How to build a Retrieval-Augmented Generation (RAG) system using Llama3, Ollama, DSPy, and Milvus

Zilliz · Follow
5 min read · Apr 28, 2024

By now, you are probably already familiar with the Retrieval-Augmented Generation (RAG system, a framework used in NLP applications. In this article, we aim to guide readers through constructing an RAG system using four key technologies: Llama3, Ollama, DSPy, and Milvus. First, let's understand what they are.

Introducing Llama3, Ollama, DSPy, and Milvus

Llama3 is a language model that generates coherent and contextually appropriate answers. It is built on the latest advancements in NLP technology, offering high accuracy in understanding and responding to complex queries.

Ollama, a reliable hosting service, empowers the deployment and scaling of Llama3. It enables the RAG system to manage increased loads and simultaneous requests with ease, maintaining optimal performance.

DSPy is a flexible programming framework designed to develop and optimize machine learning models, including those used in RAG systems. It provides tools that streamline the model training process, enhance algorithm efficiency, and simplify the implementation of complex machine learning pipelines.

Finally, Milvus is an advanced vector database engineered for efficient similarity search and quick data retrieval. It offers scalable storage solutions and high-speed search capabilities. Integrating Milvus into the RAG system significantly enhances its ability to find and retrieve relevant information quickly.

Setup and Installation

First, create and activate a Python environment suitable for handling our dependencies. You can use virtual environments like venv or conda to isolate and manage your project packages.

Install the necessary Python libraries by running the following command in your terminal. pip install dspy pymilvus openai pandas

Import the installed libraries into your Python script to ensure they are ready for use.

```
import dspy
import milvus
import openai
import os
import pandas as pd
```

To configure API keys for OpenAI, set them in your environment variables. Replace ‘your_openai_api_key’ with the actual OpenAI API key.

```
# Configure the OpenAI and Milvus API keys (you need to replace
'your_key_here' with actual keys)
os.environ['OPENAI_API_KEY'] = 'your_openai_api_key'
```

Milvus Configuration

After setting up the OpenAI API key, the next step is configuring Milvus to manage embeddings within your RAG system. Start initializing the Milvus client to connect to your database using the provided URI. This connection enables your system to interact with the Milvus vector database, which is critical for efficiently storing and retrieving large volumes of vector data. Once connected, you'll need to set up a collection specifically designed for the embeddings. Define the collection with necessary fields for storing embeddings and unique identifiers. Specify the vector dimensions and choose an appropriate index type to optimize the search capabilities within Milvus. Here's how to initialize the Milvus client and set up the collection.

```
from milvus import Milvus, MetricType, IndexType, DataType

# Initialize Milvus client
milvus_client = Milvus(uri='your_milvus_uri')

# Define collection parameters
collection_name = 'ZillizBlogCollection'
collection_params = {
    'fields': [
        {'name': 'text', 'type': DataType.FLOAT_VECTOR, 'params':
{'dim': 768}, 'indexes': [{'index_type': IndexType.IVF_FLAT,
'metric_type': MetricType.L2}]},
        {'name': 'id', 'type': DataType.INT64, 'auto_id': True}
    ]
}

# Create collection if it doesn't exist
if not milvus_client.has_collection(collection_name):
    milvus_client.create_collection(collection_name,
collection_params)
```

For data ingestion, preprocess your text data and convert it into embeddings before inserting them into the collection. This process typically involves using a model to generate embeddings from the text, as demonstrated in the function below.

```
# Function to create Milvus vectors for the blog posts
def create_milvus_vectors(data_frame, text_column='text'):
    # Assuming the use of an OpenAI model to generate embeddings
    embeddings = openai.Embedding.create(
        input=data_frame[text_column].tolist(),
        model="text-embedding-ada-002"
    )
    return embeddings['data']
```

MilvusRM Integration

MilvusRM is a component used to integrate retrieval mechanisms with Milvus collections efficiently. Initialize MilvusRM by specifying the collection name and connection URI, as shown in the following code.

```
from dspy.retrieve.milvus_rm import MilvusRM

# Initialize the MilvusRM retriever
milvus_retriever = MilvusRM(
    collection_name=collection_name,
    uri='your_milvus_uri',
    k=5
)
```

Llama3 and Ollama Setup

To connect your application to Llama3 through the Ollama hosting service, configure the necessary settings such as the model version and the token limits. This setup enables your system to access Llama3's capabilities for generating responses. Here is how you can establish this connection.

```
# Connect to Llama3 hosted with Ollama
llama3_ollama = dspy.OllamaLocal(
    model="llama3:8b-instruct-q5_1",
    max_tokens=4000,
    timeout_s=480
)
```

