Local RAG From Scratch

Develop and deploy an entirely local RAG system from scratch

Joe Sasson · Follow
Published in Towards Data Science · 18 min read · 2 days ago

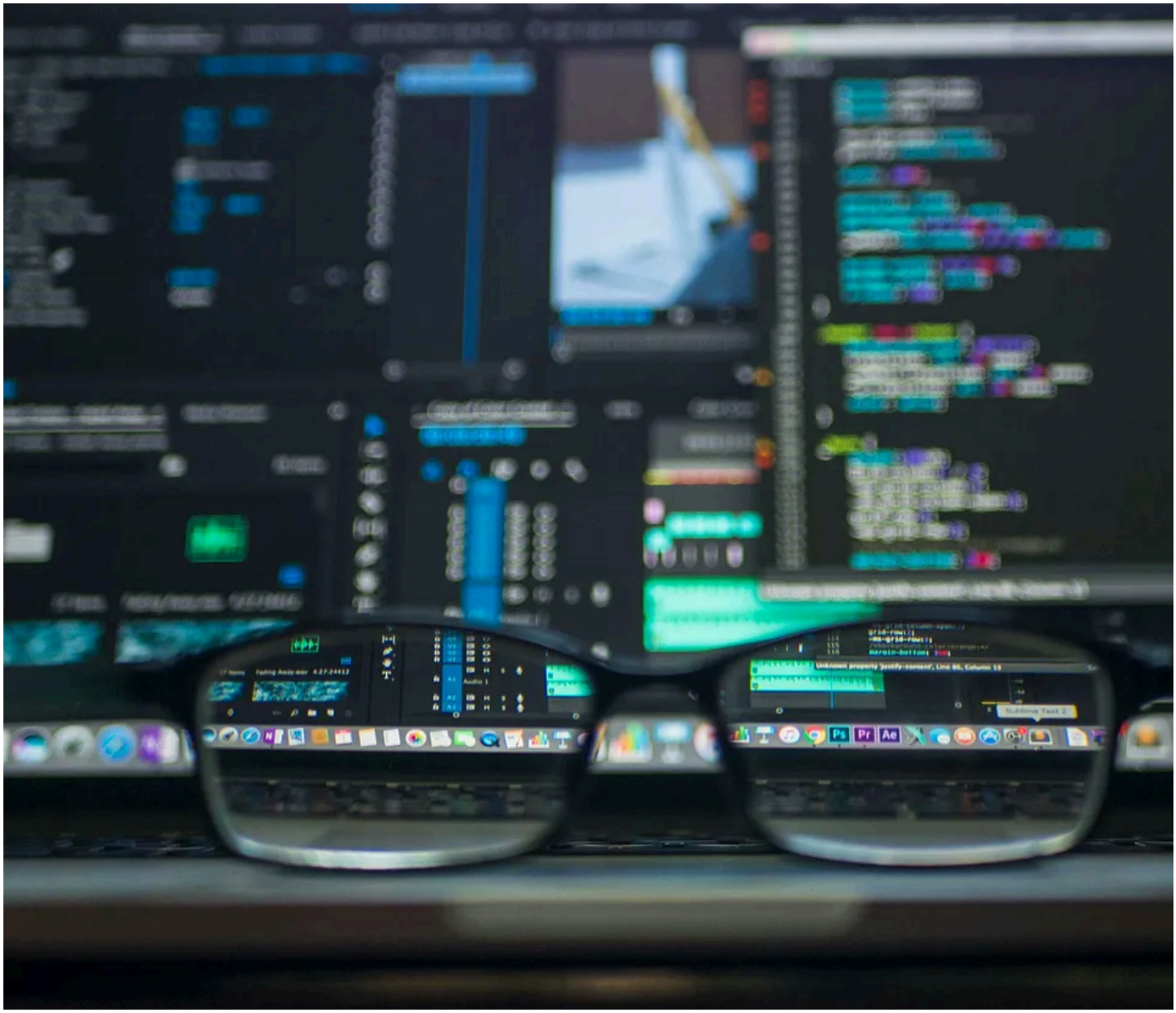


Photo by Kevin Ku on Unsplash

Introduction

High-level abstractions offered by libraries like llama-index and Langchain have simplified the development of Retrieval

Augmented Generation (RAG) systems. Yet, a deep understanding of the underlying mechanics enabling these libraries remains crucial for any machine learning engineer aiming to fully leverage their potential. In this article, I will guide you through the process of developing a RAG system from the ground up. I will also take it a step further, and we will create a containerized flask API. I have designed this to be highly practical: this walkthrough is inspired by real-life use cases, ensuring that the insights you gain are not only theoretical but immediately applicable.

