RAG Chatbot Pipeline: Technical Code Walkthrough

1. Data Preprocessing

> **Purpose:** Prepare raw Kisan Call Centre Q&A data for high-quality semantic retrieval by cleaning, filtering, and deduplicating entries.

a. Loading Data

```
data = pd.read_csv(r"C:\Users\amank\Downloads\Agri-chatbot-versions\data\data\questionsv4.csv
data.head()
```

- **Functionality:** Loads the CSV containing all Q&A pairs into a DataFrame for processing.

b. Handling Missing and Non-informative Values

```
data.isna().sum()
hyphen_rows = data[data['answers'].astype(str).str.strip('-') == ""]
data = data[~data['answers'].astype(str).str.strip('-').eq("")]
call_transferred_rows = data[data['questions'].str.contains("test call", case=False, na=False
data = data[~data['questions'].str.contains("test call", case=False, na=False)]
call_transferred_rows = data[data['answers'].str.contains("transfered", case=False, na=False)
data = data[~data['answers'].str.contains("transfered", case=False, na=False)]
```

- Removes rows with missing or placeholder answers (e.g., `--`).
- Filters out administrative or test entries (e.g., `test call`, `transfered`).

> **Advantage:** Ensures only relevant, informative Q&A pairs remain.

c. Filtering Numeric-only Answers

```
def contains_alphabet(value):
    return bool(re.search(r'[a-zA-Z]', str(value)))

df_without_letters = data[~data['answers'].astype(str).apply(contains_alphabet)]
df_with_letters = data[data['answers'].astype(str).apply(contains_alphabet)]
```

- Separates out answers that contain no alphabetic characters (likely non-informative).

> **Advantage:** Further improves dataset quality by removing irrelevant entries.

d. Semantic Deduplication

```
df['qa_combined'] = df['questions'].str.lower().str.strip() + ' ' + df['answers'].str.lower(
device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = SentenceTransformer('all-MiniLM-L6-v2').to(device)
embeddings = model.encode(df['qa_combined'], convert_to_tensor=True, device=device, show_prog
cosine_scores = util.pytorch_cos_sim(embeddings, embeddings)
to_drop = set()
for i in range(len(df)):
    if i in to_drop:
        continue
    for j in range(i+1, len(df)):
        if cosine_scores[i][j] > 0.95:
            to_drop.add(j)
df_dedup = df.drop(index=list(to_drop)).reset_index(drop=True)
```

- Concatenates Q&A for semantic comparison.
- Computes embeddings using a transformer model.
- Removes semantically duplicate entries (cosine similarity > 0.95).

> **Advantage over traditional deduplication:** Detects near-duplicates even if phrased differently, resulting in a more diverse and relevant dataset.

2. Chunking and Vector Embedding

> **Purpose:** Transform each Q&A pair into a vector embedding, ready for semantic search.

a. Row-based Chunking

- **Implementation:** Each DataFrame row (a Q&A pair) is treated as a single document.

> **Advantage:** Maintains semantic coherence and aligns with retrieval needs.

b. Embedding with Ollama

```
class OllamaEmbeddings:
    def __init__(self, client):
        self.client = client
    def embed_documents(self, texts):
        return [self.ollama_embed(text) for text in texts]
    def embed_query(self, text):
        return self.ollama_embed(text)
    def ollama_embed(self, text: str):
        response = self.client.embeddings.create(
            model='nomic-embed-text:latest',
            input=text
        )
        return response.data[0].embedding
```

- Wraps embedding generation using a local Ollama server.
- Ensures all embeddings (documents and queries) are consistent.

> **Advantage:** Local, private, and fast; easily swappable for future models.

3. Vector Database Storage (ChromaDB)

> **Purpose:** Persistently store document embeddings for fast, scalable semantic search.

a. Document Preparation and Storage

```
class ChromaDBBuilder:
    def load_csv_to_documents(self):
        df = pd.read_csv(self.csv_path)
        self.documents = [
            Document(
                page_content=f"Question: {row['questions']}\nAnswer: {row['answers']}",
                metadata={"row": i}
            )
            for i, row in df.iterrows()
        ]
    def store_documents_to_chroma(self):
        db = Chroma.from_documents(
            documents=self.documents,
            embedding=self.embedding_function,
            persist_directory=self.persist_dir
        )
```

- Loads Q&A pairs as Document objects.
- Embeds and stores them in ChromaDB with metadata.

