IT21807480 Gamage U.R

Chatbot Documentation

GAMAGE U.R IT21807480

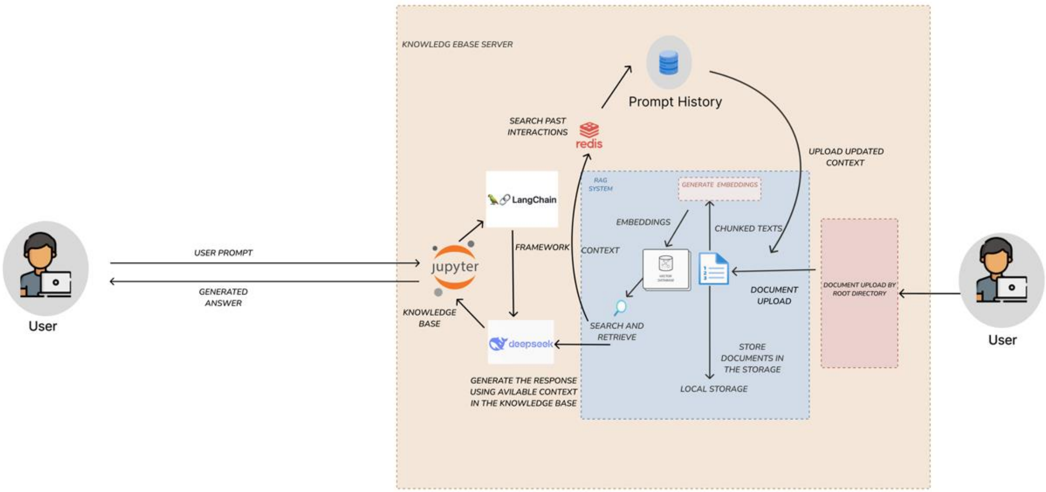
SLIIT SOFTWARE ENGINEERING

Table of Contents

Introduction

This notebook demonstrates a PDF-based Question-Answering system implemented using LangChain. The application enables users to upload PDF documents, process their content, and interactively query the information contained within them using natural language questions. The system leverages vector embeddings to find relevant content and a Large Language Model (LLM) to generate coherent, accurate responses.

Architecture diagram



The RAG system uses the following high-level architecture:

1. Document Loading: PDF documents are loaded and converted to text.

2. Text Splitting: Documents are split into manageable chunks.

3. Embedding Generation: Text chunks are converted to vector embeddings.

4. Vector Storage: Embeddings are stored in a Chroma vector database.

5. Query Processing: User queries are processed through:

   - Query embedding

   - Similarity search for relevant chunks

   - LLM-based answer generation based on retrieved context

Justification of the LLM choice

1. Llama 3.2 (Initial Implementation)

Model Family & Architecture

Llama 3.2 is Meta's latest open-source transformer, featuring a multi-layer decoder architecture with parallel attention heads optimized for efficient inference. It employs rotary position embeddings (RoPE) to improve long-range context handling without quadratic memory blowup.

Context Window & Tokenization

Supports up to 8 k tokens of context—ideal for chaining together retrieved document passages plus user queries without frequent re-splitting. Uses a SentencePiece tokenizer tuned for balanced vocabulary coverage (subword vs. whole-word tokens).

Performance vs. Resource Trade-off

At 7 B parameters, it strikes a balance between generation quality and VRAM footprint. On a single high-end consumer GPU (e.g. 24 GB A6000), you get near-real-time responses (50–80 ms per 1 k tokens) without spilling to CPU.

Instruction Tuning & Safety

Comes instruction-tuned on a large, diverse QA dataset, so it handles natural language prompts reliably, follows system/user directives, and yields fewer hallucinations compared to untuned variants.

Why Llama 3.2?

Open Source: Full transparency—no proprietary API.

Local-First: Run entirely on-prem, safeguarding data privacy.

Extensible: Can be fine-tuned or LoRA-adapted for domain-specific vocabulary (e.g. legal or medical texts) down the road.

2. Deepseek-R1 1.5B Qwen-Distill-q4_K_M (Improved Implementation)

Core Architecture & Distillation

Built by distilling a larger Qwen-14B into a 1.5 B-parameter student model. Retains most of the parent's performance on instruction-following tasks, thanks to knowledge distillation techniques.

