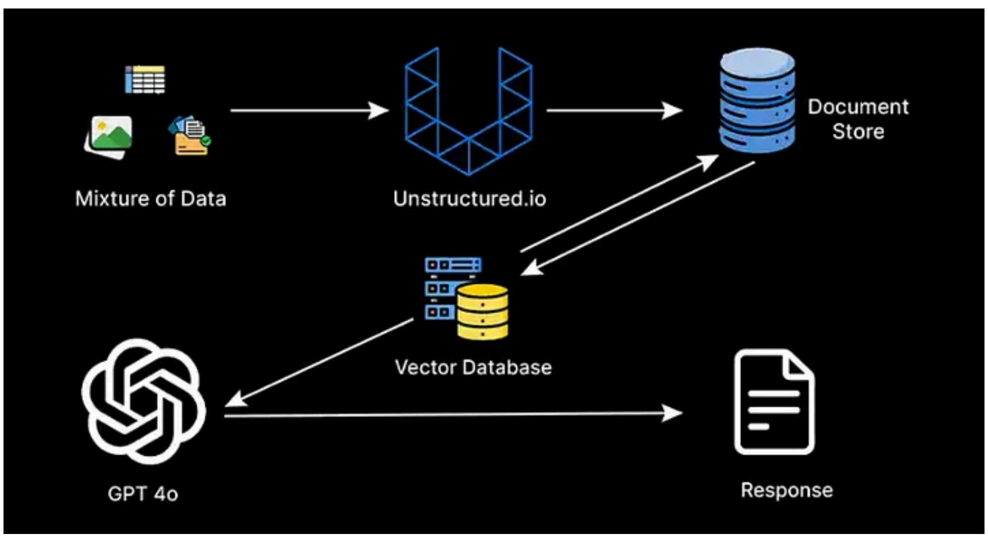
Building a Multimodal Retrieval-Augmented Generation (RAG) Application with Streamlit

Aashish Singh · Follow
11 min read · 1 day ago



In today's data-rich environment, organizations face the challenge of efficiently extracting insights from diverse types of content — text, tables, images, and more. Traditional search methods based solely on keyword matching often fall short when faced with such variety. In this post, we'll build an interactive Streamlit application that leverages state-of-the-art techniques in natural language processing (NLP) and vector search to create a robust multimodal Retrieval-Augmented Generation (RAG) system.

Our demo app processes PDFs, extracts text, tables, and images, summarizes them using advanced language models, stores the summaries in a vector database, and finally retrieves the most relevant content to answer user queries. Let's break down the code into digestible sections and understand the role of each component.

1. Setting the Stage: Libraries and Environment Setup

The first step is importing the libraries and setting up our environment. We rely on several powerful tools:

- **Streamlit** to create a user-friendly web interface.
- **LangChain** for chaining together document processing, summarization, and retrieval tasks.
- **Qdrant** as our vector database to store embeddings and perform similarity search.
- **HuggingFace models** and **OpenAI's GPT-4o-mini** for text and image summarization.
- Additional libraries for PDF processing, OCR, and metadata extraction (like Tesseract, NLTK, and LlamaIndex).

```
import streamlit as st
import os
import shutil
from dotenv import load_dotenv

# LLM / Summaries / Documents
from langchain_core.documents import Document
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.messages import HumanMessage
from langchain.chat_models import ChatOpenAI

# Embeddings
from langchain.embeddings import HuggingFaceEmbeddings

# PDF / Unstructured
from streamlit_pdf_viewer import pdf_viewer  # For displaying PDF in the front-e
from langchain_community.document_loaders import UnstructuredPDFLoader
import htmltabletomd

# Qdrant
from qdrant_client import QdrantClient
from qdrant_client.http.models import VectorParams, Distance, Filter, FieldCondi
from langchain.vectorstores import Qdrant as QdrantVectorStore

# Additional retrieval components
```

```
from langchain.retrievers import EnsembleRetriever
from langchain_community.retrievers import BM25Retriever
from langchain_community.cross_encoders import HuggingFaceCrossEncoder
from langchain.retrievers.document_compressors import CrossEncoderReranker
from langchain.retrievers import ContextualCompressionRetriever

# NLTK / OCR
import nltk
from unstructured_pytesseract import pytesseract

# LlamaIndex for metadata extraction
import nest_asyncio
nest_asyncio.apply()
from llama_index.llms.openai import OpenAI as LlamaIndexOpenAI
from llama_index.core.ingestion import IngestionPipeline
from llama_index.core.node_parser import TokenTextSplitter
from llama_index.core.schema import Document as LlamaDocument
from llama_index.core.extractors import TitleExtractor, QuestionsAnsweredExtract
from pydantic import Field

# openai for optional use in filtering
import openai
import json
import re
import base64
import io  # for decoding and displaying images

# Tesseract (if needed)
pytesseract.tesseract_cmd = r"C:\Program Files\Tesseract-OCR\tesseract.exe"

load_dotenv()

# Ensure NLTK downloads required resources
nltk.download("punkt")
nltk.download("averaged_perceptron_tagger")
```

