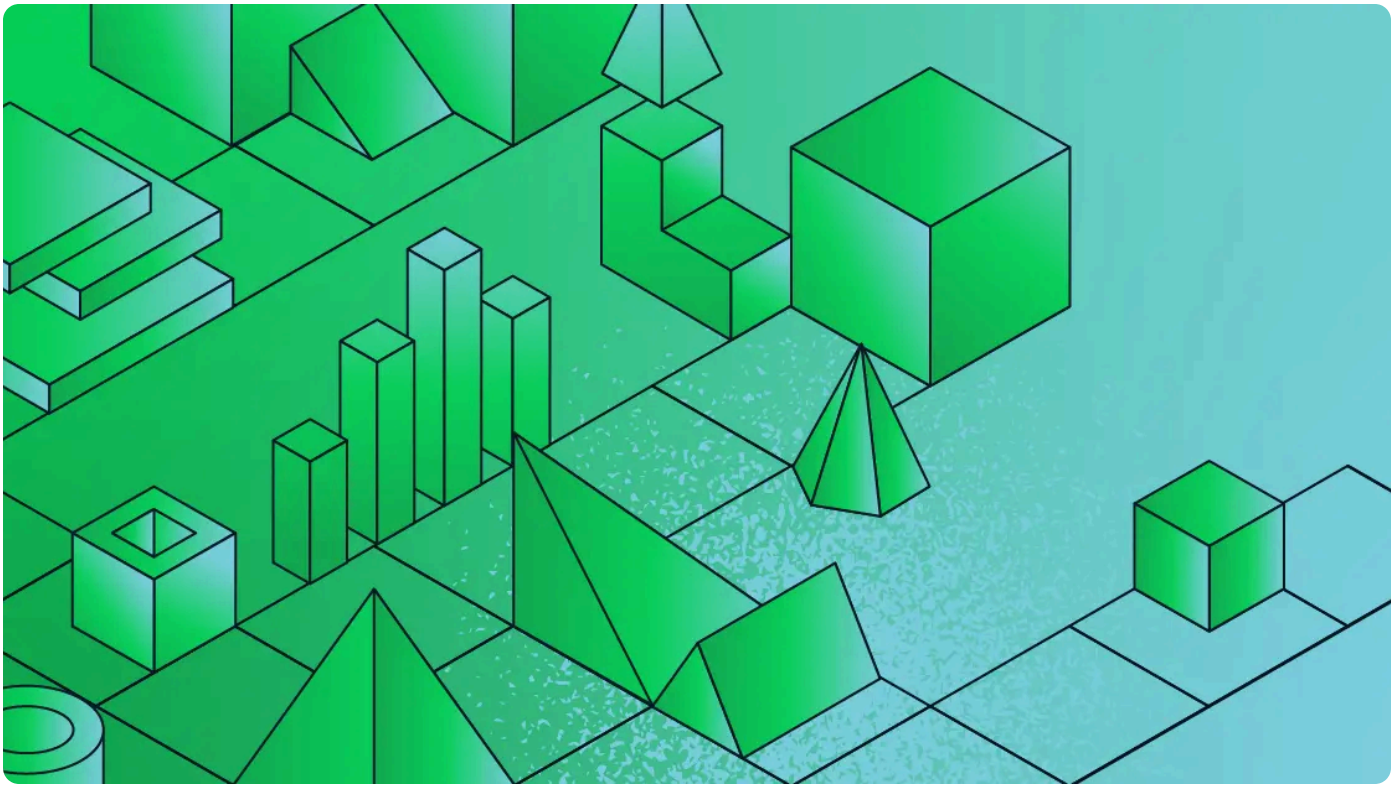
How to Evaluate Your LLM Application

Apoorva Joshi

20 min read • Published Jun 24, 2024 • Updated Jun 24, 2024

AI Python Atlas



If you have ever deployed machine learning models in production, you know that evaluation is an important part of the process. Evaluation is how you pick the right model for your use case, ensure that your model's performance translates from prototype to production, and catch performance regressions. While evaluating Generative AI applications (also referred to as LLM applications) might look a little different, the same tenets for why we should evaluate these models apply.

In this tutorial, we will break down how to evaluate LLM applications, with the example of a Retrieval Augmented Generation (RAG) application. Specifically, we will cover the following:

- Challenges with evaluating LLM applications
- Defining metrics to evaluate LLM applications
- How to evaluate a RAG application

> Before we begin, it is important to distinguish LLM model evaluation from LLM application evaluation. Evaluating LLM models involves measuring the performance of a given model across different tasks, whereas LLM application evaluation is about evaluating different components of an LLM application such as prompts, retrievers, etc., and the system as a whole. In this tutorial, we will focus on evaluating LLM applications.

Challenges with evaluating LLM applications

The reason we don't hear as much about evaluating LLM applications is that it is currently challenging and time-consuming. Conventional machine learning models such as regression and classification have a mathematically well-defined set of metrics such as mean squared error (MSE), precision, and recall for

evaluation. In many cases, ground truth is also readily available for evaluation. However, this is not the case with LLM applications.

LLM applications today are being used for complex tasks such as summarization, long-form question-answering, and code generation. Conventional metrics such as precision and accuracy in their original form don't apply in these scenarios, since the output from these tasks is not a simple binary prediction or a floating point value to calculate true/false positives or residuals from. Metrics such as faithfulness and relevance that are more applicable to these tasks are emerging but hard to quantify definitively. The probabilistic nature of LLMs also makes evaluation challenging – simple formatting changes at the prompt level, such as adding new lines or bullet points, can have a significant impact on model outputs. And finally, ground truth is hard to come by and is time-consuming to create manually.

How to evaluate LLM applications

While there is no prescribed way to evaluate LLM applications today, some guiding principles are emerging.