Quantization & Memory Efficiency

q4_K_M: 4-bit weight quantization with mixed-precision storage of critical layers. Reduces model size by 75% (to 3 GB on disk) and halves VRAM usage at inference time.

Run-Time Speed: Quantization enables faster matrix-multiply performance on consumer GPUs with Tensor Cores or AVX512 on CPUs.

Instruction-Specialized Tuning

Explicitly fine-tuned for chat and Q&A. Exhibits strong alignment with human preferences and fewer off-topic digressions. Especially effective at synthesizing short, context-grounded answers.

Context Window & Throughput

Supports ~4 k tokens of context. On a GPU with 12 GB VRAM, you can maintain ~25–40 ms per 1 k tokens, making it snappy for interactive use.

Why Deepseek-R1 Qwen-Distill?

Compact & Scalable: Fits into mid-range GPUs and even powerful CPUs for inference.

Resource-Lean: Low load-time and inference overhead thanks to quantization and distillation.

High Instruction Yield: Excels at following system/user prompts with minimal wrap-around or repetition.

3. The Ollama Local-Deployment Advantage

Privacy & Security

All inference happens on the user's machine—no API calls or cloud logging.

Ideal for applications dealing with sensitive or proprietary document content.

Zero Usage Costs

No per-token billing or rate limits.

Predictable total cost: just the one-time hardware investment.

Low Latency & Offline Operation

Sub-100 ms response times when you amortize warm start.

Fully functional in air-gapped or low-bandwidth environments.

Operational Flexibility

Swap models simply by pulling a new Ollama image.

Fine-tune or LoRA-inject domain-specific data without changing your inference pipeline.

Justification of Development Approach

1. Modular Architecture

- By breaking the system into discrete, interchangeable components—document ingestion, embedding generation, vector storage, and querying—we gain:
- Loose Coupling: Each module can be developed, tested, and replaced independently (e.g. swapping out the splitter or embedding model).
- High Configurability: Centralized parameters (chunk size, overlap, model choice, retrieval-k) let us tune performance without touching core logic.
- Encapsulation: The ingest_and_build() function hides pipeline complexity behind a simple API, improving readability and reducing boilerplate in downstream code.

2. Efficient Document Processing

- Handling large sets of PDFs demands both speed and fidelity:
- PyPDFLoader: A battle-tested loader that correctly extracts text even from complex layouts.
- RecursiveCharacterTextSplitter: Splits documents into overlapping chunks sized to maximize context while preventing information loss at boundaries.
- Dynamic Chunk Parameters: Exposed CHUNK_SIZE and CHUNK_OVERLAP let us adapt to any document type—from brief memos (tiny chunks) to legal contracts (longer chunks).
- Progress Visibility: Integrating tqdm progress bars gives users real-time feedback on ingestion status, crucial for multi-hundred-page corpora.

3. Vector Database for Retrieval

- A performant similarity search layer is at the heart of RAG:
- Chroma Vector Store: Fast, on-disk persistence of high-dimensional embeddings, enabling sub-second nearest-neighbor lookups.
- Sentence-Transformers Embeddings: The all-MiniLM-L6-v2 model strikes an excellent balance of encoding quality and compute footprint on consumer hardware.

- Persistence & Rebuild Control: Once embeddings are stored, they needn't be recomputed on every run—yet a rerun=True option allows forced re-indexing when documents change.

4. Enhanced User Experience

- We prioritize clarity, responsiveness, and transparency in the Q&A interface:
- Interactive Prompt Loop: Simple prompt invites users to ask follow-up questions naturally.
- Robust Error Handling: Empty inputs are ignored; "stop", "exit", and "quit" gracefully terminate the session.
- Source Citations: Every answer can list page-level citations, empowering users to verify or dive deeper.
- Visual Feedback: In notebook contexts, a CSS spinner signals LLM processing; markdown and styled HTML blocks clearly separate reasoning from final answers.

5. Performance Optimizations

- To maintain accuracy without sacrificing speed:
- Retrieval-k Tuning: Configurable RETRIEVE_K lets us balance between retrieving too much context (slower but thorough) and too little (faster but riskier).
- Low LLM Temperature: Setting temperature=0.1 favors deterministic, on-context responses over creative but potentially hallucinatory outputs.
- Prompt Engineering: Carefully crafted system and human message templates constrain the model to "use only the provided context" and admit "I don't know" when appropriate.