After configuring the connection, conduct a simple test to ensure that the connection to Llama3 is operational. This verification step is important to check if the system can effectively communicate with Llama3 and receive responses.

```
# Test connection
test_query = "What is the latest in AI?"
test_response = llama3_ollama(test_query)
print("Test Llama3 response:", test_response)
```

Building the RAG System

Define a Python class for the RAG system to integrate both the retrieval capabilities of MilvusRM and the generative power of Llama3. This class structure allows the system to efficiently handle queries by retrieving relevant information and generating responses.

```
class RAG(dspy.Module):
    def __init__(self, retriever, generator, k=5):
        super().__init__()
        self.retrieve = dspy.Retrieve(k=k,
retriever_model=retriever)
        self.generate_answer = dspy.Predict(generator)

    def forward(self, question):
        context = self.retrieve(question).passages
        pred = self.generate_answer(context=context,
question=question).answer
        return dspy.Prediction(context=context, answer=pred,
question=question)
```

Instantiate the RAG system with the previously initialized components, setting it up to process and respond to queries.

```
# Instantiate the RAG system
rag_system = RAG(retriever=milvus_retriever,
generator=llama3_ollama)
```

MIPRO Optimization

Define the optimization metric to evaluate the responses generated by the RAG system. This metric should compare the generated responses to ground truth answers to assess accuracy and relevance.

```
# Define the evaluation metric
def metric_function(gold, pred):
```

Configure and initiate the MIPRO optimizer to refine the prompts used by Llama3 based on the retrieved documents. This optimization process aims to enhance the relevance and quality of the generated responses.

```
from dspy.teleprompt import MIPRO

# Assume 'trainset' is a DataFrame loaded with question-answer pairs
trainset = pd.DataFrame({
    'question': ['What is AI?', 'Explain machine learning'],
    'gold_answer': ['Artificial intelligence is the simulation of
human intelligence in machines.', 'Machine learning is a subset of
AI that allows systems to learn and improve from experience.']
})

# Configure and run MIPRO optimizer
mipro_optimizer = MIPRO(
    prompt_model=llama3_ollama,
    task_model=llama3_ollama,
    metric=metric_function,
    num_candidates=3
)
```

Evaluation and Testing

Testing the RAG system involves feeding it with predetermined queries and analyzing the responses it generates. We assess the accuracy of the system's answers by comparing them against known correct answers, often referred to as "gold" answers. This comparison allows us to determine how frequently the system provides correct information.

In addition to accuracy, we also evaluate the relevance of the generated responses. Relevance measures how well the system's answers align with the intent of the queries. This assessment helps ensure that the system delivers information that is correct, contextually appropriate, and useful to the user.

Conclusion

This tutorial taught us the significance of integrating retrieval and generation components for effective natural language processing. We've gained insights into configuring and optimizing each component and understanding their roles in handling queries and generating responses.

We can consider expanding the dataset to improve the diversity and quality of responses to enhance our system. Moreover, integrating more sophisticated evaluation metrics would provide deeper insights into system performance and relevance.

