Explore Developer Center's New Chatbot! MongoDB AI Chatbot can be accessed at the top of your navigation to answer all your MongoDB questions.

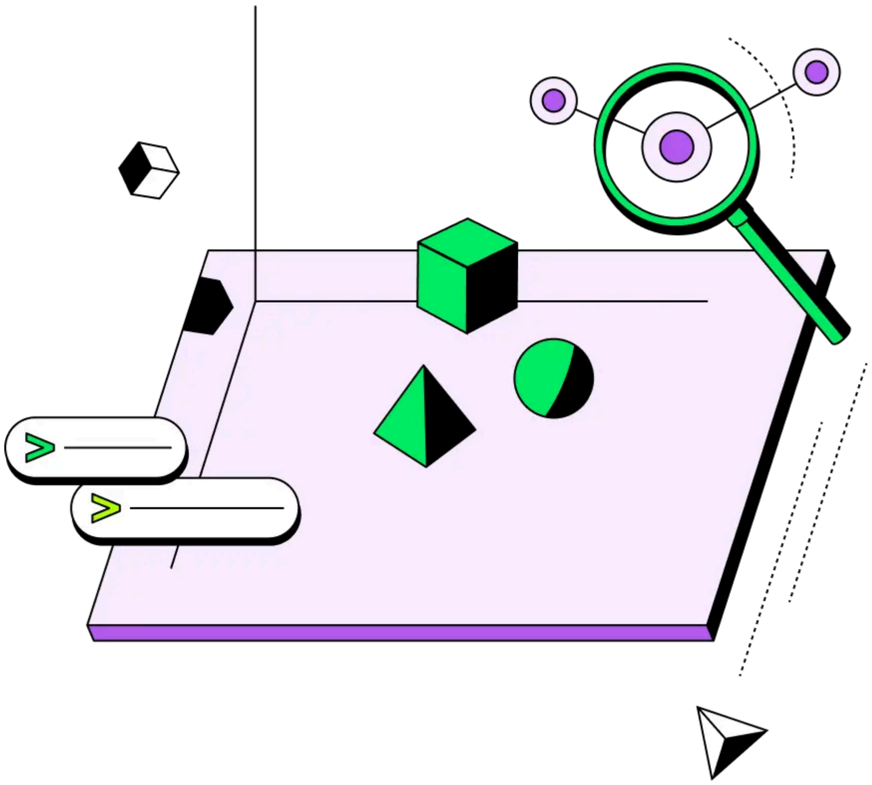
VIEW MONGO AI

Quickstart Guide to RAG Application Using LangChain and LlamaIndex

Kushagra Kesav

10 min read • Published Jun 05, 2024 • Updated Sep 18, 2024

AI Atlas Vector Search Python



Rate this quickstart

Introduction

While large language models such as GPT-4 are very good at generating content and logical reasoning, they face limitations when it comes to accessing and retrieving precise facts, or contextually relevant information. One popular approach to address this involves implementing a retrieval-augmented generation (RAG) system. This system integrates the language model with a vector database such as MongoDB Atlas Vector Search to form a comprehensive AI framework capable of orchestrating interactions between these components.

As the demand for efficient information retrieval continues to increase, understanding the syntax and capabilities of various frameworks becomes important. In this article, we will see the basics of vector search in simple terms. We'll look at LangChain, LlamaIndex, and PyMongo, showing you step-by-step how to use their methods for semantic search. By delving into these frameworks, we aim to understand their respective syntax, and showcasing how they stack up with MongoDB vector search.

Key takeaways

- Overview of vector search and RAG
- Extract saved data from MongoDB, convert to embeddings, store back, and run semantic search for contextual information
- Build an end-to-end RAG system using MongoDB alongside AI frameworks like LlamaIndex and LangChain, then contrast their syntax