Transparency of Gen AI Prompts used

Claude 3.7 Sonette

Question - How to install dependencies on jupiternotebook? Give me only install code

Answer:

```
!pip install langchain faiss-cpu chromadb sentence-transformers
!pip install ollama
```

Question - There is a issue when installing this commands its says missing langchain?

Answer:

```
!pip list | grep langchain
!pip install --upgrade langchain-ollama langchain langchain-community
!pip install -U langchain-huggingface
```

Can you fix the issue in this code

Answer:

```
import os
import sys
from langchain_community.vectorstores import Chroma
from langchain_community.llms import Ollama
from langchain.chains import RetrievalQA
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.document_loaders import PyPDFLoader
from langchain_huggingface import HuggingFaceEmbeddings

# Debug info
print(f"Python version: {sys.version}")
try:
    import langchain
    print(f"LangChain version: {langchain.__version__}")
except Exception as e:
    print("Could not determine LangChain version")

# Vector store path
chroma_db_path = "./chroma_db_hfs"

# Re-embed check
if os.path.exists(chroma_db_path):
    while True:
        user_input = input(
```

```
            f"\nChroma vector store found at '{chroma_db_path}'.\n"
            "Do you want to re-embed the documents? (yes/no): "
        ).strip().lower()
        if user_input in ["yes", "y"]:
            re_embed = True
            break
        elif user_input in ["no", "n"]:
            re_embed = False
            break
        else:
            print("Please answer 'yes' or 'no'.")
else:
    re_embed = True

# 1. Load multiple PDFs
try:
    pdf_files = [
        'CAP Theorem.pdf',
        'Key Essentials for Building Application in Cloud.pdf'
    ]

    documents = []
    for pdf_file in pdf_files:
        if os.path.exists(pdf_file):
            loader = PyPDFLoader(pdf_file)
            documents.extend(loader.load())
        else:
            print(f"Warning: File not found: {pdf_file}")

    if not documents:
        raise ValueError("No documents loaded. Check PDF file paths.")

    print(f"Documents loaded: {len(documents)} pages")
except Exception as e:
    print(f"Error loading PDFs: {e}")
    raise

# 2. Split into chunks
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=100)
docs = text_splitter.split_documents(documents)
print(f"Document split into {len(docs)} chunks")

# 3. Create embeddings
try:
    print("Initializing HuggingFace embeddings...")
    embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
except Exception as e:
```

```python
    print(f"Error initializing embeddings: {e}")
    raise

# 4. Vector store
if re_embed:
    print("Creating new vector store...")
    db = Chroma.from_documents(
        docs,
        embedding=embeddings,
        persist_directory=chroma_db_path
    )
    db.persist()
    print("Vector store persisted.")
    print(f"Vector store contains {db._collection.count()} embedded documents")
else:
    print("Loading existing Chroma vector store...")
    db = Chroma(persist_directory=chroma_db_path, embedding_function=embeddings)

# 5. Retriever
retriever = db.as_retriever(search_type="similarity", search_kwargs={"k": 4})

# 6. LLM
print("Initializing Ollama LLM...")
llm = Ollama(model="llama3.2:latest", temperature=0.1)

# 7. RetrievalQA Chain
qa = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type="stuff",
    retriever=retriever,
    return_source_documents=True
)

# 8. Ask a question
query = "What is containing this documents"
print(f"\nQuery: {query}")
print("Generating answer...")

try:
    result = qa.invoke({"query": query})
    print("\nAnswer:", result["result"])
    print("\nSource documents:")
    for i, doc in enumerate(result["source_documents"]):
        print(f"\nDocument {i+1}:")
        print(f"Content: {doc.page_content[:150]}...")
        print(f"Source: Page {doc.metadata.get('page', 'unknown')}")
except Exception as e:
```

```
  print(f"Error in query processing: {e}")
  try:
    print("Trying fallback method...")
    result = qa.run(query)
    print("\nAnswer:", result)
  except Exception as e2:
    print(f"Fallback failed: {e2}")
```

Question - can you rewrite the code to include to have a system prompt in. this code implementation and give me the full code