References

Milvus Documentation

DSPy Documentation

Llama3 Cookbook

Engineering

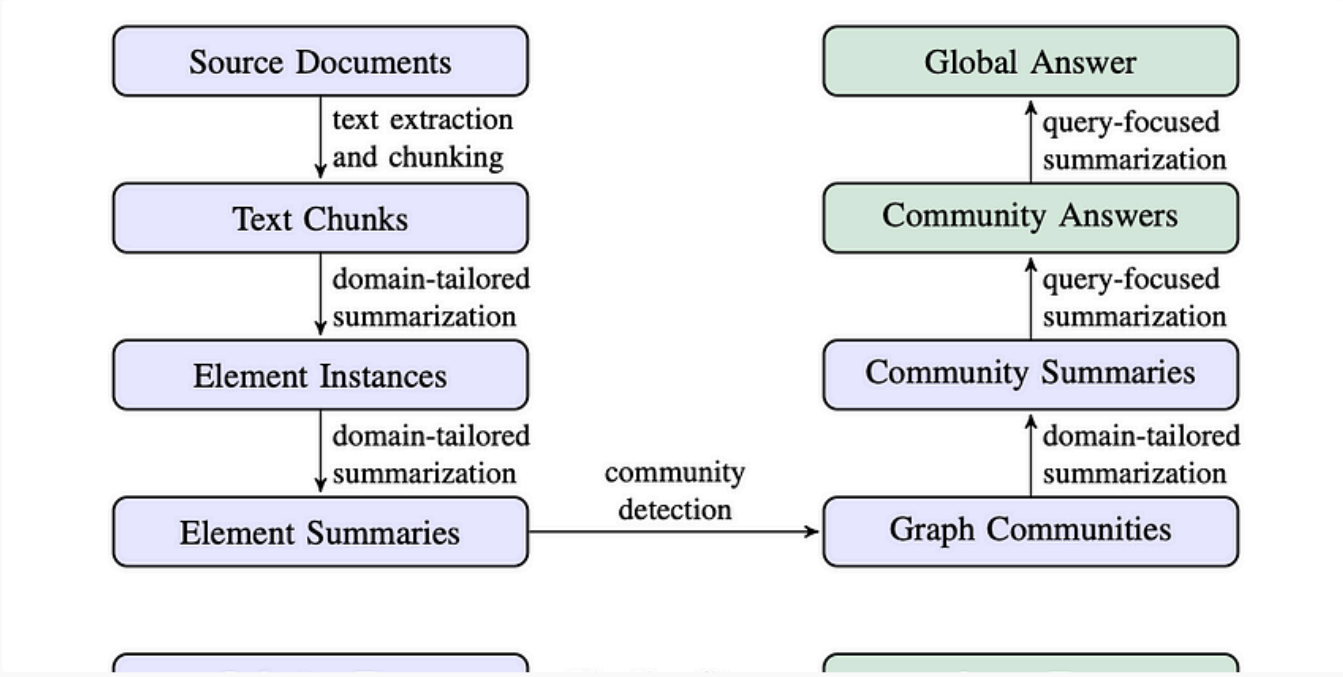
Follow

Written by Zilliz

238 Followers

Building the #VectorDatabase for enterprise-grade AI.

More from Zilliz



Zilliz

GraphRAG Explained: Enhancing RAG with Knowledge Graphs

Introduction to RAG and Its Challenges

Aug 7 29 1

Zilliz

Function Calling with Ollama, Llama 3.1 and Milvus

Jul 30 37

Zilliz

Vectorizing JSON Data with Milvus for Similarity Search

May 22 70 2

Zilliz

An Ultimate Guide to Vectorizing and Querying Structured Data

May 29 4

See all from Zilliz

Recommended from Medium

Ming

Comparing LangChain and LlamaIndex with 4 tasks

LangChain v.s. LlamaIndex — How do they compare? Show me the code!

yment	Cloud Deploy
	☑ (Self-hosted)
	☑ (Managed)
	☑ (Self-hosted)
	☑ (Self-hosted)
	☑ (Self-hosted)
	☒
	☑ (Self-hosted)