2. Summarizing Images for Enhanced Retrieval

Images in PDFs — such as graphs, charts, or tables — contain valuable information. However, raw image data cannot be directly compared with text queries. Our solution involves converting images to base64 strings and summarizing them using an LLM.

Image Encoding and Summarization Functions

```
def encode_image(image_path):
    """Read an image from disk and return a base64-encoded string."""
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode("utf-8")

def image_summarize(img_base64, prompt):
    """Call a GPT-4-like model to generate a summary of the image."""
    chat = ChatOpenAI(model_name="gpt-4o-mini", temperature=0)
    msg = chat.invoke(
```

```
            [
                HumanMessage(
                    content=(
                        prompt
                        + "\n\nHere is the image in base64:\n"
                        + f"data:image/jpeg;base64,{img_base64}"
                    )
                )
            ]
        )
        return msg.content

    def generate_img_summaries(image_folder):
        """
        Process each .jpg image in the provided folder.
        Returns two lists:
        - img_base64_list: Base64-encoded images.
        - image_summaries: LLM-generated summaries for each image.
        """
        img_base64_list = []
        image_summaries = []
        prompt = """You are an assistant tasked with summarizing images for retrieva
    Remember these images could potentially contain graphs, charts or tables as well
    Provide a detailed, retrieval-optimized summary of the image content without add

        for i, fn in enumerate(sorted(os.listdir(image_folder))):
            if fn.lower().endswith(".jpg"):
                full_path = os.path.join(image_folder, fn)
                b64_img = encode_image(full_path)
                img_base64_list.append(b64_img)
                summary = image_summarize(b64_img, prompt)
                image_summaries.append(summary)
        return img_base64_list, image_summaries
```

Why Summarize Images?

Converting images to descriptive summaries makes it possible for the system to:

- **Index visual data** similarly to text.
- **Retrieve and display relevant images** based on user queries.
- **Enhance the overall multimodal search capability** of the application.

3. Metadata Extraction and Filtering

A critical part of our application is the ability to extract metadata from the content and later use that metadata to refine search results. For instance, when a user poses a question, we can dynamically decide which metadata keys are relevant to the query.

Filtering Metadata Based on User Queries

```
def filter_metadata_by_query(unique_values_json, user_query, openai_api_key):
    """
    Use an LLM to determine which metadata key-value pairs from the master JSON
    Returns a dict that can be used to filter search results in Qdrant.
    """
    openai.api_key = openai_api_key

    prompt = f"""
You will be given a user query and a master JSON describing possible metadata.
Your job is to figure out which key-value pair(s) in the master JSON best match
Focus only on picking keys from the master data and one of their possible values

Master data JSON: {json.dumps(unique_values_json)}
User Query: {user_query}

Return a valid JSON (key-value pairs) with no extra text.
If no metadata match, return an empty JSON like {{}}
"""
    system_prompt = {
        "role": "system",
        "content": "You are a JSON extraction expert. Output valid JSON only."
    }
    user_msg = {"role": "user", "content": prompt}

    try:
        response = openai.chat.completions.create(
            model="gpt-4o-mini",
            messages=[system_prompt, user_msg],
            temperature=0
        )
        content = response.choices[0].message.content.strip()

        # Extract JSON from the response
        json_pattern = r"\{.*?\}"
        matches = re.findall(json_pattern, content, re.DOTALL)
        if not matches:
            return {}

        extracted_json = json.loads(matches[0])
        return extracted_json
    except Exception:
        return {}
```

The Role of Metadata in Retrieval

- **Dynamic Filtering:** By matching query terms to metadata keys, the system can perform more context-aware searches.
- **Improved Relevance:** Filtering ensures that only documents matching the desired criteria are considered during retrieval.
- **Enhanced User Experience:** Users receive more precise answers, as the system filters out irrelevant content before generating a response.