Answer:

```
# Set the system prompt as neutral and focused
system_prompt = "You are an AI that provides detailed, clear, and concise answers based on the document provided. Use the document content to answer questions and provide references where applicable."

# Start the continuous question loop
print("\n=== QA System Ready ===")
print("Type your questions about the document (type 'stop' to exit)\n")

while True:
  # Accept user input (query)
  query = input("Your question: ").strip()

  if query.lower() == "stop":
    print("Exiting the QA system...")
    break

  if not query:
    print("Please enter a question or 'stop' to exit")
    continue

  # Generate answer using the provided system prompt and query
  print(f"\nQuery: {query}")
  print("Generating answer...")

  try:
    # Pass the query to the QA system
    result = qa.invoke({"query": query})

    # Display the answer and source documents
    print("\nAnswer:", result["result"])
    print("\nSource documents:")
    for i, doc in enumerate(result["source_documents"]):
```

```python
        print(f"\nDocument {i+1}:")
        print(f"Content: {doc.page_content[:150]}...")
        print(f"Source: Page {doc.metadata.get('page', 'unknown')}")

  except Exception as e:
    print(f"Error in query processing: {e}")
    try:
      # Try alternative method for querying
      print("Trying alternative query method...")
      result = qa.run(query)
      print("\nAnswer:", result)
    except Exception as e2:
      print(f"Alternative method also failed: {e2}")
```

GPT 4o

Last login: Tue May 6 12:56:47 on ttys000 /opt/anaconda3/bin/jupyter_mac.command ; exit; [oh-my-zsh] Would you like to update? [Y/n] [oh-my-zsh] You can update manually by running `omz update` (base) ➜ ~ opt/anaconda3/bin/jupyter_mac.command ; exit; zsh: no such file or directory: opt/anaconda3/bin/jupyter_mac.command Saving session...completed. [Process completed]

Answer:

The error you're seeing:

```
bash
CopyEdit
zsh: no such file or directory: opt/anaconda3/bin/jupyter_mac.command
```

means that the terminal is trying to run a file located at `opt/anaconda3/bin/jupyter_mac.command`, but it **doesn't exist**(or the path is incorrect).

Here's how to fix it:

1. Correct the path:

You're missing a `/` at the beginning of the path. It should be:

```
bash
CopyEdit
/opt/anaconda3/bin/jupyter_mac.command
```

To try again, run:

```bash
CopyEdit
/opt/anaconda3/bin/jupyter_mac.command
```

2. Check if the file exists:

To confirm the file is really there, run:

```bash
CopyEdit
ls /opt/anaconda3/bin/jupyter_mac.command
```

If it says **"No such file or directory"**, then the file is either:

- Deleted or not installed, or
- Named differently.

3. Open Jupyter another way:

Try starting Jupyter Notebook manually with:

```bash
CopyEdit
jupyter notebook
```

Or JupyterLab:

```bash
CopyEdit
jupyter lab
```

Make sure Anaconda is correctly installed and your environment is activated (`(base)` indicates that).

Would you like help checking if Jupyter is installed properly or launching it differently?

Realtime screen scanning and give answers to question llm

Answer:

To build a **real-time screen scanning and question-answering system using LLMs**, you'll need to combine multiple components:

System Overview

1. **Screen Capturing Module**: Captures the user's screen in real-time or at intervals.
2. **OCR (Optical Character Recognition)**: Converts screen image to text.
3. **LLM (e.g., GPT-4, local model)**: Answers questions based on extracted text.
4. **User Interface**: Lets the user ask questions and view answers.

Technologies You Can Use

Component	Tools / Libraries
Screen Capture	Python: `mss`, `pyautogui`, `PIL`
OCR	`Tesseract OCR` via `pytesseract`
LLM (Question Answering)	OpenAI API, HuggingFace Transformers, LangChain
GUI (Optional)	`Tkinter`, `PyQt5`, `Gradio`, or web frontend

Workflow

1. **Capture**: Periodically capture screen (or specific window).
2. **Extract Text**: Run OCR to get readable text.
3. **Input**: Let user ask a question.
4. **Query**: Send screen-extracted text + user question to LLM.
5. **Respond**: Display the answer.