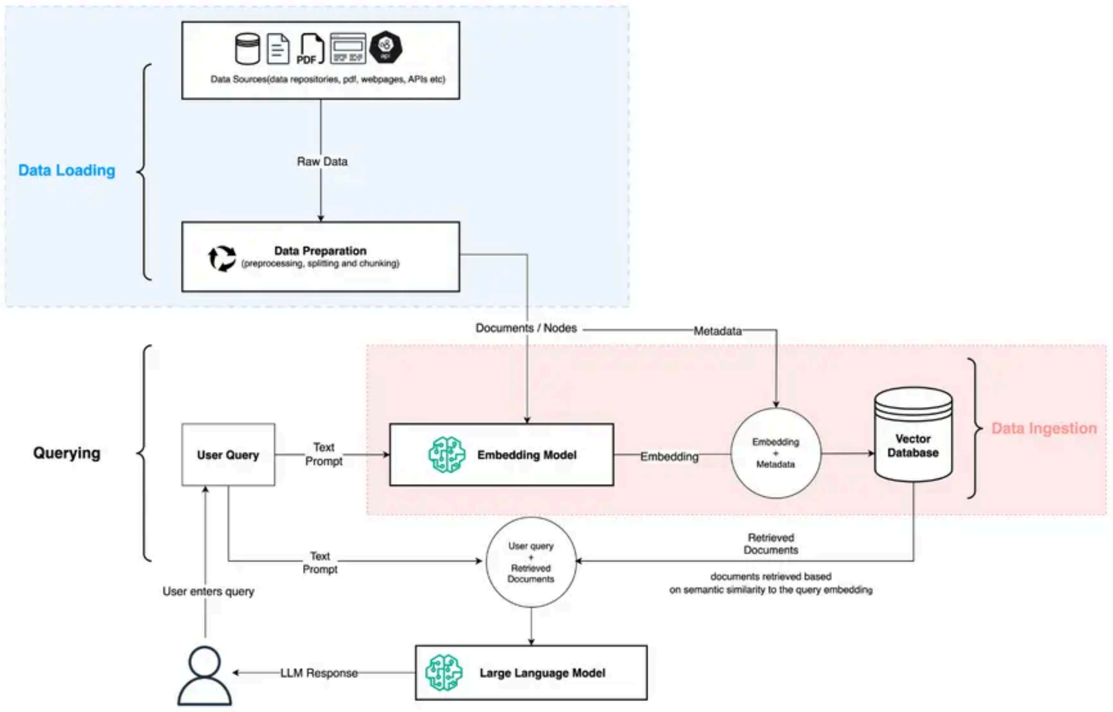
What is vector search?

A vector database is a type of data storage solution that manages and searches large amounts of high-dimensional numerical data (also known as vectorised data). This data is represented as vectors, which are created using an embedding model that takes input --- such as images, audio, video, and text --- and converts them into vectors. These vectors are stored in a database and can be queried using semantic search methods for fast and accurate retrieval of similar data objects.

When multiple vector representations are mapped into a high-dimensional space, the distance between these vectors in that space reflects the similarity between them. This is because the vectors capture the context and meaning of the original data, allowing for a refined understanding of their relationships. Vector databases are designed to quickly calculate the distance between vectors, enabling efficient retrieval of information based on a query vector or semantic search. In contrast, traditional databases rely on keyword matching to retrieve information, which is a fundamentally different approach.

What is RAG?

Retrieval-augmented generation (RAG) is a approach that uses information retrieval and generative AI to deliver accurate and relevant responses to user queries by gathering semantically related data to enrich user queries with extra context, processed as input for LLMs.



Getting started with LangChain framework

Step 1: Installing required libraries for LangChain framework

This section guides you through the installation process of the essential libraries needed to implement the RAG application with LangChain. Here is the list of required libraries:

- langchain: The Python toolkit for LangChain
- langchain-mongodb: A Python package to use MongoDB as a vector store, semantic cache, chat history store, etc., in LangChain
- pymongo: The Python toolkit for MongoDB
- openAI: A Python library for the OpenAI API
- nest-asyncio: A utility library for running an embedded asyncio event loop

Step 2: Data cleaning and loading

Here we are going to use the `embedding_movies` sample collection from `sample_mflix`, and we will do some cleaning before using it. Please run the following command to remove the documents that do not contain the `plot` field, so we will not run into errors:

```
db.embedded_movies.deleteMany({ plot: { $exists: false } } )

{ acknowledged: true, deletedCount: 80 }
```

Now, we will create a Jupyter notebook and write down the following code, which will extract the data from the specified collection.

```
import os
from dotenv import load_dotenv
from langchain_community.document_loaders.mongodb import MongodbLoader
import nest_asyncio

nest_asyncio.apply()
load_dotenv()
loader = MongodbLoader(
    connection_string=os.environ['MONGODB_URI'],
    db_name=os.environ['MONGODB_DB'],
    collection_name=os.environ['MONGODB_COLL'],
    filter_criteria={},
    field_names=["title", "plot"]
)
docs = loader.load()

print(len(docs))
docs[0]
```

