes or loads a ChromaDB vector store."""
    os.makedirs(persist_directory, exist_ok=True)
    try:
        vector_store = Chroma(
            persist_directory=persist_directory,
            embedding_function=embedding_function,
            collection_name=collection_name
        )
        return vector_store
    except Exception as e:
        print(f"Error initializing ChromaDB: {e}")
        raise

def add_documents_to_vector_store(vector_store: Chroma, documents: List[Document]):
    """Adds a list of documents to the given ChromaDB vector store."""
    if not documents: return
    try:
        vector_store.add_documents(documents)
        vector_store.persist() # Save changes to disk
```

```
    except Exception as e:
        print(f"Error adding documents to vector store: {e}")
        raise
```

Concept: ChromaDB is our "smart index" or "library catalog." It stores the embeddings and their corresponding text chunks. When we query it, it uses the embedding of our question to find the most semantically similar chunks, allowing for meaning-based retrieval rather than just keyword matching. persist_directory ensures our data is saved on disk and not lost when the application closes.

- **Rationale:** Provides efficient storage and retrieval of vector embeddings. persist_directory is key for local persistence, meaning you don't have to re-embed all data every time you run the app.
- **Other Options:** For larger-scale or multi-user applications, you might use cloud-based vector databases like Pinecone, Milvus, Weaviate, or Qdrant.

C. Chatbot Interaction: Bringing It to Life

This part focuses on the user interface and how the chatbot processes live queries.

- **Code File:** app.py

```
# app.py (Simplified structure)
import streamlit as st
import os
from src.config_loader import load_config
from src.llm_model import get_ollama_llm
from src.embedding_model import get_ollama_embeddings
from src.vector_store import get_chroma_vector_store
from src.rag_chain import build_rag_chain

# --- Load Configuration (cached to run once) ---
@st.cache_resource
def load_application_config():
    return load_config()
config = load_application_config()

# --- Initialize RAG System (LLM, Embeddings, Vector Store, RAG Chain) ---
@st.cache_resource
def setup_rag_system(ollama_cfg, rag_cfg, vector_store_cfg):
    # 1. Initialize LLM (using src/llm_model.py)
    llm = get_ollama_llm(model_name=ollama_cfg['llm_model'],
base_url=ollama_cfg['host'])
    # 2. Initialize Embedding Model (for ChromaDB connection)
```

```
    embeddings = get_ollama_embeddings(model_name=ollama_cfg['embedding_model'],
base_url=ollama_cfg['host'])
    # 3. Connect to ChromaDB
    vector_db = get_chroma_vector_store(
        persist_directory=vector_store_cfg['persist_directory'],
        collection_name=vector_store_cfg['collection_name'],
        embedding_function=embeddings
    )
    retriever = vector_db.as_retriever(search_kwargs={"k": rag_cfg['retrieval_k']})
    # 4. Build RAG chain (using src/rag_chain.py)
    rag_chain = build_rag_chain(llm=llm, retriever=retriever,
chain_type=rag_cfg['chain_type'], return_source_documents=True)
    return rag_chain

# Setup the RAG system (runs once and is cached by Streamlit)
rag_chain = setup_rag_system(config['ollama'], config['rag'],
config['data_ingestion']['vector_store'])

# --- Streamlit UI Setup & Chat Logic ---
st.set_page_config(page_title=config['app_name'], layout="wide")
st.title(config['app_name'])

# Initialize chat history in Streamlit's session state
if "messages" not in st.session_state:
    st.session_state.messages = []

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

# Handle new user input
if prompt := st.chat_input("Ask me about the loaded data..."):
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    with st.chat_message("assistant"):
        message_placeholder = st.empty()
        full_response_content = ""

        # Invoke the RAG chain
        result = rag_chain.invoke({"query": prompt})
        response_content = result.get("result", "Sorry, I couldn't find an answer.")
        source_documents = result.get("source_documents", [])

        # Display response (simulated streaming)
```

```
        for word in response_content.split():
            full_response_content += word + " "
            message_placeholder.markdown(full_response_content + "▌")

        # Add sources to response
        sources_info = ""
        if source_documents:
            sources_info = "\n\n**Sources:**\n"
            for i, doc in enumerate(source_documents):
                source_name = doc.metadata.get('source', f"Document {i+1}")
                page_number = doc.metadata.get('page', None)
                if page_number is not None:
                    source_name += f" (Page: {page_number})"
                sources_info += f"- {source_name}\n"