Example (Python + OpenAI API)

```
python
CopyEdit
import pytesseract
import mss
from PIL import Image
import openai

openai.api_key = "YOUR_API_KEY"

def capture_screen():
    with mss.mss() as sct:
        screenshot = sct.grab(sct.monitors[1])
        img = Image.frombytes("RGB", screenshot.size, screenshot.bgra,
"raw", "BGRX")
        return img

def extract_text(img):
    return pytesseract.image_to_string(img)

def ask_llm(question, context):
    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are a helpful assistant
reading screen content."},
            {"role": "user", "content": f"Context: {context}\n\nQuestion:
{question}"}
        ]
    )
    return response.choices[0].message['content']

# --- Example usage ---
img = capture_screen()
text = extract_text(img)
answer = ask_llm("What is this screen about?", text)
print("Answer:", answer)
```

Considerations

- **Privacy**: Make sure users know screen content is being scanned and sent to LLM.
- **Performance**: Run screen captures and OCR efficiently (consider threading).
- **LLM Tokens**: Large screen text may exceed token limits – you may need summarization or chunking.

Would you like a sample app, or a GUI version of this?

Set the system prompt as neutral and focused. system_prompt = "You are an AI that provides detailed, clear, and concise answers based on the document provided. Use the document content to answer questions and provide references where applicable." # Now the query can be directly set by you as the user. query = "What is the '5-second rule' mentioned in the document? Please explain the concept in detail and provide references to the specific pages where this rule is discussed." # Execute the query print(f"\nQuery: {query}") print("Generating answer...") try: result = qa.invoke({"query": query}) print("\nAnswer:", result["result"]) print("\nSource documents:") for i, doc in enumerate(result["source_documents"]): print(f"\nDocument {i+1}:") print(f"Content: {doc.page_content[:150]}...") print(f"Source: Page {doc.metadata.get('page', 'unknown')}") except Exception as e: print(f"Error in query processing: {e}") try: print("Trying alternative query method...") result = qa.run(query) print("\nAnswer:", result) except Exception as e2: print(f"Alternative method also failed: {e2}") query is ask by me from user input untill i say stop it should ready to give me asnwers

Answer:

ChatGPT said:

To continuously accept user input and process queries until you say "stop," you can modify the script to allow for an ongoing loop that takes user input, processes the query, and then responds accordingly. Here's how you can structure the code:

Modified Version with Continuous User Input:

```
python
CopyEdit
# Set the system prompt as neutral and focused
system_prompt = "You are an AI that provides detailed, clear, and concise answers based on the document provided. Use the document content to answer questions and provide references where applicable."
```

```
# Start the continuous question loop
print("\n=== QA System Ready ===")
print("Type your questions about the document (type 'stop' to exit)\n")

while True:
    # Accept user input (query)
    query = input("Your question: ").strip()

    if query.lower() == "stop":
        print("Exiting the QA system...")
        break

    if not query:
        print("Please enter a question or 'stop' to exit")
        continue

    # Generate answer using the provided system prompt and query
    print(f"\nQuery: {query}")
    print("Generating answer...")

    try:
        # Pass the query to the QA system
        result = qa.invoke({"query": query})

        # Display the answer and source documents
        print("\nAnswer:", result["result"])
        print("\nSource documents:")
        for i, doc in enumerate(result["source_documents"]):
            print(f"\nDocument {i+1}:")
            print(f"Content: {doc.page_content[:150]}...")
            print(f"Source: Page {doc.metadata.get('page', 'unknown')}")

    except Exception as e:
        print(f"Error in query processing: {e}")
        try:
            # Try alternative method for querying
            print("Trying alternative query method...")
            result = qa.run(query)
            print("\nAnswer:", result)
        except Exception as e2:
            print(f"Alternative method also failed: {e2}")
```
Explanation of Modifications:

1. **Continuous Loop**:
    - The loop will continuously prompt the user for input and process the query until the user types "stop."
2. **User Query**:
    - Each time the loop runs, the script takes user input (`query`), processes it with the QA system, and outputs the result.
3. **Stop Condition**:
    - The loop breaks and stops when the user types "stop."
4. **Error Handling**:
    - If an error occurs while processing the query, it will attempt a fallback query method (`qa.run()`), and if that fails too, it prints the error.