Here we see some environment variables used in the code. So, we'll make a .env file and put these variables in it.

```
# MongoDB URI
# Replace <username> and <password> with your MongoDB credentials
MONGODB_URI='mongodb+srv://<username>:<password>@kushagracluster.abcedf.mongod

# Name of the MongoDB database to use
MONGODB_DB='sample_mflix'

# Name of the MongoDB collection to use within the specified database
MONGODB_COLL='embedded_movies'

# API key for OpenAI.
OPENAI_API_KEY="<OPENAI_API_KEY>"
```

```

# Name of the index to use for vector storage
MONGODB_VECTOR_INDEX='vector_index'

# Name of the collection in MongoDB where embedding data is stored
MONGODB_VECTOR_COLL_LANGCHAIN='langchain_coll'
```

In the above code, we are using the `MongoLoader` class to retrieve documents from MongoDB. The dotenv library is used to load environment variables from a .env file, and the `nest_asyncio` library enables asyncio event loop nesting.

Additionally, we also initialize the MongoLoader class with parameters such as the MongoDB connection string, database name, collection name, etc. to get data from each document.

The `load()` method of the `MongoLoader` instance is invoked to fetch documents from MongoDB based on the specified parameters, and further, we print it.

Step 3: Create embeddings with OpenAI

Now, we will add another code cell in the Jupyter notebook and run the following code to create the embeddings with OpenAI.

```
from pymongo import MongoClient
from langchain.vectorstores import MongoDBAtlasVectorSearch
from langchain_community.embeddings import OpenAIEmbeddings
from langchain.llms import OpenAI
from langchain_community.chat_models import ChatOpenAI
from langchain.prompts import ChatPromptTemplate
from langchain.chains import LLMChain

client = MongoClient(os.environ['MONGODB_URI'], appname="devrel.content.langch
collection = client.get_database(os.environ['MONGODB_DB']).get_collection(os.e

vector_search = MongoDBAtlasVectorSearch.from_documents(
  documents=docs,
  embedding=OpenAIEmbeddings(openai_api_key=os.environ['OPENAI_API_KEY']),
  collection=collection,
  index_name=os.environ['MONGODB_VECTOR_INDEX'])
```

This code segment performs a couple of tasks related to setting up a search system using MongoDB Vector Search, LangChain, and OpenAI embeddings.

First, we initialize a MongoDB client using `client = MongoClient(os.environ['MONGODB_URI'])` and use the client instance to access a specific collection within the MongoDB database. This collection will hold the embedding data along with the text.

Here is the sample document:

```
{
 "_id": {
   "$oid": "66375d1ff77b69f7b5f0c7ec"
 },
 "text": "The Black Pirate Seeking revenge, an athletic young man joins the pi
 "embedding": [
   -0.003676240383024406,
   -0.02721614428071944,
   ...
   0.02543453162153851,
]
],
 "database": "sample_mflix",
 "collection": "embedded_movies"
}
```

Further, we set up vector search by initializing a vector search object using LangChain's MongoDBAtlasVectorSearch class.

Then, we pass the 'docs' variable, containing documents fetched from MongoDB earlier, to be used for setting up the vector search. This specific line of code, `embedding=OpenAIEmbeddings(openai_api_key=os.environ['OPENAI_API_KEY'])`, uses the OpenAI API key and creates an embedding. It then passes the collection instance to the 'collection' parameter where vector embeddings will be stored.