Use-case overview — This implementation is designed to handle a wide array of document types. While the current example utilizes many small documents, each depicting individual products with details such as SKU, name, description, price, and dimensions, the approach is highly adaptable. Whether the task involves indexing a diverse library of books, mining data from extensive contracts, or any other set of documents, the system can be tailored to meet the specific needs of these varied contexts. This flexibility allows for the seamless integration and processing of different types of information.

Quick note — this implementation will work solely with text data. Similar steps can be followed to convert images to embeddings using a multi-modal model like CLIP, which you can then index and query against.

Table of Contents

Modular Framework

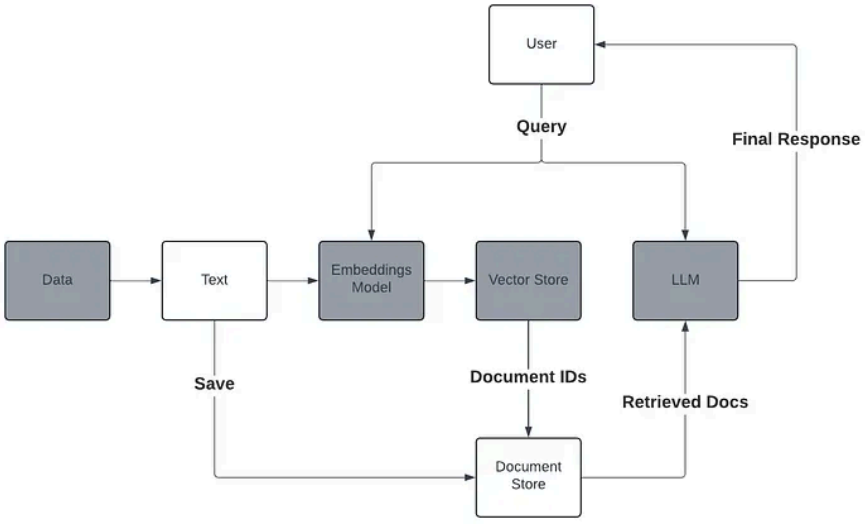
The implementation has four main components that can be swapped out.

- Text data
- Embedding model
- LLM
- Vector store

Integrating these services into your project is highly flexible, allowing you to tailor them according to your specific requirements. In this example

implementation, I start with a scenario where the initial data is in a JSON format, which conveniently provides the data as a string. However, you might encounter data in various other formats such as PDFs, emails, or Excel spreadsheets. In such cases, it is essential to "normalize" this data by converting it into a string format. Depending on the needs of your project, you can either convert the data to a string in memory or save it to a text file for further refinement or downstream processing.

Similarly, the choices of embeddings model, vector store, and LLM can be customized to fit your project's needs. Whether you require a smaller or larger model, or perhaps an external model, the flexibility of this approach allows you to simply swap in the appropriate options. This plug-and-play capability ensures that your project can adapt to various requirements without significant alterations to the core architecture.



Simplified Modular Framework. Image by author.

I highlighted the main components in gray. In this implementation our vector store will simply be a JSON file. Once again, depending on your use-case, you may want to just use an in-memory vector store (Python dict) if you're only processing one file at a time. If you need to persist this data, like we do for this use-case, you can save it to a JSON file locally. If you need to store hundreds of thousands or millions of vectors you would need an external vector store (Pinecone, Azure Cognitive Search, etc...).