        final_response = full_response_content + sources_info
        message_placeholder.markdown(final_response)
        st.session_state.messages.append({"role": "assistant", "content":
final_response})
```

Concept: This is the main application file, creating our interactive chatbot.

- **@st.cache_resource:** This decorator is crucial! It tells Streamlit to run the load_application_config and setup_rag_system functions only *once* when the app starts, and then reuse their results across all user interactions. This prevents slow re-initialization of LLMs and databases on every chat message.
  - **Rationale:** Streamlit reruns the entire script from top to bottom on every user interaction (e.g., typing a message). Without caching, the LLM and vector database would be re-initialized every time, making the app extremely slow.
  - **Other Options:** For more complex state management beyond simple caching, you might consider st.session_state for smaller, dynamic variables, or external state management libraries for very large applications.
- **st.session_state.messages:** Streamlit reruns the entire script on every interaction. session_state allows us to "remember" the conversation history so it persists and is displayed correctly.
  - **Rationale:** Essential for maintaining conversational context and displaying a continuous chat history in a stateless web framework like Streamlit.
  - **Other Options:** For persistent chat history across browser sessions or users, you would need to store messages in a database (e.g., Firestore, SQL DB).

- **rag_chain.invoke({"query": prompt}):** This is where the magic happens! The user's question triggers the entire RAG process (retrieval, augmentation, generation) through our rag_chain.
    - **Rationale:** This single call orchestrates the complex RAG workflow defined in src/rag_chain.py, abstracting away the underlying steps.
    - **Other Options:** For more fine-grained control or custom logic, you could manually call the retriever and then the LLM, but RetrievalQA simplifies common RAG patterns.

- **Code File:** src/llm_model.py

```
# src/llm_model.py (Snippet)
from langchain_community.llms import Ollama
from langchain_community.chat_models import ChatOllama
from langchain_core.language_models import BaseChatModel, BaseLLM
from typing import Union

def get_ollama_llm(model_name: str, base_url: str = "http://localhost:11434") ->
Union[BaseLLM, BaseChatModel]:
    """Initializes and returns an Ollama LLM or ChatOllama instance."""
    try:
        llm = ChatOllama(model=model_name, base_url=base_url)
        return llm
    except Exception as e:
        # Fallback or error handling if ChatOllama fails
        llm = Ollama(model=model_name, base_url=base_url) # Try base Ollama LLM
        return llm
```

Concept: This module connects to our locally running LLM via Ollama. It acts as the "brain" of our chatbot, capable of understanding the user's query and generating coherent, human-like responses once provided with context.

Rationale: Provides the generative capability. Using Ollama enables running powerful LLMs locally, offering privacy and cost control. ChatOllama is often preferred for conversational agents.

Other Options: Can be swapped with other LangChain-supported LLM integrations (e.g., ChatOpenAI, ChatGoogleGenerativeAI) if using cloud APIs.

- **Code File:** src/rag_chain.py

```
# src/rag_chain.py (Snippet with Custom Prompt)
from langchain.chains import RetrievalQA
from langchain_core.language_models import BaseChatModel, BaseLLM
from langchain_core.retrievers import BaseRetriever
from langchain_core.prompts import PromptTemplate
from typing import Any, Union

def build_rag_chain(
    llm: Union[BaseLLM, BaseChatModel],
    retriever: BaseRetriever,
    chain_type: str = "stuff",
    return_source_documents: bool = True
) -> Any:
    """Builds and returns a LangChain RetrievalQA chain."""

    # --- CUSTOMIZATION POINT: The Prompt Template ---
    custom_prompt_template = """Based on the context provided, answer the question
clearly and concisely.
    Present your answer using bullet points if multiple distinct facts are available,
or a single paragraph otherwise.
    If the answer is not found in the context, politely state that the information is
not available in the provided documents.

    Context:
    {context}

    Question: {question}

    Answer:"""
    QA_CHAIN_PROMPT = PromptTemplate.from_template(custom_prompt_template)

    try:
        qa_chain = RetrievalQA.from_chain_type(
            llm=llm,
            chain_type=chain_type,
            retriever=retriever,
            return_source_documents=return_source_documents,
            chain_type_kwargs={"prompt": QA_CHAIN_PROMPT} # Pass our custom prompt
        )
        return qa_chain
    except Exception as e:
        print(f"Error building RAG chain: {e}")
        raise
```

Concept: This module is the heart of our RAG system. It combines the llm (our language model) and the retriever (which fetches relevant documents). The custom_prompt_template is extremely important here: it's our direct instruction to the LLM on *how* to answer the question, *what tone* to use, and *how to format* its response (e.g., using bullet points, as we customized). Notice how {context} and {question} are placeholders that LangChain fills in dynamically. return_source_documents=True ensures we get back the documents that informed the answer.

- **Rationale:** Orchestrates the retrieval and generation steps. The prompt template is the primary control for the LLM's behavior and output format.
- **Other Options:** LangChain offers various chain types (map_reduce, refine, map_rerank) for handling different document volumes. You could also build custom chains using LangChain Expression Language (LCEL) for more complex workflows.

Conclusion

By breaking down the project into these modular components, we can see how each piece plays a vital role in the overall RAG chatbot system. From configuring our settings to ingesting data and finally interacting with the chatbot, every part is designed for clarity, efficiency, and future extensibility. This foundation allows us to build powerful, grounded AI applications.