4. PDF Ingestion, Chunking, and Summarization

Most documents come in the form of PDFs, which can be complex and contain various data types. We leverage the `UnstructuredPDFLoader` to extract text, images, and tables. Then, we split the content into smaller chunks and summarize each chunk.

Loading and Processing PDFs

```
def load_and_process_pdf(pdf_path, images_path):
    """
    Load a PDF with UnstructuredPDFLoader which extracts text, tables, and image
    Saves extracted images to the specified directory.
    """
    loader = UnstructuredPDFLoader(
        file_path=pdf_path,
        strategy="hi_res",
        extract_images_in_pdf=True,
        infer_table_structure=True,
        chunking_strategy="by_title",
        max_characters=1000,
        new_after_n_chars=1000,
        mode="elements",
        image_output_dir_path=images_path
    )
    data = loader.load()
    return data
```

Splitting Documents and Converting Tables

After extraction, we separate text and table chunks, converting table HTML to Markdown for better readability and further processing.

```
def split_docs_and_tables(data):
    """Separate text chunks from table chunks and convert table HTML to Markdown
    docs = []
    tables = []
    for doc in data:
        if "category" in doc.metadata:
            docs.append(doc)
            if "text_as_html" in doc.metadata:
                tables.append(doc)
    for table in tables:
        html_table = table.metadata.get("text_as_html", "")
        markdown_table = htmltabletomd.convert_table(html_table)
        table.metadata["table_markdown"] = markdown_table
    return docs, tables
```

Summarizing Chunks with an LLM Chain

We create a summarization chain that sends each chunk through a GPT model for a concise yet detailed summary.

```
def create_summarization_chain(chat_model):
    """Create a chain that summarizes content (text or tables) using a ChatPromp
    prompt_text = """
        You are an assistant tasked with summarizing tables and text for semanti
        These summaries will be embedded and used to retrieve the raw content.
        Provide a detailed summary that is optimized for retrieval.
        Do not include any extraneous words like "Summary:".

        Content:
        {element}
    """
    prompt = ChatPromptTemplate.from_template(prompt_text)
    summarize_chain = (
        {"element": RunnablePassthrough()}
        | prompt
        | chat_model
        | StrOutputParser()
    )
    return summarize_chain

def summarize_chunks(docs, tables, summarize_chain):
    """Generate summaries for text and table chunks."""
    text_chunks = [doc.page_content for doc in docs]
    table_chunks = [tbl.metadata.get("table_markdown", "") for tbl in tables]
    text_summaries = summarize_chain.batch(text_chunks, {"max_concurrency": 5})
    table_summaries = summarize_chain.batch(table_chunks, {"max_concurrency": 5}
    return text_summaries, table_summaries
```

Importance of Document Summarization

- **Efficient Indexing:** Summaries reduce the noise from long documents, allowing us to focus on essential information.
- **Improved Retrieval:** Semantic summaries enable better matching between user queries and stored content.
- **Faster Response Times:** Smaller chunks mean quicker processing during the retrieval and generation phases.