Example Interaction:

```
vbnet
```

```
CopyEdit
=== QA System Ready ===
Type your questions about the document (type 'stop' to exit)

Your question: What is the '5-second rule' mentioned in the document?
Generating answer...

Answer: The '5-second rule' is a concept discussed in the document. It
suggests that food dropped on the floor is still safe to eat if picked up
within five seconds...
Source documents:
Document 1:
Content: The '5-second rule' is a popular idea that claims food dropped on
the floor is safe to eat if retrieved within five seconds...
Source: Page 5

Your question: How does this rule apply to health?
Generating answer...

Answer: According to the document, the '5-second rule' is more about
perception than actual science. It discusses how bacteria can contaminate
food instantly upon contact...
Source documents:
Document 2:
Content: The document argues that the rule is more of a social myth rather
than based on scientific evidence, as contamination can occur
immediately...
Source: Page 7

Your question: stop
Exiting the QA system...
```
Key Features:

- Continuous interaction with the system, processing new queries until you stop.
- Each question is answered based on the document with references to the source pages.
- The code gracefully handles errors and fallback methods in case the query processing fails.

This should meet your requirement for a continuous, interactive QA system that only stops when you say "stop."

Conclusion

How to access the system and final brief about the chatbot

Installation Requirements

Prerequisites:
- Python 3.8 or higher
- pip package manager

Required Packages:

Install the following packages:

pip install langchain langchain-community langchain-ollama faiss-cpu chromadb sentence-transformers ollama langchain-huggingface

Make sure to update to the latest versions:

pip install --upgrade langchain-ollama langchain langchain-community
pip install -U langchain-huggingface

Ollama Setup:
1. Install Ollama by following instructions at https://ollama.com
2. Pull the required models:
   ollama pull llama3.2:latest
   ollama pull deepseek-r1:1.5b-qwen-distill-q4_K_M

Implementation Details

The implementation consists of two main components:

1. Document Ingestion Pipeline
   - PDF Directory: Current directory (./)
   - Vector Store: Chroma database stored in ./chroma_db_sha
   - Chunk Size: 200 characters
   - Chunk Overlap: 50 characters
   - Embedding Model: all-MiniLM-L6-v2 (HuggingFace)

2. Interactive Q&A Interface
   - Interactive question answering
   - Source document references
   - Visual loading indicator (in notebook environment)
   - Clear presentation of answers

Usage Instructions

Preparing Documents:

1. Place PDF files in the script's directory.
2. Run the document ingestion function with rerun=True to force re-embedding:

```
qa = ingest_and_build(rerun=True)
```

Querying Documents:

Use the interactive query interface:

```
print("Type questions below (or 'stop' to exit):")
while True:
    query = input("□ ").strip()
    if query.lower() in ("stop", "exit", "quit"):
        clear_output(wait=True)
        print("Goodbye! □")
        break
    if not query:
        continue
    answer_question(qa, query)
```

Technical Components

Document Loading:
Uses PyPDFLoader from LangChain to extract text from PDF files.

Text Splitting:
Uses RecursiveCharacterTextSplitter to divide documents into manageable chunks.

Embeddings:
Uses HuggingFace's sentence transformer model to create vector embeddings.

Vector Store:
Uses Chroma for storing and retrieving embeddings.

LLM Integration:
Uses Ollama to run LLMs locally.

Prompt Template:
Uses a custom prompt to instruct the LLM.
RetrievalQA Chain:
Combines retriever with LLM to create a QA system.

Customization Options

Changing Models:
To use a different model, update the LLM_MODEL variable:

```
LLM_MODEL = "your-preferred-model"
```

Adjusting Retrieval Parameters:
To change how many document chunks are retrieved:

```
RETRIEVE_K = 4
```

Modifying Chunk Size:
For different document types, adjust chunking parameters:

```
CHUNK_SIZE = 200
CHUNK_OVERLAP = 50
```

Troubleshooting

Common Issues:

1. Model not found errors:

   Ensure you've pulled the required models with Ollama.

2. Empty or "I don't know" responses:

   - Check that your PDFs were loaded correctly
   - Try adjusting the chunk size and overlap
   - Increase the number of retrieved documents (RETRIEVE_K)

3. Slow response times:

   - Try a smaller LLM if available
   - Reduce the number of retrieved documents
   - Check available system resources

Video link –

https://drive.google.com/file/d/1LDH-LzqOrvDR2q0--9beUBUWBmSd-Z-t/view?usp=sharing

Github repo –

https://github.com/UdeeshaRukshan/Assignment2.git

References

[1] https://www.langflow.org

[2] https://www.langchain.com

[3] https://platform.openai.com/docs/overview

[4] https://medium.com/@udeesharukshan/understanding-rag-a-simple-guide-to-smarter-answers-bd079d9773bb

[5] https://www.llama.com

[6] https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-7B