Whether it's choosing embedding models or evaluating LLM applications, focus on your specific task. This is especially applicable while choosing parameters for evaluation. Here are a few examples:

Task	Evaluation parameters
Content moderation	Recall and precision on toxicity and bias
Query generation	Correct output syntax and attributes, extracts the right information upon execution
Dialogue (chatbots, summarization, Q&A)	Faithfulness, relevance

Tasks like content moderation and query generation are more straightforward since they have definite expected answers. However, for open-ended tasks involving dialogue, the best we can do is to check for factual consistency (faithfulness) and relevance of the answer to the user question. Currently, a common approach for performing such evaluations is using strong LLMs. While this technique may be subject to some of the challenges we face with LLMs today, such as hallucinations and biases, it scales better than human evaluation. When choosing an evaluator LLM, the Chatbot Arena Leaderboard is a good resource since it is a crowdsourced list of the best-performing LLMs ranked by human preference.

Once you have figured out the parameters for evaluation, you need an evaluation dataset. It is worth spending the time and effort to handcraft a small dataset (even 50 samples is a good start!) consisting of the most common questions users might ask your application, some edge (read: complex) cases, as well as questions that help assess the response of your system to malicious and/or inappropriate inputs. You can evaluate the system separately on each of these question sets to get a more granular understanding of the strengths and weaknesses of your system. In addition to curating a dataset of questions, you may also want to write out ground truth answers to the questions. While these are especially important for tasks like query generation that have a definitive right or wrong answer, they can also be useful for grounding LLMs when using them as a judge for evaluation.

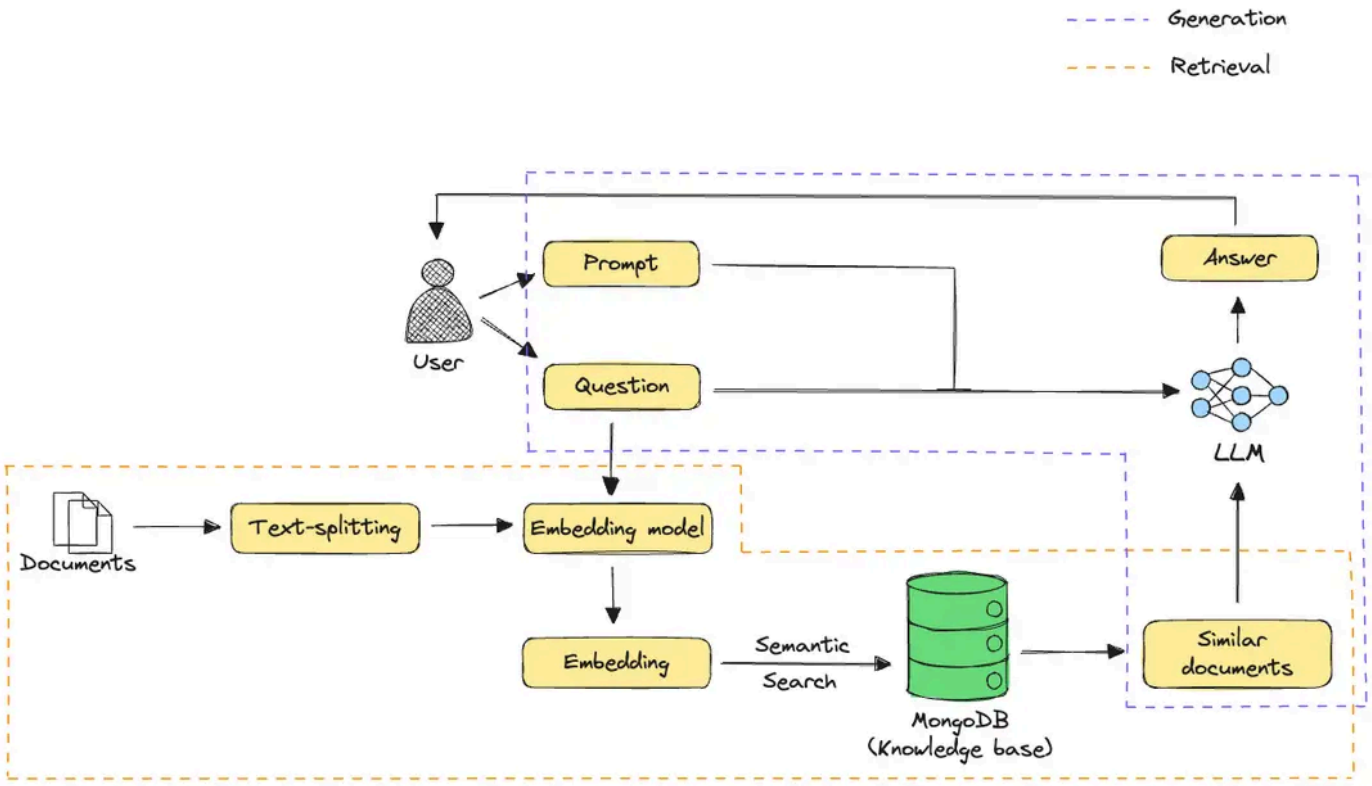
As with any software, you will want to evaluate each component separately and the system as a whole. In RAG systems, for example, you will want to evaluate the retrieval and generation to ensure that you are retrieving the right context and generating suitable answers, whereas in tool-calling agents, you will want to validate the intermediate responses from each of the tools. You will also want to evaluate the overall system for correctness, typically done by comparing the final answer to the ground truth answer.

Finally, think about how you will collect feedback from your users, incorporate it into your evaluation pipeline, and track the performance of your application over time.

RAG — a very quick refresher

For the rest of the tutorial, we will take RAG as an example to demonstrate how to evaluate an LLM application. But before that, here's a very quick refresher on RAG.

This is what a RAG application might look like:



In a RAG application, the goal is to enhance the quality of responses generated by an LLM by supplementing its parametric knowledge with context retrieved from an external knowledge base. To build the knowledge base, large reference documents are broken up into smaller chunks, and each chunk is stored in a database along with its vector embedding generated using an embedding model.