5. Storing Summaries in Qdrant for Semantic Search

Once the content is summarized, we store it in Qdrant — a vector database optimized for similarity search. Each summary is converted into an embedding, which represents the semantic meaning of the text.

```
def store_summaries_in_qdrant(
    text_summaries,
    table_summaries,
    docs,
```

```
        tables,
        img_base64_list,
        image_summaries,
        embeddings
    ):
        """
        Create or recreate a Qdrant collection and store text, table, and image summ
        This function also merges unstructured PDF metadata with metadata from Llama
        """
        emb_dim = embeddings.client.get_sentence_embedding_dimension()
        collection_name = "test_collection"
        client = QdrantClient(url="http://localhost:6333")
        client.recreate_collection(
            collection_name=collection_name,
            vectors_config=VectorParams(size=emb_dim, distance=Distance.COSINE),
        )
        vectorstore = QdrantVectorStore(
            client=client,
            collection_name=collection_name,
            embeddings=embeddings
        )
        summary_docs = []

        # Process text summaries
        for idx, summ in enumerate(text_summaries):
            orig_chunk = docs[idx]
            file_name = orig_chunk.metadata.get("filename", "unknown_file.pdf")
            file_title = orig_chunk.metadata.get("title", "Generic Title")
            nodes = extract_metadata_with_llamaindex(orig_chunk.page_content, file_t
            node_meta = nodes[0].metadata if nodes else {}
            for k, v in orig_chunk.metadata.items():
                node_meta.setdefault(k, v)
            doc_for_store = Document(page_content=summ, metadata=node_meta)
            summary_docs.append(doc_for_store)

        # Process table summaries
        for idx, summ in enumerate(table_summaries):
            orig_table = tables[idx]
            file_name = orig_table.metadata.get("filename", "unknown_file.pdf")
            file_title = orig_table.metadata.get("title", "Generic Title")
            nodes = extract_metadata_with_llamaindex(orig_table.metadata.get("table_
            node_meta = nodes[0].metadata if nodes else {}
            for k, v in orig_table.metadata.items():
                node_meta.setdefault(k, v)
            doc_for_store = Document(page_content=summ, metadata=node_meta)
            summary_docs.append(doc_for_store)

        # Process image summaries
        image_docs = []
        for i, summary_text in enumerate(image_summaries):
            b64_img = img_base64_list[i]
            nodes = extract_metadata_with_llamaindex(summary_text, "Image Chunk", f"
            meta_img = nodes[0].metadata if nodes else {}
            meta_img["is_image"] = True
            meta_img["image_base64"] = b64_img
            doc_for_store = Document(page_content=summary_text, metadata=meta_img)
            image_docs.append(doc_for_store)

        all_docs = summary_docs + image_docs
        vectorstore.add_documents(all_docs)
        return vectorstore, all_docs
```

6. Fusion Retriever: Combining Multiple Retrieval Strategies

Our retrieval system leverages an ensemble of different retrievers to maximize accuracy:

- **BM25 Retriever:** Uses traditional keyword matching.
- **Qdrant Retriever:** Uses semantic similarity search based on embeddings.
- **Cross-Encoder Reranker:** Fine-tunes the retrieved results by scoring them with a cross-encoder model.

Building the Fusion Retriever

```
def build_fusion_retriever(collection_name, embeddings, all_docs, top_k=5):
    bm25_retriever = BM25Retriever.from_documents(all_docs, k=top_k)
    client = QdrantClient(url="http://localhost:6333")
    qdrant_store = QdrantVectorStore(
        client=client,
        collection_name=collection_name,
        embeddings=embeddings
    )
    qdrant_retriever = qdrant_store.as_retriever(
        search_type="similarity",
        search_kwargs={"k": top_k}
    )
    ensemble_retriever = EnsembleRetriever(
        retrievers=[bm25_retriever, qdrant_retriever],
        weights=[0.5, 0.5]
    )
    reranker_model = HuggingFaceCrossEncoder(model_name="BAAI/bge-reranker-v2-m3
    reranker_compressor = CrossEncoderReranker(model=reranker_model, top_n=top_k
    final_retriever = ContextualCompressionRetriever(
        base_compressor=reranker_compressor,
        base_retriever=ensemble_retriever
    )
    return final_retriever
```

The Fusion Advantage

By combining different approaches, our system benefits from:

- **Robustness:** Mitigates the weaknesses of a single retrieval method.
- **Accuracy:** A cross-encoder reranker further refines results to ensure the best match is found.
- **Flexibility:** The ensemble can easily be tuned (via weights) based on performance and use case.