Prepare Data

As mentioned above, this implementation starts with JSON data. I used GPT-4 and Claude to generate it synthetically. The data contains product descriptions for different pieces of furniture each with its own SKU. Here is an example:

```
{
    "MBR-2001": "Traditional sleigh bed crafted in rich walnut wood, featuring a
    "MBR-2002": "Art Deco-inspired vanity table in a polished ebony finish, feat
    "MBR-2003": "Set of sheer linen drapes in soft ivory, offering a delicate ar

    "LVR-3001": "Convertible sofa bed upholstered in navy blue linen fabric, eas
    "LVR-3002": "Ornate Persian area rug in deep red and gold, hand-knotted from
    "LVR-3003": "Contemporary TV stand in matte black with tempered glass doors

    "OPT-4001": "Modular outdoor sofa set in espresso brown polyethylene wicker,
    "OPT-4002": "Cantilever umbrella in sunflower yellow, featuring a 10-foot ca
    "OPT-4003": "Rustic fire pit table made from faux stone, includes a natural

    "ENT-5001": "Digital jukebox with touchscreen interface and built-in speaker
    "ENT-5002": "Gaming console storage unit in sleek black, featuring designate
    "ENT-5003": "Virtual reality gaming set by VR Innovations, includes headset,

    "KIT-6001": "Chef's rolling kitchen cart in stainless steel, features two sh
    "KIT-6002": "Contemporary pendant light cluster with three frosted glass sha
    "KIT-6003": "Eight-piece ceramic dinnerware set in ocean blue, includes dinn

    "GBR-7001": "Twin-size daybed with trundle in brushed silver metal, ideal fo
    "GBR-7002": "Wall art set featuring three abstract prints in blue and grey t
    "GBR-7003": "Set of two bedside lamps in brushed nickel with white fabric sh

    "BMT-8001": "Industrial-style pool table with a slate top and black felt, in
    "BMT-8002": "Leather home theater recliner set in black, includes four conne
    "BMT-8003": "Adjustable height pub table set with four stools, featuring a r
}
```


In a real world scenario, we can extrapolate this to millions of SKUs and descriptions, most likely all residing in different places. The effort of aggregating and organizing this data seems trivial in this scenario, but generally data in the wild would need to be organized into a structure like this.

The next step is to simply convert each SKU into its own text file. In total there are 105 text files (SKUs). ***Note — you can find all the data/code linked in my GitHub at the bottom of the article.***

I used this prompt to generate the data and sent it numerous times:

```
Given different "categories" for furniture, I want you to generate a synthetic '

Generate 3 for each category. Be extremely granular with your details and descri

Every response should follow this format and should be only JSON:
{<SKU>:<description>}.

- master bedroom
- living room
- outdoor patio
- entertainment
- kitchen
- guest bedroom
- finished basement
```

To move forward, you should have a directory with text files containing your product descriptions with the SKUs as the filenames.