Given a user query, it is first embedded using the same embedding model, and the most relevant chunks are retrieved based on the similarity between the query and chunk vectors. An LLM then uses the user's question, prompt, and the retrieved documents to generate an answer to the question.

How to evaluate a RAG application

The main elements to evaluate in a RAG application are as follows:

- Retrieval: This involves experimenting with different data processing strategies, embedding models, etc., and evaluating how they impact retrieval.
- Generation: Once you decide on the best settings for the retriever, this step involves experimenting with different LLMs to find the best completion model for the task.

In this tutorial, we will evaluate different embedding models for retrieval, different completion models for generation, and the system as a whole with the best-performing models.

Before we begin

Metrics

We will use the following metrics for evaluation:

Retrieval

- Context precision: Evaluates the ability of the retriever to rank retrieved items in order of relevance to the ground truth answer
- Context recall: Measures the extent to which the retrieved context aligns with the ground truth answer

Generation

- Faithfulness: Measures the factual consistency of the generated answer against the retrieved context
- Answer relevance: Measures how relevant the generated answer is to the given prompt (question + retrieved context)

Overall

- Answer semantic similarity: Measures the semantic similarity between the generated answer and the ground truth
- Answer correctness: Measures the accuracy of the generated answer compared to the ground truth

You can read more about how these metrics are calculated.

Tools

We will use LangChain to create a sample RAG application and the RAGAS framework for evaluation. RAGAS is open-source, has out-of-the-box support for all the above metrics, supports custom evaluation prompts, and has integrations with frameworks such as LangChain, LlamaIndex, and observability tools such as LangSmith and Arize Phoenix.

Dataset

We will use the ragas-wikiqa dataset available on Hugging Face. The dataset consists of ~230 general knowledge questions, including the ground truth answers for these questions. Your evaluation dataset, however, should be a good representation of how users will interact with your application.

Where's the code?

The Jupyter Notebook for this tutorial can be found on GitHub.

Step 1: Install the required libraries

We will require the following libraries for this tutorial:

- datasets: Python library to get access to datasets available on Hugging Face Hub
- ragas: Python library for the RAGAS framework
- langchain: Python library to develop LLM applications using LangChain
- langchain-mongodb: Python package to use MongoDB Atlas as a vector store with LangChain
- langchain-openai: Python package to use OpenAI models in LangChain
- pymongo: Python driver for interacting with MongoDB
- pandas: Python library for data analysis, exploration, and manipulation
- tdqm: Python module to show a progress meter for loops
- matplotlib, seaborn: Python libraries for data visualization

```
! pip install -qU datasets ragas langchain langchain-mongodb langchain-openai \
pymongo pandas tqdm matplotlib seaborn
```

Step 2: Setup pre-requisites

In this tutorial, we will use MongoDB Atlas as a vector store and retriever. But first, you will need a MongoDB Atlas account with a database cluster and get the connection string to connect to your cluster. Follow these steps to get set up:

- Register for a free MongoDB Atlas account.
- Follow the instructions to create a new database cluster.
- Follow the instructions to obtain the connection string for your database cluster.

> Don't forget to add the IP of your host machine to the IP Access list for your cluster.

Once you have the connection string, set it in your code:

```
import getpass
MONGODB_URI = getpass.getpass("Enter your MongoDB connection string:")
```

We will be using OpenAI's embedding and chat completion models, so you'll also need to obtain an OpenAI API key and set it as an environment variable for the OpenAI client to use:

```
import os
from openai import OpenAI
os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter your OpenAI API Key:")
openai_client = OpenAI()
```

Step 3: Download the evaluation dataset

As mentioned previously, we will use the ragas-wikiqa dataset available on Hugging Face. We will download it using the datasets library and convert it into a pandas dataframe:

```
from datasets import load_dataset
import pandas as pd

data = load_dataset("explodinggradients/ragas-wikiqa", split="train")
df = pd.DataFrame(data)
```

The dataset has the following columns that are important to us:

- question: User questions
- correct_answer: Ground truth answers to the user questions
- context: List of reference texts to answer the user questions

Step 4: Create reference document chunks

We noticed that the reference texts in the `context` column are quite long. Typically for RAG, large texts are broken down into smaller chunks at ingest time. Given a user query, only the most relevant chunks are retrieved, to pass on as context to the LLM. So as a next step, we will chunk up our reference texts before embedding and ingesting them into MongoDB:

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Split text by tokens using the tiktoken tokenizer
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
    encoding_name="cl100k_base", keep_separator=False, chunk_size=200,   chunk
)

def split_texts(texts):
    chunked_texts = []
    for text in texts:
        chunks = text_splitter.create_documents([text])
        chunked_texts.extend([chunk.page_content for chunk in chunks])
    return chunked_texts

# Split the context field into chunks
df["chunks"] = df["context"].apply(lambda x: split_texts(x))
# Aggregate list of all chunks
all_chunks = df["chunks"].tolist()
docs = [item for chunk in all_chunks for item in chunk]
```

The above code does the following:

- Defines how to split the text into chunks: We use the `from_tiktoken_encoder` method of the `RecursiveCharacterTextSplitter` class in LangChain. This way, the texts are split by character and recursively merged into tokens by the tokenizer as long as the chunk size (in terms of number of tokens) is less than the specified chunk size (`chunk_size`). Some overlap between chunks has been shown to improve retrieval, so we set an overlap of 30 characters in the `chunk_overlap` parameter. The `keep_separator` parameter indicates

whether or not to keep the default separators such as `\n\n`, `\n`, etc. in the chunked text, and the `encoding_name` indicates the model to use to generate tokens.

- Defines a `split_texts` function: This function takes a list of reference texts (`texts`) as input, splits them using the text splitter, and returns the list of chunked texts.
- Applies the `split_texts` function to the `context` column of our dataset
- Creates a list of chunked texts for the entire dataset

> In practice, you may want to experiment with different chunking strategies as well while evaluating retrieval, but for this tutorial, we are only focusing on evaluating different embedding models.

Step 5: Create embeddings and ingest them into MongoDB

Now that we have chunked up our reference documents, let's embed and ingest them into MongoDB Atlas to build a knowledge base (vector store) for our RAG application. Since we want to evaluate two embedding models for the retriever, we will create separate vector stores (collections) using each model.

We will be evaluating the text-embedding-ada-002 and text-embedding-3-small (we will call them ada-002 and 3-small in the rest of the tutorial) embedding models from OpenAI, so first, let's define a function to generate embeddings using OpenAI's Embeddings API:

```
def get_embeddings(docs: List[str], model: str) -> List[List[float]]:
    """
    Generate embeddings using the OpenAI API.