> **Advantage:** Ensures persistence (no need to re-embed after restart). Fast similarity search and easy integration with LangChain.

4. Semantic Retrieval & Reranking

> **Purpose:** Retrieve the most relevant Q&A pairs for a user's query using semantic similarity.

a. Query Embedding and Similarity Search

```
class ChromaQuervHandler:
    def get answer(self, question: str) -> str:
        raw results = self.db.similarity search with score(question, k=10)
        relevant_docs = self.rerank_documents(question, raw_results)
```

- Embeds the user's query.
- Retrieves top-k (e.g., 10) most similar documents from ChromaDB.

b. Cosine Reranking

```
def rerank documents(self, question, results, top k=5):
    query embedding = self.embedding_function.embed_query(question)
    reranked = sorted(
        results,
        key=lambda x: self.cosine sim(query embedding,
                                      self.embedding_function.embed_query(x[0].page_content))
        reverse=True
    )
    return [doc.page_content for doc, _ in reranked[:top_k]]
```

- Further reranks the retrieved documents using cosine similarity.
- Selects the top 5 for prompt context.

> **Advantage:** Balances recall (broad search) and precision (fine reranking). Filters out less relevant results, improving LLM response quality.

5. Prompt Construction & LLM Response

> **Purpose:** Combine retrieved context and user query into a prompt for the LLM, then

> generate a factual, context-grounded answer.

a. Prompt Construction

```
def construct_prompt(self, context, question):
    return self.PROMPT_TEMPLATE.format(context=context, question=question)
```

- Inserts the top 5 Q&A contexts and the user's question into a structured prompt.
- Instructs the LLM to only use provided context, avoid hallucinations, and provide fallback if insufficient information.

b. LLM Inference

```
response = self.client.chat.completions.create(
    model=self.model_name,
    messages=messages,
    temperature=0.3
)
```

- Sends the prompt to the locally hosted LLM (e.g., gemma3:27b via Ollama).
- Receives and returns the synthesized answer.

6. FastAPI Web Application

> **Purpose:** Provide a user-friendly web interface for interacting with the chatbot.

a. FastAPI Setup

```
app = FastAPI()
app.mount("/static", StaticFiles(directory="static"), name="static")
templates = Jinja2Templates(directory="templates")
```

- Sets up static file serving and Jinja2 templating for HTML rendering.

b. Query Handling Endpoint

```
@app.post("/query", response_class=HTMLResponse)
async def query(request: Request, question: str = Form(...)):
    raw_answer = query_handler.get_answer(question)
    html_answer = markdown.markdown(answer_only, extensions=["extra", "nl2br"])
    return templates.TemplateResponse("index.html", {
        "request": request,
```

```
    "result": html answer,
    "question": question
})
```

- Receives user queries from the web form.
- Calls the retrieval and LLM pipeline.
- Renders the answer (converted from markdown to HTML) on the webpage.

c. Frontend (HTML/CSS)

- **index.html:** Provides a clean, responsive user interface for question submission and answer display.
- **style.css:** Ensures a professional, readable, and visually appealing UI.

Summary: Advantages & Innovations

- **Semantic Deduplication:** Removes redundant knowledge, improving retrieval diversity.
- **Row-based Chunking:** Ensures each Q&A pair is contextually coherent and maximally useful.
- **Local Embedding & LLM:** Privacy, speed, and flexibility; no reliance on third-party APIs.
- **ChromaDB Integration:** Fast, persistent, and scalable vector storage.
- **Cosine Reranking:** Increases answer relevance and precision.
- **Structured Prompting:** Reduces hallucinations, ensures factual, context-based answers.
- **Modern Web UI:** FastAPI + Jinja2 + Markdown for a seamless user experience.