Chunking, Indexing, & Retrieval

Chunking

Given a piece of text, we need to efficiently chunk it so that it is optimized for retrieval. I tried to model this after the llama-index SentenceSplitter class.

```
import re
import os
import uuid
from transformers import AutoTokenizer, AutoModel

def document_chunker(directory_path,
                     model_name,
                     paragraph_separator='\n\n',
                     chunk_size=1024,
                     separator=' ',
                     secondary_chunking_regex=r'\S+?[\.,;!?]',
                     chunk_overlap=0):

    tokenizer = AutoTokenizer.from_pretrained(model_name)  # Load tokenizer for
    documents = {}  # Initialize dictionary to store results

    # Read each file in the specified directory
    for filename in os.listdir(directory_path):
        file_path = os.path.join(directory_path, filename)
        base = os.path.basename(file_path)
        sku = os.path.splitext(base)[0]
        if os.path.isfile(file_path):
            with open(file_path, 'r', encoding='utf-8') as file:
                text = file.read()

            # Generate a unique identifier for the document
            doc_id = str(uuid.uuid4())

            # Process each file using the existing chunking logic
            paragraphs = re.split(paragraph_separator, text)
            all_chunks = {}
            for paragraph in paragraphs:
                words = paragraph.split(separator)
                current_chunk = ""
                chunks = []

                for word in words:
                    new_chunk = current_chunk + (separator if current_chunk else
                    if len(tokenizer.tokenize(new_chunk)) <= chunk_size:
                        current_chunk = new_chunk
                    else:
                        if current_chunk:
                            chunks.append(current_chunk)
                        current_chunk = word

                if current_chunk:
                    chunks.append(current_chunk)

                refined_chunks = []
                for chunk in chunks:
                    if len(tokenizer.tokenize(chunk)) > chunk_size:
                        sub_chunks = re.split(secondary_chunking_regex, chunk)
                        sub_chunk_accum = ""
                        for sub_chunk in sub_chunks:
                            if sub_chunk_accum and len(tokenizer.tokenize(sub_ch
                                refined_chunks.append(sub_chunk_accum.strip())
                                sub_chunk_accum = sub_chunk
```

```
                        else:
                            sub_chunk_accum += (sub_chunk + ' ')
                    if sub_chunk_accum:
                        refined_chunks.append(sub_chunk_accum.strip())
                else:
                    refined_chunks.append(chunk)

            final_chunks = []
            if chunk_overlap > 0 and len(refined_chunks) > 1:
                for i in range(len(refined_chunks) - 1):
                    final_chunks.append(refined_chunks[i])
                    overlap_start = max(0, len(refined_chunks[i]) - chunk_ov
                    overlap_end = min(chunk_overlap, len(refined_chunks[i+1]
                    overlap_chunk = refined_chunks[i][overlap_start:] + ' '
                    final_chunks.append(overlap_chunk)
                final_chunks.append(refined_chunks[-1])
            else:
                final_chunks = refined_chunks

            # Assign a UUID for each chunk and structure it with text and me
            for chunk in final_chunks:
                chunk_id = str(uuid.uuid4())
                all_chunks[chunk_id] = {"text": chunk, "metadata": {"file_na

        # Map the document UUID to its chunk dictionary
        documents[doc_id] = all_chunks

    return documents
```

The most important parameter here is the “chunk_size”. As you can see, we are using the transformers library to count the number of tokens in a given string. Therefore, the chunk_size represents the number of tokens in a chunk.

Here is breakdown of what is happening inside the function:

For every file in the specified directory →

1. ***Split Text into Paragraphs:***
   - Divide the input text into paragraphs using a specified separator.
2. ***Chunk Paragraphs into Words:***
   - For each paragraph, split it into words.
   - Create chunks of these words without exceeding a specified token count (chunk_size).
3. ***Refine Chunks:***
   - If any chunk exceeds the chunk_size, further split it using a regular expression based on punctuation.
   - Merge sub-chunks if necessary to optimize chunk size.
4. ***Apply Overlap:***
   - For sequences with multiple chunks, create overlaps between them to ensure contextual continuity.
5. ***Compile and Return Chunks:***
   - Loop over every final chunk, assign it a unique ID which maps to the text

and metadata of that chunk, and finally assign this chunk dictionary to the doc ID.

In this example, where we are indexing numerous smaller documents, the chunking process is relatively straightforward. Each document, being brief, requires minimal segmentation. This contrasts sharply with scenarios involving more extensive texts, such as extracting specific sections from lengthy contracts or indexing entire novels. To accommodate a variety of document sizes and complexities, I developed the `document_chunker` function. This allows you to input your data—regardless of its length or format—and apply the same efficient chunking process. Whether you are dealing with concise product descriptions or expansive literary works, the `document_chunker` ensures that your data is appropriately segmented for optimal indexing and retrieval.

Usage:

```
docs = document_chunker(directory_path='/Users/joesasson/Desktop/articles/rag-fr
                        model_name='BAAI/bge-small-en-v1.5',
                        chunk_size=256)

keys = list(docs.keys())
print(len(docs))
print(docs[keys[0]])

Out -->
105
{'61d6318e-644b-48cd-a635-9490a1d84711': {'text': 'Gaming console storage unit i
```

We now have a mapping with a unique doc ID, that points to all the chunks in that doc, each chunk having its own unique ID which points to the text and metadata of that chunk.