7. Bringing It All Together: The Interactive Streamlit App

The final step is integrating all these components into a seamless, interactive web application using Streamlit. The app allows users to:

- **Upload PDFs:** The user can upload one or more PDFs for processing.
- **Visualize Content:** PDFs are displayed in-browser, and summaries for text, tables, and images are shown.
- **Ask Questions:** Users can input natural language queries and receive precise, context-aware answers based on the retrieved content.

The Main Application Function

```
def main():
    st.set_page_config(page_title="Multimodal RAG Demo", layout="wide")
    st.title("Multimodal RAG: Text, Tables & Images")

    # Load API keys and check configuration
    openai_key = os.getenv("OPENAI_API_KEY", "")
    if not openai_key:
        st.warning("No OPENAI_API_KEY found in .env. Some GPT calls may fail.")

    # Initialize session state for retriever and documents
    if "final_retriever" not in st.session_state:
        st.session_state["final_retriever"] = None
    if "unique_values_per_key" not in st.session_state:
        st.session_state["unique_values_per_key"] = {}
    if "all_docs" not in st.session_state:
        st.session_state["all_docs"] = []

    st.sidebar.header("Upload & Summarize PDFs")
    uploaded_files = st.sidebar.file_uploader(
        "Upload one or more PDF files",
        type=["pdf"],
        accept_multiple_files=True
    )

    if uploaded_files and st.sidebar.button("Process PDFs"):
        embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all
        chat_model = ChatOpenAI(model_name="gpt-4o-mini", temperature=0)
        summarize_chain = create_summarization_chain(chat_model)

        combined_docs, combined_tables = [], []
        combined_text_summaries, combined_table_summaries = [], []
        combined_imgs_base64, combined_image_summaries = [], []

        for file_idx, uploaded_file in enumerate(uploaded_files):
            safe_pdf_name = sanitize_filename(uploaded_file.name)
            pdf_folder = f"{safe_pdf_name}_files"
            ensure_folder(pdf_folder)

            pdf_path = os.path.join(pdf_folder, uploaded_file.name)
            with open(pdf_path, "wb") as f:
                f.write(uploaded_file.getvalue())

            st.subheader(f"Processing PDF #{file_idx+1}: {uploaded_file.name}")
```

```
        st.write(f"Saved PDF locally at: {pdf_path}")

        with st.expander(f"1) View PDF ({uploaded_file.name})", expanded=Fal
            pdf_viewer(uploaded_file.getvalue(), width=700)

        images_path = "./figures"
        if os.path.exists(images_path):
            shutil.rmtree(images_path)
        os.makedirs(images_path)

        with st.expander(f"2) Parse {uploaded_file.name}", expanded=False):
            data = load_and_process_pdf(pdf_path, images_path)
            st.write(f"Number of data elements loaded: {len(data)}")

        with st.expander(f"3) Split into Text & Table chunks", expanded=Fals
            docs, tables = split_docs_and_tables(data)
            st.write(f"Found {len(docs)} text chunks, {len(tables)} table ch

        with st.expander(f"4) Summarize Text & Tables", expanded=False):
            text_summaries, table_summaries = summarize_chunks(docs, tables,
            st.write(f"Summaries => {len(text_summaries)} text / {len(table_

        with st.expander(f"5) Extract & Summarize Images", expanded=False):
            if os.path.exists(images_path):
                imgs_base64, image_summaries = generate_img_summaries(images
                st.write(f"Extracted {len(imgs_base64)} images.")
            else:
                imgs_base64, image_summaries = [], []
                st.write("No images folder found.")

        text_chunks_info = []
        if text_summaries:
            with st.expander("Text Chunks", expanded=False):
                for i, doc_obj in enumerate(docs):
                    chunk_item = {
                        "chunk_index": i+1,
                        "chunk_text": doc_obj.page_content,
                        "metadata": doc_obj.metadata,
                        "summary": text_summaries[i],
                    }
                    text_chunks_info.append(chunk_item)
                    st.markdown(f"**Text Chunk {i+1}**")
                    st.write("**Original Text:**", doc_obj.page_content)
                    st.write("**Metadata:**", doc_obj.metadata)
                    st.write("**Summary:**", text_summaries[i])
                    st.markdown("---")

        table_chunks_info = []
        if table_summaries:
            with st.expander("Table Chunks", expanded=False):
                for i, tbl_obj in enumerate(tables):
                    md_table = tbl_obj.metadata.get("table_markdown", "")
                    chunk_item = {
                        "table_index": i+1,
                        "markdown_table": md_table,
                        "metadata": tbl_obj.metadata,
                        "summary": table_summaries[i],
                    }
                    table_chunks_info.append(chunk_item)
                    st.markdown(f"**Table {i+1}**")
                    st.markdown(md_table)