    Args:
        docs (List[str]): List of texts to embed
        model (str, optional): Model name. Defaults to "text-embedding-3-large

    Returns:
```

```
        List[float]: Array of embeddings
    """
    # replace newlines, which can negatively affect performance.
    docs = [doc.replace("\n", " ") for doc in docs]
    response = openai_client.embeddings.create(input=docs, model=model)
    response = [r.embedding for r in response.data]
    return response
```

The embedding function above takes a list of texts (`docs`) and a model name (`model`) as arguments and returns a list of embeddings generated using the specified model. The OpenAI API returns a list of embedding objects, which need to be parsed to get the final list of embeddings. A sample response from the API looks like the following:

```
{
  "data": [
    {
      "embedding": [
        0.018429679796099663,
        -0.009457024745643139
    .
    .
    .
      ],
      "index": 0,
      "object": "embedding"
    }
  ],
  "model": "text-embedding-3-small",
  "object": "list",
  "usage": {
    "prompt_tokens": 183,
    "total_tokens": 183
  }
}
```

Now, let's use each model to embed the chunked texts and ingest them along with their embeddings into a MongoDB collection:

```
from pymongo import MongoClient
from tqdm.auto import tqdm

client = MongoClient(MONGODB_URI)
```

```
DB_NAME = "ragas_evals"
db = client[DB_NAME]
batch_size = 128

EVAL_EMBEDDING_MODELS = ["text-embedding-ada-002", "text-embedding-3-small"]

for model in EVAL_EMBEDDING_MODELS:
    embedded_docs = []
    print(f"Getting embeddings for the {model} model")
    for i in tqdm(range(0, len(docs), batch_size)):
        end = min(len(docs), i + batch_size)
        batch = docs[i:end]
        # Generate embeddings for current batch
        batch_embeddings = get_embeddings(batch, model)
        # Creating the documents to ingest into MongoDB for current batch
        batch_embedded_docs = [
            {"text": batch[i], "embedding": batch_embeddings[i]}
            for i in range(len(batch))
        ]
        embedded_docs.extend(batch_embedded_docs)
    print(f"Finished getting embeddings for the {model} model")

    # Bulk insert documents into a MongoDB collection
    print(f"Inserting embeddings for the {model} model")
    collection = db[model]
    collection.delete_many({})
    collection.insert_many(embedded_docs)
    print(f"Finished inserting embeddings for the {model} model")
```

The above code does the following:

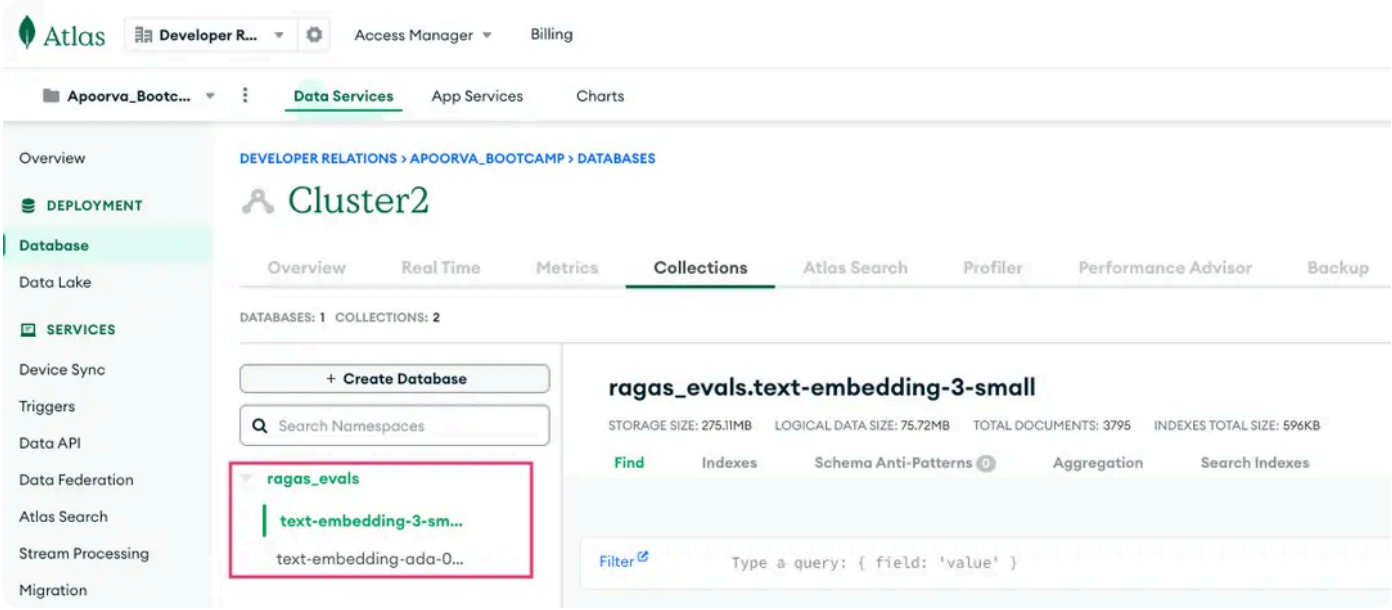
- Creates a PyMongo client (`client`) to connect to a MongoDB Atlas cluster
- Specifies the database (`DB_NAME`) to connect to – we are calling the database ragas_evals; if the database doesn't exist, it will be created at ingest time
- Specifies the batch size (`batch_size`) for generating embeddings in bulk
- Specifies the embedding models (`EVAL_EMBEDDING_MODELS`) to use for generating embeddings

- For each embedding model, generates embeddings for the entire evaluation set and creates the documents to be ingested into MongoDB – an example document looks like the following:

```
{
  "text": "For the purposes of authentication, most countries require commerci
  "embedding": [
    0.018429679796099663,
    -0.009457024745643139,
    .
    .
    .
  ]
}
```

- Deletes any existing documents in the collection named after the model, and bulk inserts the documents into it using the `insert_many()` method

To verify that the above code ran as expected, navigate to the Atlas UI and ensure that you see two collections, namely text-embedding-ada-002 and text-embedding-3-small, in the ragas_evals database:



While you are in the Atlas UI, create vector indexes for both collections. The vector index definition specifies the path to the embedding field, dimensions, and the

similarity metric to use while retrieving documents using vector search. Ensure that the index name is `vector_index` for each collection and that the index definition looks as follows:

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

The number of embedding dimensions in both index definitions is 1536 since ada-002 and 3-small have the same number of dimensions.

Step 6: Compare embedding models for retrieval

As a first step in the evaluation process, we want to ensure that we are retrieving the right context for the LLM. While there are several factors (chunking, re-ranking, etc.) that can impact retrieval, in this tutorial, we will only experiment with different embedding models. We will use the same models that we used in Step 5. We will use LangChain to create a vector store using MongoDB Atlas and use it as a retriever in our RAG application.

```
from langchain_openai import OpenAIEmbeddings
from langchain_mongodb import MongoDBAtlasVectorSearch
from langchain_core.vectorstores import VectorStoreRetriever

def get_retriever(model: str, k: int) -> VectorStoreRetriever:
    """
    Given an embedding model and top k, get a vector store retriever object

    Args:
        model (str): Embedding model to use
        k (int): Number of results to retrieve

```

```
    Returns:
        VectorStoreRetriever: A vector store retriever object
    """
    embeddings = OpenAIEmbeddings(model=model)

    vector_store = MongoDBAtlasVectorSearch.from_connection_string(
        connection_string=MONGODB_URI,
        namespace=f"{DB_NAME}.{model}",
        embedding=embeddings,
        index_name="vector_index",
        text_key="text",
    )

    retriever = vector_store.as_retriever(
        search_type="similarity", search_kwargs={"k": k}
    )
    return retriever
```

The above code defines a `get_retriever` function that takes an embedding model (`model`) and the number of documents to retrieve (`k`) as arguments and returns a retriever object as the output. The function creates a MongoDB Atlas vector store using the `MongoDBAtlasVectorSearch` class from the `langchain-mongodb` integration. Specifically, it uses the `from_connection_string` method of the class to create the vector store from the MongoDB connection string which we obtained in Step 2 above. It also takes additional arguments such as:

- namespace: The (database, collection) combination to use as the vector store
- embedding: Embedding model to use to generate the query embedding for retrieval
- index_name: The MongoDB Atlas vector search index name (as set in Step 5)
- text_key: The field in the reference documents that contains the text

Finally, it uses the `as_retriever` method in LangChain to use the vector store as a retriever. `as_retriever` can take arguments such as `search_type` which specifies the metric to use to retrieve documents. Here, we choose

`similarity` since we want to retrieve the most similar documents to a given query. We can also specify additional search arguments such as `k` which is the number of documents to retrieve.

To evaluate the retriever, we will use the `context_precision` and `context_recall` metrics from the ragas library. These metrics use the retrieved context, ground truth answers, and the questions. So let's first gather the list of ground truth answers and questions:

```
QUESTIONS = df["question"].to_list()
GROUND_TRUTH = df["correct_answer"].tolist()
```

The above code snippet simply converts the `question` and `correct_answer` columns from the dataframe we created in Step 3 to lists. We will reuse these lists in the steps that follow.

Finally, here's the code to evaluate the retriever:

```
from datasets import Dataset
from ragas import evaluate, RunConfig
from ragas.metrics import context_precision, context_recall
import nest_asyncio

# Allow nested use of asyncio (used by RAGAS)
nest_asyncio.apply()

for model in EVAL_EMBEDDING_MODELS:
    data = {"question": [], "ground_truth": [], "contexts": []}
    data["question"] = QUESTIONS
    data["ground_truth"] = GROUND_TRUTH

    retriever = get_retriever(model, 2)
    # Getting relevant documents for the evaluation dataset
    for i in tqdm(range(0, len(QUESTIONS))):
        data["contexts"].append(
            [doc.page_content for doc in retriever.get_relevant_documents(QUES
        )
    # RAGAS expects a Dataset object
    dataset = Dataset.from_dict(data)
    # RAGAS runtime settings to avoid hitting OpenAI rate limits
    run_config = RunConfig(max_workers=4, max_wait=180)
    result = evaluate(
```