The metadata can hold arbitrary key/value pairs. Here I am setting the file name (SKU) as the metadata so we can trace our models results back to the original product.

Indexing

Now that we've created the document store, we need to create the vector store.

You may have already noticed, but we are using ***BAAI/bge-small-en-v1.5*** as our embeddings model. In the previous function, we only use it for tokenization, now we will use it to vectorize our text.

To prepare for deployment, let's save the tokenizer and model locally.

```
from transformers import AutoModel, AutoTokenizer

model_name = "BAAI/bge-small-en-v1.5"

tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)

tokenizer.save_pretrained("model/tokenizer")
model.save_pretrained("model/embedding")
```

```
def compute_embeddings(text):
    tokenizer = AutoTokenizer.from_pretrained("/model/tokenizer")
    model = AutoModel.from_pretrained("/model/embedding")

    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)

    # Generate the embeddings
    with torch.no_grad():
        embeddings = model(**inputs).last_hidden_state.mean(dim=1).squeeze()

    return embeddings.tolist()
```

```
def create_vector_store(doc_store):
    vector_store = {}
    for doc_id, chunks in doc_store.items():
        doc_vectors = {}
        for chunk_id, chunk_dict in chunks.items():
            # Generate an embedding for each chunk of text
            doc_vectors[chunk_id] = compute_embeddings(chunk_dict.get("text"))
        # Store the document's chunk embeddings mapped by their chunk UUIDs
        vector_store[doc_id] = doc_vectors
    return vector_store
```

All we've done is simply convert the chunks in the document store to embeddings. You can plug in any embeddings model, and any vector store. Since our vector store is just a dictionary, all we have to do is dump it into a JSON file to persist.

Retrieval

Now let's test it out with a query!

```
def compute_matches(vector_store, query_str, top_k):
    """
    This function takes in a vector store dictionary, a query string, and an int
    It computes embeddings for the query string and then calculates the cosine s
    The top_k matches are returned based on the highest similarity scores.
    """
    # Get the embedding for the query string
    query_str_embedding = np.array(compute_embeddings(query_str))
    scores = {}

    # Calculate the cosine similarity between the query embedding and each chunk
    for doc_id, chunks in vector_store.items():
        for chunk_id, chunk_embedding in chunks.items():
            chunk_embedding_array = np.array(chunk_embedding)
            # Normalize embeddings to unit vectors for cosine similarity calcula
```

```
            norm_query = np.linalg.norm(query_str_embedding)
            norm_chunk = np.linalg.norm(chunk_embedding_array)
            if norm_query == 0 or norm_chunk == 0:
                # Avoid division by zero
                score = 0
            else:
                score = np.dot(chunk_embedding_array, query_str_embedding) / (nc

            # Store the score along with a reference to both the document and th
            scores[(doc_id, chunk_id)] = score

    # Sort scores and return the top_k results
    sorted_scores = sorted(scores.items(), key=lambda item: item[1], reverse=Tru
    top_results = [(doc_id, chunk_id, score) for ((doc_id, chunk_id), score) in

    return top_results
```

The `compute_matches` function is designed to identify the top_k most similar text chunks to a given query string from a stored collection of text embeddings. Here's a breakdown:

1. Embed the query string
2. Calculate cosine similarity. For each chunk, the cosine similarity between the query vector and the chunk vector is computed.
   Here, `np.linalg.norm` computes the Euclidean norm (L2 norm) of the vectors, which is required for cosine similarity calculation.
3. Handle normilzation and compute dot product. The cosine similarity is defined as:

$$\text{cosine\_similarity} = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|}$$

Where **A** and **B** are vectors and **||A||** and **||B||** are their norms.