Plaban Nayak in The AI Forum

Which Vector Database Should You Use? Choosing the Best One for Your Needs

Introduction

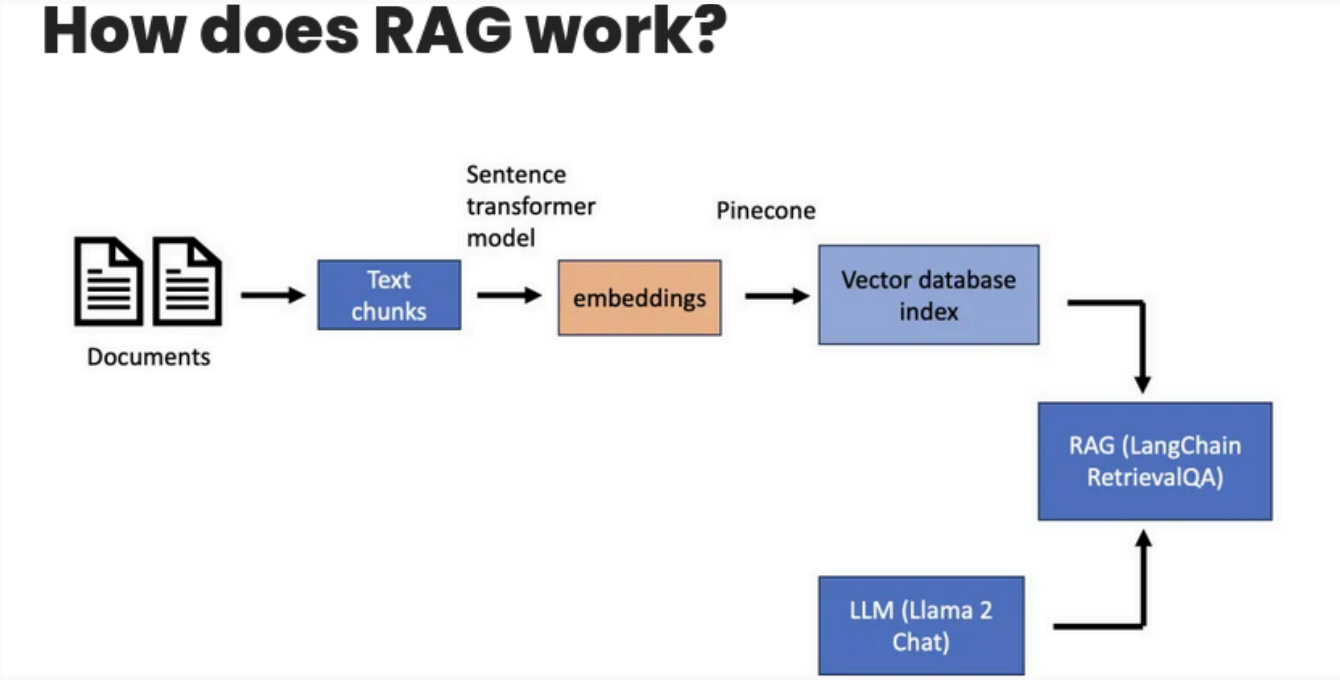
Apr 19 522 8

Isaiah Bjorklund

Creating a RAG Chatbot with Llama 3.1: A Step-by-Step Guide

In this blog post, we'll explore how to create a Retrieval-Augmented Generation (RAG) chatbot using Llama 3.1, focusing on both the 405...

Jul 31 55

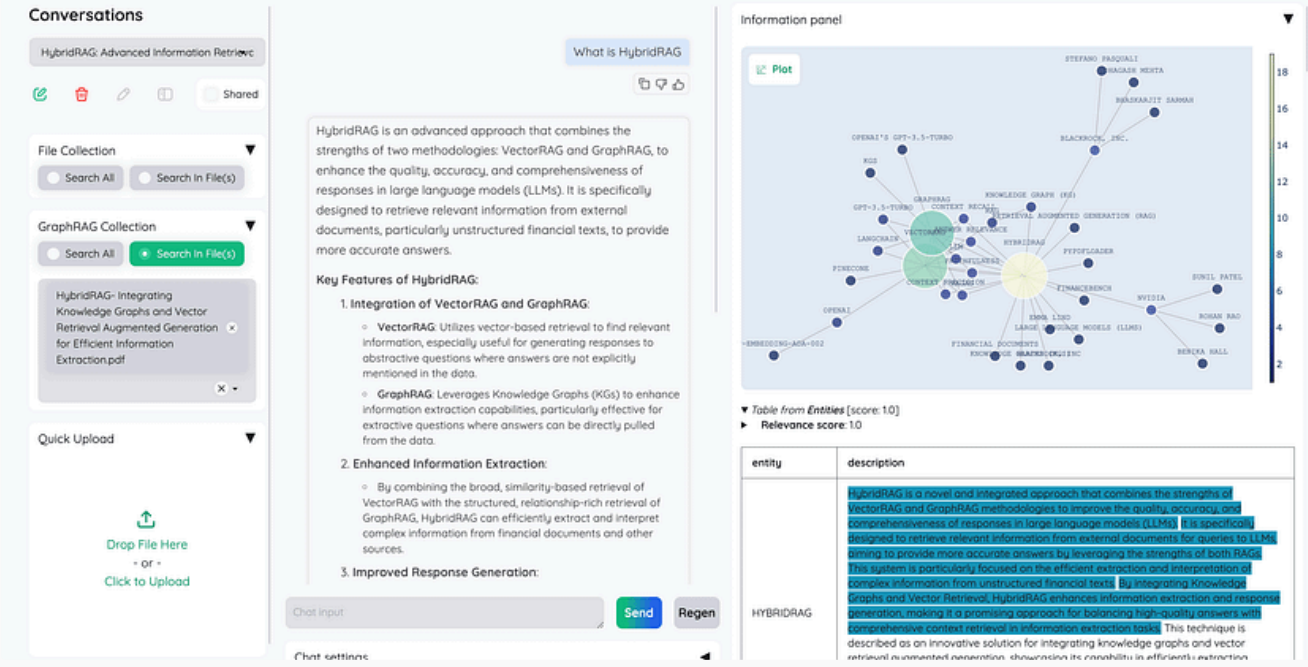


mobin shaterian in Stackademic

Build a RAG with llama

The article describes Retrieval-Augmented Generation (RAG) which helps large language models answer questions based on unseen documents...

Jun 6



Bhavik Jikadara

Kotaemon: Open-source GraphRAG UI On Local Machine

Kotaemon is an open-source, clean, and customizable Retrieval-Augmented Generation (RAG) User Interface (UI) designed for both end-users...

Sep 1 73

See more recommendations