```
        dataset=dataset,
        metrics=[context_precision, context_recall],
        run_config=run_config,
        raise_exceptions=False,
    )
    print(f"Result for the {model} model: {result}")
```

The above code does the following for each of the models that we are evaluating:

- Creates a dictionary (`data`) with `question` , `ground_truth` , and `contexts` as keys, corresponding to the questions in the evaluation dataset, their ground truth answers, and retrieved contexts
- Creates a `retriever` that retrieves the top two most similar documents to a given query
- Uses the `get_relevant_documents` method to obtain the most relevant documents for each question in the evaluation dataset and add them to the `contexts` list in the `data` dictionary
- Converts the `data` dictionary to a Dataset object
- Creates a runtime config for RAGAS to override its default concurrency and retry settings – we had to do this to avoid running into OpenAI's rate limits, but this might be a non-issue depending on your usage tier, or if you are not using OpenAI models
- Uses the `evaluate` method from the ragas library to get the overall evaluation metrics for the evaluation dataset

The evaluation results for embedding models we compared look as follows on our dataset:

Model	Context precision	Context recall
ada-002	0.9310	0.8561
3-small	0.9116	0.8826

Based on the above numbers, ada-002 is better at retrieving the most relevant results at the top but 3-small is better at retrieving contexts that are more aligned with the ground truth answers. So we conclude that 3-small is the better embedding model for retrieval.

Step 7: Compare completion models for generation

Now that we've found the best model for our retriever, let's find the best completion model for the generator component in our RAG application.

But first, let's build out our RAG "application." In LangChain, we do this using chains. Chains in LangChain are a sequence of calls either to an LLM, a tool, or a data processing step. Each component in a chain is referred to as a Runnable, and the recommended way to compose chains is using the LangChain Expression Language (LCEL).

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough
from langchain_core.runnables.base import RunnableSequence
from langchain_core.output_parsers import StrOutputParser

def get_rag_chain(retriever: VectorStoreRetriever, model: str) -> RunnableSequ
    """
    Create a basic RAG chain

    Args:
        retriever (VectorStoreRetriever): Vector store retriever object
        model (str): Chat completion model to use

    Returns:
        RunnableSequence: A RAG chain
    """
    # Generate context using the retriever, and pass the user question through
    retrieve = {
        "context": retriever
        | (lambda docs: "\n\n".join([d.page_content for d in docs])),
        "question": RunnablePassthrough(),
    }
    template = """Answer the question based only on the following context: \
    {context}
```

```
26
27      Question: {question}
28      """
29      # Defining the chat prompt
30      prompt = ChatPromptTemplate.from_template(template)
31      # Defining the model to be used for chat completion
32      llm = ChatOpenAI(temperature=0, model=model)
33      # Parse output as a string
34      parse_output = StrOutputParser()
35
36      # Naive RAG chain
37      rag_chain = retrieve | prompt | llm | parse_output
38      return rag_chain
```

In the above code, we define a `get_rag_chain` function that takes a `retriever` object and a chat completion model name (`model`) as arguments and returns a RAG chain as the output. The function creates the following components that together make up the RAG chain:

- retrieve: Takes the user input (a question) and sends it to the retriever to obtain similar documents; it also formats the output to match the input format expected by the next runnable, which in this case is a dictionary with `context` and `question` as keys; the RunnablePassthrough() call for the question key indicates that the user input is simply passed through to the next stage under the question key
- prompt: Crafts a prompt by populating a prompt template with the context and question from the retrieve stage
- llm: Specifies the chat model to use for completion
- parse_output: A simple output parser that parses the result from the LLM into a string

Finally, it creates a RAG chain (`rag_chain`) using LCEL pipe (|) notation to chain together the above components.

For completion models, we will be evaluating the latest updated version of gpt-3.5-turbo and an older version of GPT-3.5 Turbo, i.e., gpt-3.5-turbo-1106. The

evaluation code for the generator looks largely similar to what we had in Step 6 except it has additional steps to initialize the RAG chain and invoke it for each question in our evaluation dataset in order to generate answers:

```
from ragas.metrics import faithfulness, answer_relevancy

for model in ["gpt-3.5-turbo-1106", "gpt-3.5-turbo"]:
    data = {"question": [], "ground_truth": [], "contexts": [], "answer": []}
    data["question"] = QUESTIONS
    data["ground_truth"] = GROUND_TRUTH
    # Using the best embedding model from the retriever evaluation
    retriever = get_retriever("text-embedding-3-small", 2)
    rag_chain = get_rag_chain(retriever, model)
    for i in tqdm(range(0, len(QUESTIONS))):
        question = QUESTIONS[i]
        data["answer"].append(rag_chain.invoke(question))
        data["contexts"].append(
            [doc.page_content for doc in retriever.get_relevant_documents(ques
        )
    # RAGAS expects a Dataset object
    dataset = Dataset.from_dict(data)
    # RAGAS runtime settings to avoid hitting OpenAI rate limits
    run_config = RunConfig(max_workers=4, max_wait=180)
    result = evaluate(
        dataset=dataset,
        metrics=[faithfulness, answer_relevancy],
        run_config=run_config,
        raise_exceptions=False,
    )
    print(f"Result for the {model} model: {result}")
```

A few changes to note in the above code:

- The `data` dictionary has an additional `answer` key to accumulate answers to the questions in our evaluation dataset.
- We use the text-embedding-3-small for the retriever since we determined this to be the better embedding model in Step 6.
- We are using the metrics `faithfulness` and `answer_relevancy` to evaluate the generator.

The evaluation results for the completion models we compared look as follows on our dataset:

Model	Faithfulness	Answer relevance
gpt-3.5-turbo	0.9714	0.9087
gpt-3.5-turbo-1106	0.9671	0.9105

Based on the above numbers, the latest version of gpt-3.5-turbo produces more factually consistent results than its predecessor, while the older version produces answers that are more pertinent to the given prompt. Let's say we want to go with the more "faithful" model.

> If you don't want to choose between metrics, consider creating consolidated metrics using a weighted summation after the fact, or customize the prompts used for evaluation.

Step 8: Measure the overall performance of the RAG application

Finally, let's evaluate the overall performance of the system using the best-performing models:

```
from ragas.metrics import answer_similarity, answer_correctness

data = {"question": [], "ground_truth": [], "answer": []}
data["question"] = QUESTIONS
data["ground_truth"] = GROUND_TRUTH
# Using the best embedding model from the retriever evaluation
retriever = get_retriever("text-embedding-3-small", 2)
# Using the best completion model from the generator evaluation
rag_chain = get_rag_chain(retriever, "gpt-3.5-turbo")
for question in tqdm(QUESTIONS):
    data["answer"].append(rag_chain.invoke(question))

```

```
dataset = Dataset.from_dict(data)
run_config = RunConfig(max_workers=4, max_wait=180)
result = evaluate(
    dataset=dataset,
    metrics=[answer_similarity, answer_correctness],
    run_config=run_config,
    raise_exceptions=False,
)
print(f"Overall metrics: {result}")
```

In the above code, we use the text-embedding-3-small model for the retriever and the gpt-3.5-turbo model for the generator, to generate answers to questions in our evaluation dataset. We use the `answer_similarity` and `answer_correctness` metrics to measure the overall performance of the RAG chain.