4. Sort and select the scores. The scores are sorted in descending order, and the top_k results are selected

Usage:

```
matches = compute_matches(vector_store=vec_store,
                query_str="Wall-mounted electric fireplace with realistic LED fl
                top_k=3)

# matches
[('d56bc8ca-9bbc-4edb-9f57-d1ea2b62362f',
  '3086bed2-65e7-46cc-8266-f9099085e981',
  0.8600385118142513),
 ('240c67ce-b469-4e0f-86f7-d41c630cead2',
  '49335ccf-f4fb-404c-a67a-19af027a9fc2',
  0.7067269230771228),
```

```
 ('53faba6d-cec8-46d2-8d7f-be68c3080091',
  'b88e4295-5eb1-497c-8536-59afd84d2210',
  0.6959163226146977)]

# plug the top match document ID keys into doc_store to access the retrieved cor
docs['d56bc8ca-9bbc-4edb-9f57-d1ea2b62362f']['3086bed2-65e7-46cc-8266-f9099085e9

# result
{'text': 'Wall-mounted electric fireplace with realistic LED flames and heat set
 'metadata': {'file_name': 'ENT-4001'}}
```

Where each tuple has the document ID, followed by the chunk ID, followed by the score.

Awesome, it's working! All there's left to do is connect the LLM component and run a full end-to-end test, then we are ready to deploy!

LLM Component

To enhance the user experience by making our RAG system interactive, we will be utilizing the `llama-cpp-python` library. Our setup will use a mistral-7B parameter model with GGUF 3-bit quantization, a configuration that provides a good balance between computational efficiency and performance. Based on extensive testing, this model size has proven to be highly effective, especially when running on machines with limited resources like my M2 8GB Mac. By adopting this approach, we ensure that our RAG system not only delivers precise and relevant responses but also maintains a conversational tone, making it more engaging and accessible for end users.

Quick note on setting up the LLM locally on a Mac— my preference is to use anaconda or miniconda. Make sure you've install an arm64 version and follow the setup instructions for 'metal' from the library, here.

Now, it's quite easy. All we need to do is define a function to construct a prompt that includes the retrieved documents and the users query. The response from the LLM will be sent back to the user.

I've defined the below functions to stream the text response from the LLM and construct our final prompt.

```
from llama_cpp import Llama
import sys

def stream_and_buffer(base_prompt, llm, max_tokens=800, stop=["Q:", "\n"], echo=

    # Formatting the base prompt
    formatted_prompt = f"Q: {base_prompt} A: "

    # Streaming the response from llm
    response = llm(formatted_prompt, max_tokens=max_tokens, stop=stop, echo=echo
```

```
    buffer = ""

    for message in response:
        chunk = message['choices'][0]['text']
        buffer += chunk

        # Split at the last space to get words
        words = buffer.split(' ')
        for word in words[:-1]:  # Process all words except the last one (which
            sys.stdout.write(word + ' ')  # Write the word followed by a space
            sys.stdout.flush()  # Ensure it gets displayed immediately

        # Keep the rest in the buffer
        buffer = words[-1]

    # Print any remaining content in the buffer
    if buffer:
        sys.stdout.write(buffer)
        sys.stdout.flush()

def construct_prompt(system_prompt, retrieved_docs, user_query):
    prompt = f"""{system_prompt}

    Here is the retrieved context:
    {retrieved_docs}

    Here is the users query:
    {user_query}
    """
    return prompt

# Usage
system_prompt = """
You are an intelligent search engine. You will be provided with some retrieved c

Your job is to understand the request, and answer based on the retrieved context
"""

retrieved_docs = """
Wall-mounted electric fireplace with realistic LED flames and heat settings. Fea
"""

prompt = construct_prompt(system_prompt=system_prompt,
                          retrieved_docs=retrieved_docs,
                          user_query="I am looking for a wall-mounted electric f

llm = Llama(model_path="/Users/joesasson/Downloads/mistral-7b-instruct-v0.2.Q3_K

stream_and_buffer(prompt, llm)
```

Final output which gets returned to the user:

> *"Based on the retrieved context, and the user's query, the Hearth & Home electric fireplace with realistic LED flames fits the description. This model measures 50 inches wide, 6 inches deep, and 21 inches high, and comes with a remote control for easy operation."*