                    st.write("**Metadata:**", tbl_obj.metadata)
                    st.write("**Summary:**", table_summaries[i])
                    st.markdown("---")

        image_chunks_info = []
        if imgs_base64:
            with st.expander("Images", expanded=False):
                for i, (b64_img, summary_text) in enumerate(zip(imgs_base64,
```

```
                        chunk_item = {
                            "image_index": i+1,
                            "image_base64": b64_img,
                            "summary": summary_text,
                        }
                        image_chunks_info.append(chunk_item)
                        decoded_image = base64.b64decode(b64_img)
                        st.markdown(f"**Image {i+1}**")
                        st.image(io.BytesIO(decoded_image), caption=f"Image {i+1
                        st.write("**Summary:**", summary_text)
                        st.markdown("---")

            save_processed_data_to_json(
                save_folder=pdf_folder,
                pdf_name=uploaded_file.name,
                text_chunks_info=text_chunks_info,
                table_chunks_info=table_chunks_info,
                image_chunks_info=image_chunks_info
            )

            combined_docs.extend(docs)
            combined_tables.extend(tables)
            combined_text_summaries.extend(text_summaries)
            combined_table_summaries.extend(table_summaries)
            combined_imgs_base64.extend(imgs_base64)
            combined_image_summaries.extend(image_summaries)

            st.markdown("---")

        st.subheader("Storing Data in Qdrant")
        qdrant_store, all_docs = store_summaries_in_qdrant(
            combined_text_summaries,
            combined_table_summaries,
            combined_docs,
            combined_tables,
            combined_imgs_base64,
            combined_image_summaries,
            embeddings
        )
        st.session_state["all_docs"] = all_docs
        st.write(f"Total combined docs stored: {len(all_docs)}")

        st.subheader("Building Fusion Retriever (BM25 + Qdrant + CrossEncoder)")
        final_retriever = build_fusion_retriever("pdf_collection", embeddings, a
        st.session_state["final_retriever"] = final_retriever
        st.success("Fusion retriever is ready.")

        # Gather unique metadata for filtering purposes
        meta_values = {}
        for d in all_docs:
            for k, v in d.metadata.items():
                if not isinstance(v, (str, list)):
                    continue
                if k not in meta_values:
                    meta_values[k] = set()
                if isinstance(v, list):
                    for item in v:
                        meta_values[k].add(item.strip())
                else:
                    meta_values[k].add(v.strip())
        unique_values_per_key = {k: list(v) for k, v in meta_values.items()}
        st.session_state["unique_values_per_key"] = unique_values_per_key

        st.markdown("---")
        st.success("All PDF(s) processed. You can now ask questions below.")

    st.header("Ask a Question")
    user_question = st.text_input("Enter your question:")
    if st.button("Submit Query"):
```

```
        if not user_question.strip():
            st.warning("Please enter a question.")
            return

        final_retriever = st.session_state.get("final_retriever", None)
        if not final_retriever:
            st.warning("No retriever found. Please process PDFs first.")
            return

        all_docs = st.session_state.get("all_docs", [])
        unique_values_json = st.session_state.get("unique_values_per_key", {})

        if not unique_values_json:
            st.info("No metadata available; searching without metadata filter.")
            qdrant_filter = None
        else:
            st.info("Extracting relevant metadata from your query...")
            relevant_filter = filter_metadata_by_query(unique_values_json, user_
            st.write("LLM-chosen metadata filter =>", relevant_filter)
            qdrant_filter = None
            if relevant_filter:
                conditions = []
                for key, val in relevant_filter.items():
                    if isinstance(val, list):
                        conditions.append(FieldCondition(key=key, match=MatchAny
                    else:
                        conditions.append(FieldCondition(key=key, match=MatchVal
                if conditions:
                    qdrant_filter = Filter(should=conditions)

        if qdrant_filter:
            base_ensemble = final_retriever.base_retriever
            if len(base_ensemble.retrievers) > 1:
                base_ensemble.retrievers[1].search_kwargs["filter"] = qdrant_fil

        retrieved_docs = final_retriever.get_relevant_documents(user_question)
        prompt = build_rag_prompt(user_question, retrieved_docs)
        chat_model = ChatOpenAI(model_name="gpt-4o-mini", temperature=0)
        with st.spinner("Generating answer..."):
            response = chat_model.invoke([HumanMessage(content=prompt)])
            st.markdown("### Answer")
            st.markdown(response.content)

if __name__ == "__main__":
    main()
```