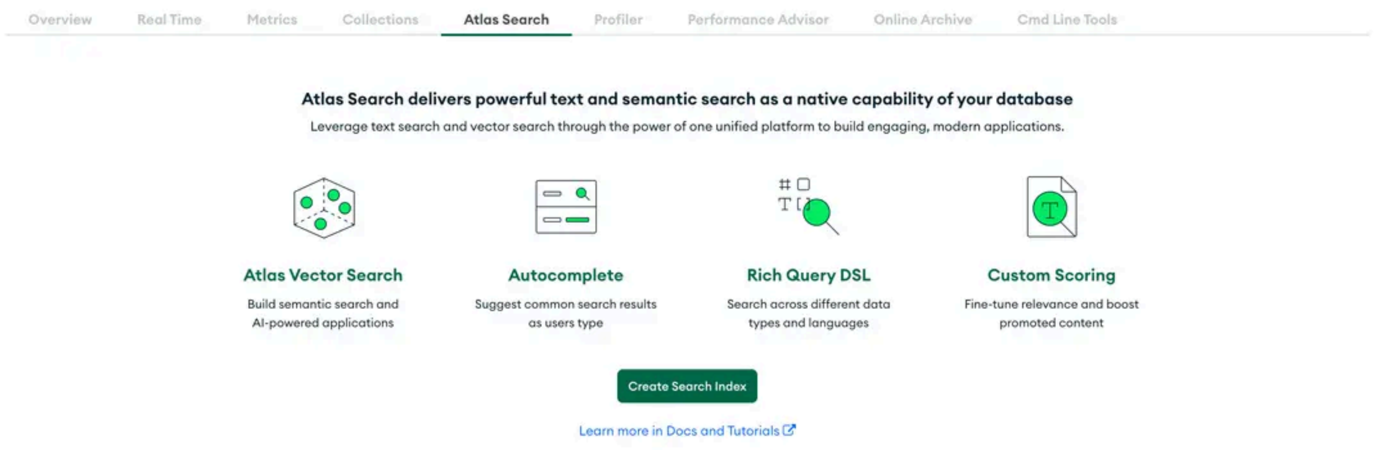
We also specify the name of the vector index (here, vector_index) within the collection which will be used for the semantic search.

Overall, this code part handles the connections to a MongoDB instance and sets up a vector search system using LangChain, with vector data stored in MongoDB and embeddings generated by OpenAI.

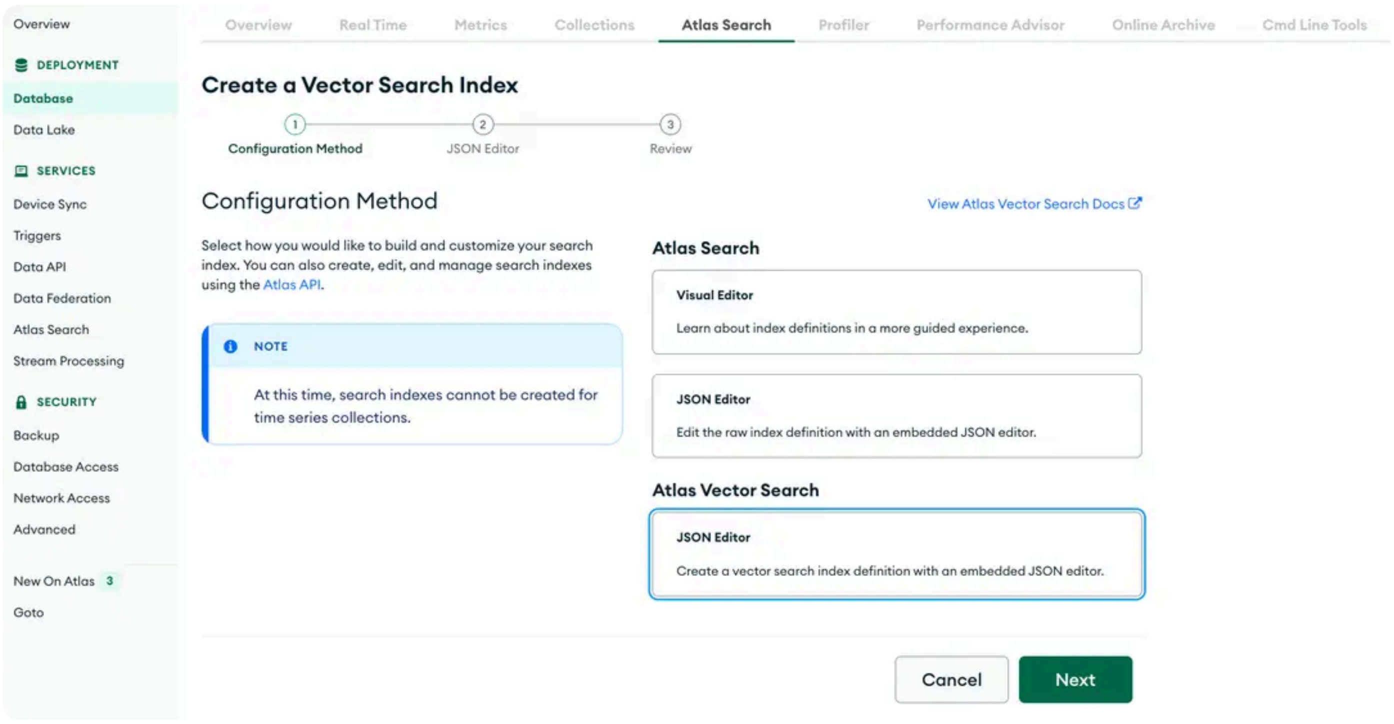
Refer to OpenAI's FAQs to learn how you can get your OPENAI_API_KEY.