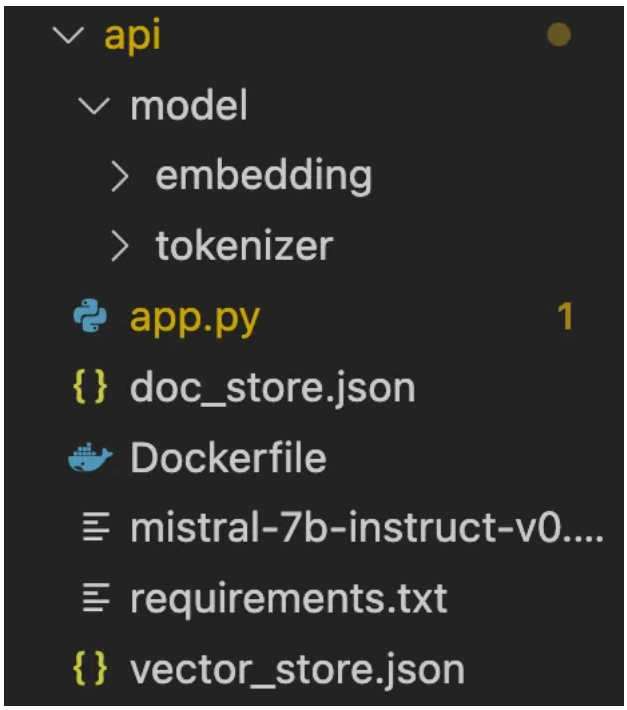
We are now ready to deploy our RAG system. Follow along in the next section and we will convert this quasi-spaghetti code into a consumable API for users.

Build & Deploy API

To extend the reach and usability of our system, we will package it into a containerized Flask application. This approach ensures that our model is encapsulated within a Docker container, providing stability and consistency regardless of the computing environment.

You should have downloaded the embeddings model and tokenizer above. Place these at the same level as your application code, requirements, and Dockerfile. You can download the LLM here.

You should have the following directory structure:



Deployment directory structure. Image by author.

app.py

```
from flask import Flask, request, jsonify
import numpy as np
import json
from typing import Dict, List, Any
from llama_cpp import Llama
import torch
import logging
from transformers import AutoModel, AutoTokenizer

app = Flask(__name__)

# Set the logger level for Flask's logger
app.logger.setLevel(logging.INFO)

def compute_embeddings(text):
    tokenizer = AutoTokenizer.from_pretrained("/app/model/tokenizer")
    model = AutoModel.from_pretrained("/app/model/embedding")

    inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)
```

```
    # Generate the embeddings
    with torch.no_grad():
        embeddings = model(**inputs).last_hidden_state.mean(dim=1).squeeze()

    return embeddings.tolist()

def compute_matches(vector_store, query_str, top_k):
    """
    This function takes in a vector store dictionary, a query string, and an int
    It computes embeddings for the query string and then calculates the cosine s
    The top_k matches are returned based on the highest similarity scores.
    """
    # Get the embedding for the query string
    query_str_embedding = np.array(compute_embeddings(query_str))
    scores = {}

    # Calculate the cosine similarity between the query embedding and each chunk
    for doc_id, chunks in vector_store.items():
        for chunk_id, chunk_embedding in chunks.items():
            chunk_embedding_array = np.array(chunk_embedding)
            # Normalize embeddings to unit vectors for cosine similarity calcula
            norm_query = np.linalg.norm(query_str_embedding)
            norm_chunk = np.linalg.norm(chunk_embedding_array)
            if norm_query == 0 or norm_chunk == 0:
                # Avoid division by zero
                score = 0
            else:
                score = np.dot(chunk_embedding_array, query_str_embedding) / (nc

            # Store the score along with a reference to both the document and th
            scores[(doc_id, chunk_id)] = score

    # Sort scores and return the top_k results
    sorted_scores = sorted(scores.items(), key=lambda item: item[1], reverse=Tru
    top_results = [(doc_id, chunk_id, score) for ((doc_id, chunk_id), score) in

    return top_results

def open_json(path):
    with open(path, 'r') as f:
        data = json.load(f)
    return data

def retrieve_docs(doc_store, matches):
    top_match = matches[0]
    doc_id = top_match[0]
    chunk_id = top_match[1]
    docs = doc_store[doc_id][chunk_id]
    return docs

def construct_prompt(system_prompt, retrieved_docs, user_query):
    prompt = f"""{system_prompt}

    Here is the retrieved context:
    {retrieved_docs}

    Here is the users query:
    {user_query}
    """
    return prompt

@app.route('/rag_endpoint', methods=['GET', 'POST'])
def main():
    app.logger.info('Processing HTTP request')

    # Process the request
    query_str = request.args.get('query') or (request.get_json() or {}).get('que
    if not query_str:
        return jsonify({"error":"missing required parameter 'query'"})

    vec_store = open_json('/app/vector_store.json')
    doc_store = open_json('/app/doc_store.json')

    matches = compute_matches(vector_store=vec_store, query_str=query_str, top_k
    retrieved_docs = retrieve_docs(doc_store, matches)
```

```
    system_prompt = """
    You are an intelligent search engine. You will be provided with some retriev

    Your job is to understand the request, and answer based on the retrieved cor
    """

    base_prompt = construct_prompt(system_prompt=system_prompt, retrieved_docs=r

    app.logger.info(f'constructed prompt: {base_prompt}')

    # Formatting the base prompt
    formatted_prompt = f"Q: {base_prompt} A: "

    llm = Llama(model_path="/app/mistral-7b-instruct-v0.2.Q3_K_L.gguf")
    response = llm(formatted_prompt, max_tokens=800, stop=["Q:", "\n"], echo=Fal

    return jsonify({"response": response})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001)
```