The User Journey

1. **Upload PDFs:**
   Users start by uploading PDFs via the sidebar. The app processes each file, extracting text, tables, and images.
2. **Visualization and Summarization:**
   Processed content is displayed in expandable sections, allowing users to see the raw text, metadata, and generated summaries.

3. **Semantic Search and Q&A:**
   After processing, users can ask natural language questions. The fusion retriever searches for the most relevant content across all uploaded documents and generates a contextual answer using GPT.

Conclusion

This demo application illustrates how to integrate diverse NLP techniques, from advanced document processing and summarization to semantic search using vector databases. By combining multiple modalities — text, tables, and images — we've built a system capable of handling complex, unstructured data and answering queries with high precision.

Key Takeaways:

- **Multimodal Processing:** Extract and summarize data from various formats for better retrieval.
- **Semantic Search:** Store document embeddings in Qdrant to perform fast, accurate similarity searches.
- **Fusion Retrieval:** Combine keyword-based and semantic search methods for robust performance.
- **Interactive Experience:** Use Streamlit to create an intuitive and interactive interface for end users.

This system has applications in research, enterprise search, document management, and any scenario where deep insights must be derived from heterogeneous data. I hope this walkthrough inspires you to build and customize your own multimodal retrieval systems.

Feel free to share your thoughts or ask questions in the comments below. Happy coding and exploring the world of advanced data retrieval!

If you enjoyed this post, please share it with your network and follow for more insights on building cutting-edge AI applications.

Happy coding, and enjoy exploring the rich landscape of multimodal AI!

Stay Tuned!!

If you liked the article please do clap and follow for more!!

Thank you for reading!

If you'd like, ***add me on Linkedin!***

Retrieval Augmented Gen | Deep Learning | Machine Learning | Generative Ai Tools | Innovation

Written by Aashish Singh

Follow

12 Followers · 72 Following

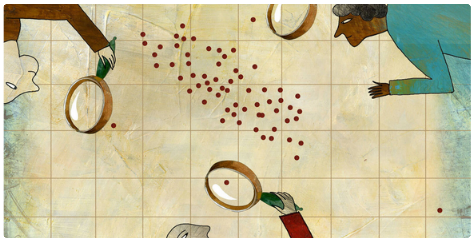
🎯 AI Engineer @ Wadhwani Center for Government Digital Transformation 🎓
MBA | Generative AI Innovator 💻 Certified ML Developer | AI Solutions Pioneer

No responses yet

What are your thoughts?

Respond

More from Aashish Singh

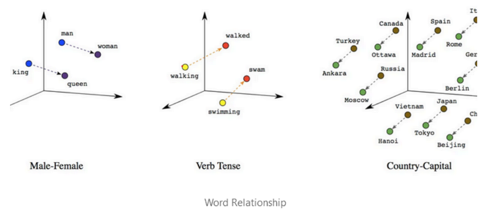


Aashish Singh

How to Identify and Remove Outliers in Your Dataset: A Guide…

Data is the lifeblood of any machine learning model. The quality of your data directly…