Now, let's move to our MongoDB Atlas UI and create a vector search index on our 'langchain_coll' collection.

To do that, visit cloud.mongodb.com and select the Atlas Search tab option on the navigation pane to create an Atlas Vector Search index. Click the Create Search Index button to create an Atlas Vector Search index.


Overview
Real Time
Metrics
Collections
Atlas Search
Profiler
Performance Advisor
Online Archive
Cmd Line Tools
Atlas Search delivers powerful text and semantic search as a native capability of your database
Leverage text search and vector search through the power of one unified platform to build engaging, modern applications.
Atlas Vector Search
Build semantic search and AI-powered applications
Autocomplete
Suggest common search results as users type
Rich Query DSL
Search across different data types and languages
Custom Scoring
Fine-tune relevance and boost promoted content
Create Search Index
Learn more in Docs and Tutorials

On the page to create a Vector Search index, select the Atlas Vector Search option that enables the creation of a vector search index by defining the index using JSON.



Create a Vector Search Index

Configuration Method

Select how you would like to build and customize your search index. You can also create, edit, and manage search indexes using the Atlas API.

NOTE

At this time, search indexes cannot be created for time series collections.

Atlas Search

Visual Editor

Learn about index definitions in a more guided experience.

JSON Editor

Edit the raw index definition with an embedded JSON editor.

Atlas Vector Search

JSON Editor

Create a vector search index definition with an embedded JSON editor.

To complete the creation of the index, select the database and collection for which the index should be created. In this case, it is the sample_mflix database and the langchain_coll collection. The JSON entered into the JSON editor should look like following:

Note: Please make sure your index name is vector_index, as you have set it in your .env file.

```
{
 "fields": [
   {
     "numDimensions": 1536,
     "path": "embedding",
     "similarity": "cosine",
     "type": "vector"
   }
 ]
}
```

Here, make sure you are creating a vector index with the same name you have passed in your *.env*. In this case, it is vector_index.

Step 4: Perform vector search on user queries

Up to this point, we have successfully done the following:

- Loaded data from our MongoDB database
- Provided each document with embeddings using the OpenAI embedding model
- Configured a MongoDB database for storing vector embeddings
- Established a connection to this database from our development environment
- Created a vector search index for optimized querying of vector embeddings

Now, head back to VS Code and write the following code which will put the vector search to work:

```
# perform a similarity search on the ingested documents
prompt='What is the best horror movie to watch?'
docs_with_score = vector_search.similarity_search_with_score(query=prompt,k=1)

llm = ChatOpenAI(openai_api_key=os.environ['OPENAI_API_KEY'])

prompt_template = ChatPromptTemplate.from_messages([
    ("system", "You are a movie recommendation engine which posts a concise an
    ("user", "List of movies: {input}")
])

# Create an LLMChain
chain = LLMChain(
    llm=llm,
    prompt=prompt_template
)

# Prepare the input for the chat model
input_docs = "\n".join([doc.page_content for doc, _ in docs_with_score])

# Invoke the chain with the input documents
response = chain.invoke({"input": input_docs})
print(response['text'])
```

This step involves the process to set up a similarity search on ingested documents to find the best horror movie recommendations and then use an AI language model to generate a concise summary of these recommendations. So now, we will pass the prompt containing the question. Using the `similarity_search_with_score` method from the `vector_search` object, the code searches for the document that most closely matches this query, returning the top result.

Next, the language model `ChatOpenAI` is initialized with an API key. The prompt template is then created using `ChatPromptTemplate`, which sets the context for the language model as a movie recommendation engine. This template includes a system message defining the AI's role and a user message template with a placeholder for the output.