Dockerfile

```
# Use an official Python runtime as a parent image
FROM --platform=linux/arm64 python:3.11

# Set the working directory in the container to /app
WORKDIR /app

# Copy the requirements file
COPY requirements.txt .

# Update system packages, install gcc and Python dependencies
RUN apt-get update && \
    apt-get install -y gcc g++ make libtool && \
    apt-get upgrade -y && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    pip install --no-cache-dir -r requirements.txt

# Copy the current directory contents into the container at /app
COPY . /app

# Expose port 5001 to the outside world
EXPOSE 5001

# Run script when the container launches
CMD ["python", "app.py"]
```

Something important to note — we are setting the working directory to '/app' in the second line of the Dockerfile. So any local paths (models, vector or document store), should be prefixed with '/app' in your application code.

Also, when you run the app in the container (on a Mac), it will not be able to access the GPU, see this thread. I've noticed it usually takes about 20 minutes to get a response using the CPU.

Build & run:

```
docker build -t <image-name>:<tag> .
```

```
docker run -p 5001:5001 <image-name>:<tag>
```

Running the container automatically launches the app (see last line of the Dockerfile). You can now access your endpoint at the following URL:

```
http://127.0.0.1:5001/rag_endpoint
```

Call the API:

```
import requests, json

def call_api(query):
    URL = "http://127.0.0.1:5001/rag_endpoint"

    # Headers for the request
    headers = {
        "Content-Type": "application/json"
    }

    # Body for the request.
    body = {"query": query}

    # Making the POST request
    response = requests.post(URL, headers=headers, data=json.dumps(body))

    # Check if the request was successful
    if response.status_code == 200:
        return response.json()
    else:
        return f"Error: {response.status_code}, Message: {response.text}"

# Test
query = "Wall-mounted electric fireplace with realistic LED flames"

result = call_api(query)
print(result)

# result
{'response': {'choices': [{'finish_reason': 'stop', 'index': 0, 'logprobs': None
```

Conclusion

I want to recap on the all the steps required to get to this point, and the workflow to retrofit this for any data / embeddings / LLM.

1. Pass your directory of text files to the `document_chunker` function to create the document store.
2. Choose your embeddings model. Save it locally.
3. Convert document store to vector store. Save both locally.
4. Download LLM from HF hub.

5. Move the files to the app directory (embeddings model, LLM, doc store and vec store JSON files).
6. Build and run Docker container.

Essentially it can be boiled down to this — use the `build` notebook to generate the doc_store and vector_store, and place these in your app.

GitHub here. Thank you for reading!