Aug 15, 2024

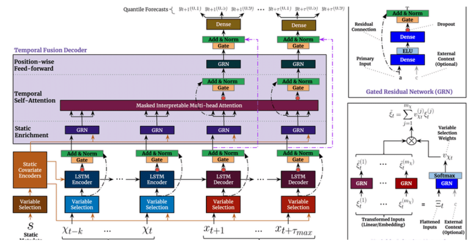


Aashish Singh

Unpacking Word Embeddings: A Journey Through Modern NLP…

In the world of Natural Language Processing (NLP), word embeddings have revolutionize…

Aug 13, 2024

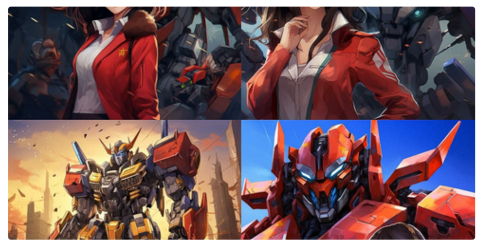


Aashish Singh

Unleashing the Power of Temporal Fusion Transformers in Time Seri…

Time series forecasting has always been a challenging task, especially when dealing…

Aug 9, 2024 61



Aashish Singh

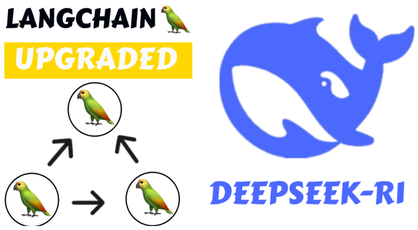
Understanding Transformer Architecture: The Backbone of…

Transformers have revolutionized the field of natural language processing (NLP) and…

Aug 10, 2024

See all from Aashish Singh

Recommended from Medium

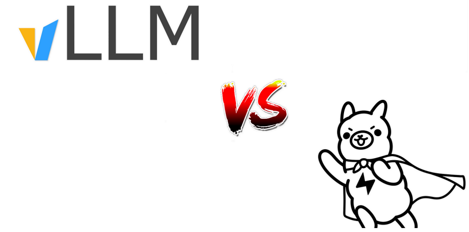


In Towards AI by Gao Dalie (高達烈)

Langchain (Upgraded) + DeepSeek-R1 + RAG Just...

Last week, I made a video about DeepSeek-V3, and it caused a huge stir in the global AI...

5d ago 516 3



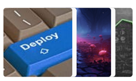
Henry Navarro

Ollama vs vLLM: which framework is better for inference? 👊 (Part I)

Part I of III: A comparative analysis of my two favorite frameworks for inferencing LLM...

Jan 21 62 1

Lists



Predictive Modeling w/ Python

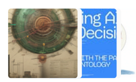
20 stories · 1811 saves

Practical Guides to Machine Learning

10 stories · 2185 saves

Natural Language Processing

1909 stories · 1568 saves



data science and AI

40 stories · 324 saves

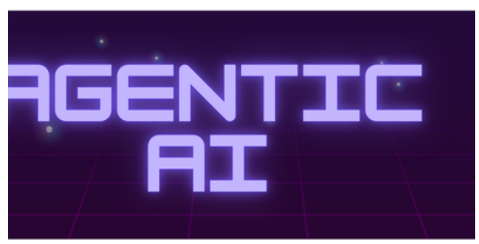
FastAP

Joey O'Neill

Full-stack RAG: FastAPI Backend (Part 1)

All of my articles are 100% free to read. Non-members can read for free by clicking this...

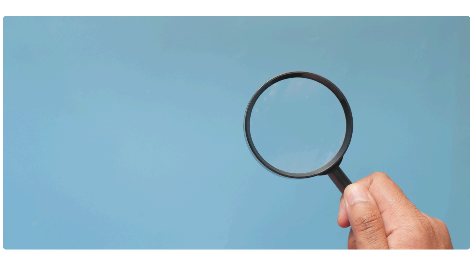
5d ago 23



In Vedcraft by Ankur Kumar

Building Intelligent Apps with Agentic AI: Top Frameworks to...

Jan 24 104 4



Shaw Talebi

Fine-Tuning Text Embeddings For Domain-Specific Search

An overview with Python code

Jan 24 273 2

Samar Singh

Qwen 2.5 Max: AI Model outperforms Deepseek V3

How to run it locally and through api?

4d ago 89 1

See more recommendations