Further, an `LLMChain` object is then created, combining the language model and the prompt template. The content of the retrieved documents is extracted and concatenated into a single string, which serves as input for the language model. And finally, we print the output text to the console.

You can execute the whole code and see the semantic search in action, where it will return the best horror movie to watch. That's what we call magic.

```
"Don't Be Afraid of the Dark" is a horror film about a young girl who discovers mysterious creatures after moving in with her father and his
girlfriend. The movie explores the girl's terrifying experiences as she tries to uncover the secrets of her new home.
```

Getting started with LlamaIndex framework

Step 1: Installing required libraries for LlamaIndex framework

This section guides you through the installation process of the essential libraries needed to implement the RAG application with LlamaIndex. Here is the list of required libraries:

- openAI: A python library for OpenAI API
- llama_index: Python package to use MongoDB as a vector store, semantic cache, chat history store, etc., in llama_index
- pymongo: Python toolkit for MongoDB

Step 2: Data cleaning and loading

Since we've already cleaned the data in the previous process, there's no need to repeat this step. However, if you're starting with this step, please refer to STEP 2 of the LangChain method above. The process for data cleansing remains the same.

Moving forward, we will use the `SimpleMongoReader` function offered by the Llamaindex to load the data:

```python
import os
from llama_index.readers.mongodb import SimpleMongoReader

reader = SimpleMongoReader(uri=os.environ["MONGODB_URI"])
documents = reader.load_data(
    os.environ["MONGODB_DB"],
    os.environ["MONGODB_COLL"],
    field_names=["title","plot"], # these is a list of the top-level fields in
    query_dict={} # this is a mongo query dict that will filter your data if y
)
print(documents[0])
```

This part helps us connect to a MongoDB database and retrieves documents from a specified collection. It uses a `SimpleMongoReader` class provided by the `llama_index` package to handle the interaction with MongoDB. Apart from basic parameters, it additionally specifies which fields from the documents should be indexed (in this case, "title" and "plot"). We can also specify a query dictionary that can be used to filter the data being indexed. Since it's `empty {}`, it indicates no specific filtering is applied.

Step 3: Create embeddings with OpenAI

```
import pymongo
import openai
from llama_index.core import VectorStoreIndex,StorageContext
from llama_index.vector_stores.mongodb import MongoDBAtlasVectorSearch

# Create a new client and connect to the server
client = pymongo.MongoClient(os.environ["MONGODB_URI"], appname="devrel.conten

store = MongoDBAtlasVectorSearch(
    client,
    db_name=os.environ['MONGODB_DB'],
    collection_name=os.environ['MONGODB_VECTOR_COLL_LLAMAINDEX'], # this is wh
    index_name=os.environ['MONGODB_VECTOR_INDEX']
)

storage_context = StorageContext.from_defaults(vector_store=store)
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context,
    show_progress=True, # this will show you a progress bar as the embeddings
)
```

In this step, we import libraries such as pymongo, openAI, and a few more from LlamaIndex to work with data stored in a MongoDB database and perform vector indexing and searching operations.

We then establish a MongoDB connection using `client = pymongo.MongoClient(os.environ["MONGODB_URI"])`, passing the environment variable `"MONGODB_URI"`. Here, the `os.environ` function retrieves the value of the specified environment variable.

Following that, we create an Atlas vector store using an object of type `MongoDBAtlasVectorSearch`. It takes the database name, collection name, and index name as parameters, and stores the embeddings in the specified MongoDB collection.

Note: Don't forget to create a vector search index as you did earlier by visiting cloud.mongodb.com, selecting the Atlas Search tab option on the navigation pane, and selecting the *sample_mflix* database and the *llamaindex_coll* collection.

The JSON entered into the JSON editor should look similar to the following:

```json
{
 "fields": [
   {
     "numDimensions": 1536,
     "path": "embedding",
     "similarity": "cosine",
     "type": "vector"
   }
 ]
}
```

Subsequently, we create a vector index using `storage_context = StorageContext.from_defaults(vector_store=store)`, which generates a storage context object using default settings and associates it with the specified vector store. Then, it creates a vector index on the documents loaded in our database. The `show_progress` parameter determines whether to display a progress bar during the indexing process or not.

Overall, this step handles setting up a connection to a MongoDB database using PyMongo and creating a vector store using a custom implementation for MongoDB Atlas. Further, it generates a vector index from a set of documents and embeds them into vector representations for efficient searching and retrieval.

Step 4: Perform vector search on user queries

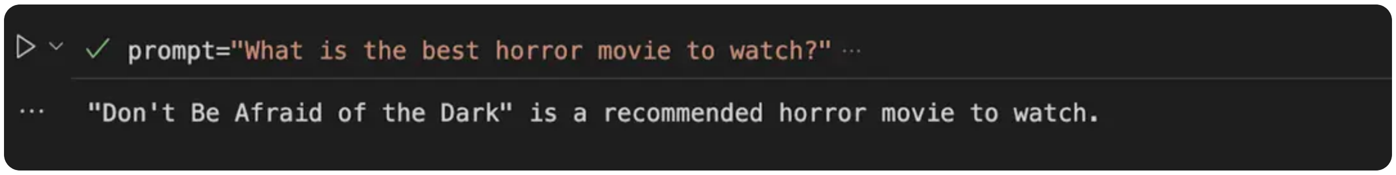
Until now, we have successfully loaded the data in our collections along with the generated embedding for it using OpenAI.

Now, this step combines all the activities from the previous step to provide the functionality of conducting vector search on stored documents based on embedded user queries. This step takes the prompt and passes it as a query. The `.as_query_engine(similarity_top_k=1)` part determines that it's configuring the index to operate as a query engine and specifying that it will return only the most similar result.

```
prompt="What is the best horror movie to watch?"
response = index.as_query_engine(similarity_top_k=1).query(prompt)

print(response)
```

And you will see the following output, after executing the whole code snippet, where it returns the best horror movie to watch:



```
prompt="What is the best horror movie to watch?" ···
"Don't Be Afraid of the Dark" is a recommended horror movie to watch.
```

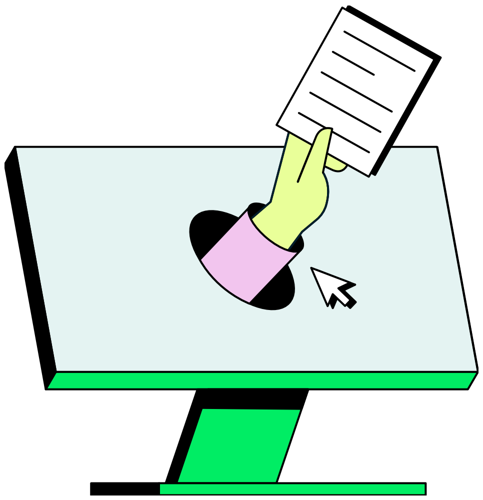
If you are looking to use a pure aggregation pipeline ($vectorsearch) for semantic search without any LLM frameworks, please refer to our tutorial. You will find step-by-step instructions on how to build a RAG system using PyMongo, OpenAI, and MongoDB.

Summing it all up

In this tutorial, we walked through the process of creating a RAG application with MongoDB using two different frameworks. I showed you how to connect your MongoDB database to LangChain and LlamaIndex separately, load the data, create embeddings, store them back to the MongoDB collection, and then execute a semantic search using MongoDB Atlas vector search capabilities. View the GitHub repo for the implementation code.

If you have any questions or feedback, reach out through the MongoDB Community forums and let us know what you build using MongoDB Atlas Vector Search.

Top Comments in Forums



There are no comments on this article yet.

Start the Conversation

Rate this quickstart

Related

TUTORIAL

Reinventing Multi-modal Search With MongoDB and Anyscale

Sep 18, 2024 | 20 min read

TUTORIAL

Java Faceted Full-Text Search API using MongoDB Atlas Search

Jan 17, 2025 | 18 min read

TUTORIAL

How to Use Cohere Embeddings and Rerank Modules With MongoDB Atlas

Mar 13, 2025 | 10 min read

TUTORIAL

Securely Hosting a Lambda Application With a Microservice Architecture and MongoDB Atlas

Sep 19, 2024 | 4 min read

Request a Quickstart

English

About

Careers

Investor Relations

Support

Contact Us

Customer Portal

Legal

Atlas Status

GitHub

Customer Support

Security Information

Manage Cookies

Trust Center

Connect with Us

Deployment Options

MongoDB Atlas

Enterprise Advanced

Community Edition

Data Basics

Vector Databases

NoSQL Databases

Document Databases

RAG Database

ACID Transactions

MERN Stack

MEAN Stack

© 2024 MongoDB, Inc.