The evaluation shows that the RAG chain produces an answer similarity of 0.8873 and an answer correctness of 0.5922 on our dataset.

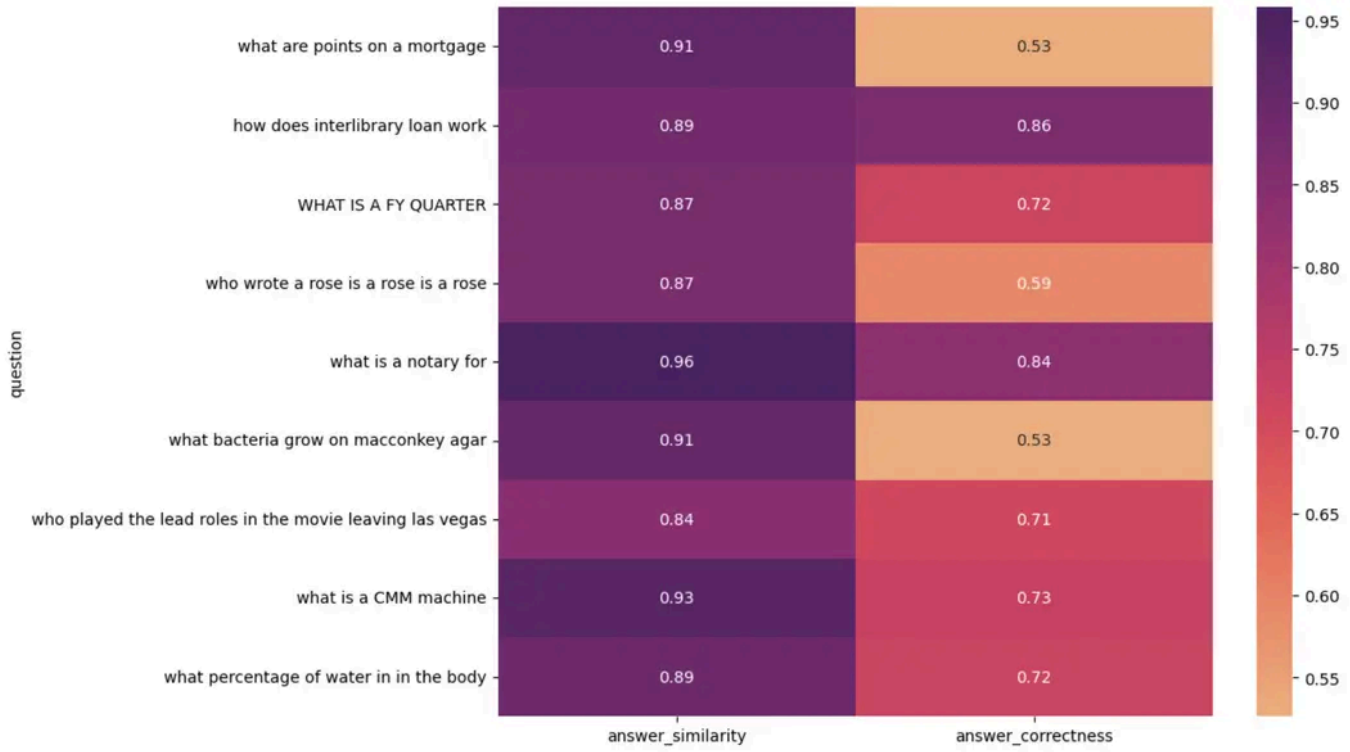
The correctness seems a bit low so let's investigate further. You can convert the results from RAGAS to a pandas dataframe to perform further analysis:

```
result_df = result.to_pandas()
result_df[result_df["answer_correctness"] < 0.7]
```

For a more visual analysis, can also create a heatmap of questions vs metrics:

```
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 8))
sns.heatmap(
    result_df[1:10].set_index("question")[["answer_similarity", "answer_correc
    annot=True,
    cmap="flare",
)
plt.show()
```



Upon manually investigating some of the low-scoring results, we observed the following:

- Some ground-truth answers in the evaluation dataset were in fact incorrect. So although the answer generated by the LLM was right, it didn't match the ground truth answer, resulting in a low score.
- Some ground-truth answers were full sentences whereas the LLM-generated answer, although factually correct, was a single word, number, etc.

The above findings emphasize the importance of spot-checking the LLM evaluations, curating accurate and representative evaluation datasets, and highlight yet another challenge with using LLMs for evaluation.

Step 9: Track performance over time

Evaluation should not be a one-time event. Each time you want to change a component in the system, you should evaluate the changes against existing settings to assess how they will impact performance. Then, once the application is

deployed in production, you should also have a way to monitor performance in real time and detect changes therein.

In this tutorial, we used MongoDB Atlas as the vector database for our RAG application. You can also use Atlas to monitor the performance of your LLM application via Atlas Charts. All you need to do is write evaluation results and any feedback metrics (e.g., number of thumbs up, thumbs down, response regenerations, etc.) that you want to track to a MongoDB collection:

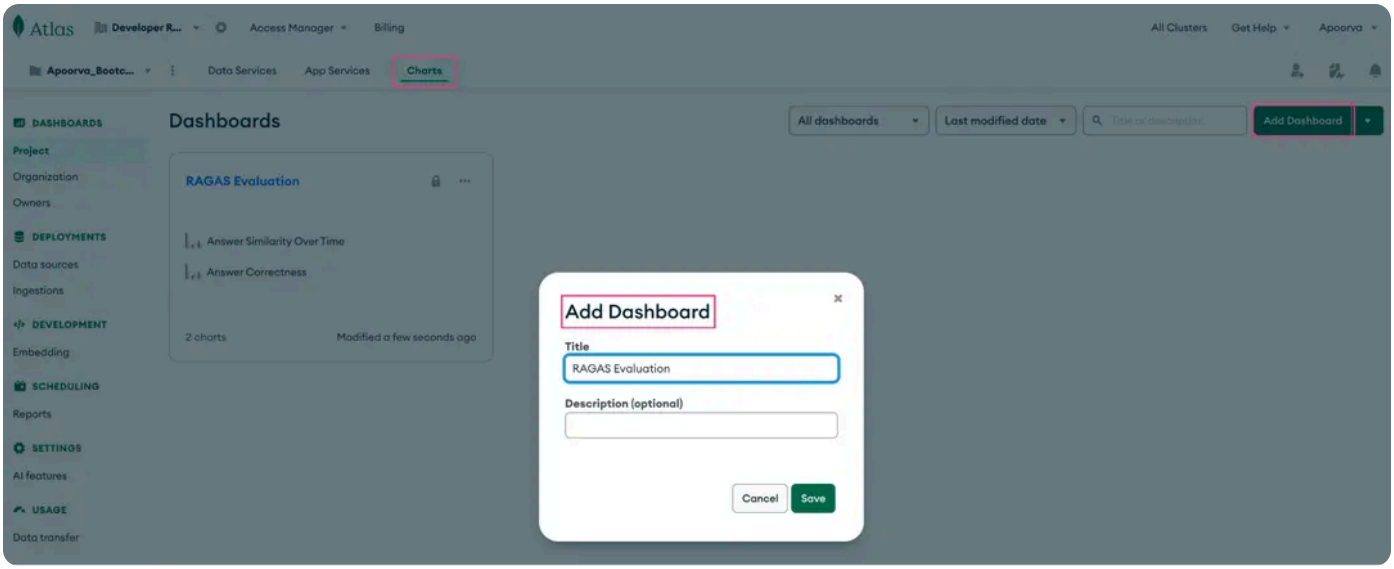
```
from datetime import datetime

result["timestamp"] = datetime.now()
collection = db["metrics"]
collection.insert_one(result)
```

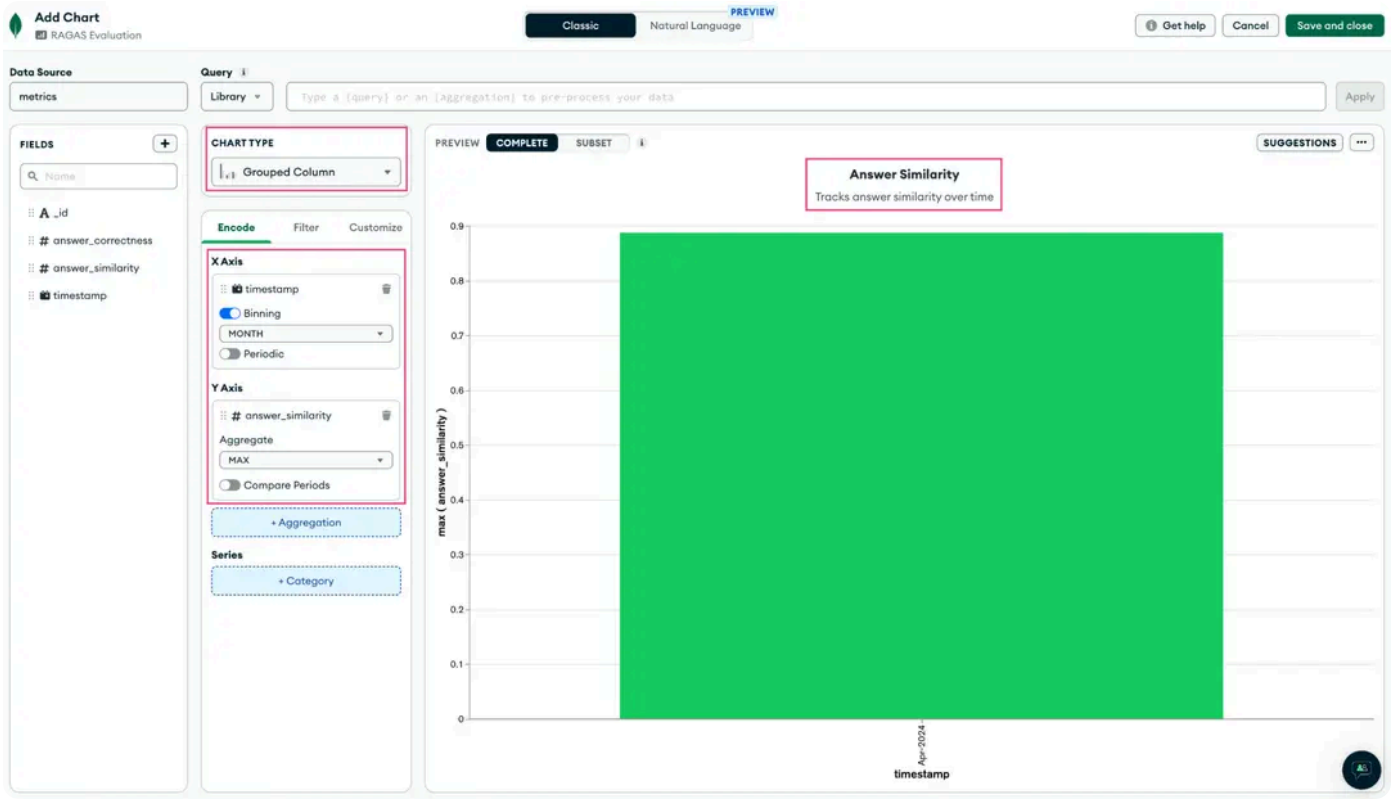
In the above code snippet, we add a `timestamp` field containing the current timestamp to the final evaluation result (`result`) from Step 8, and write it to a collection called metrics in the ragas_evals database using PyMongo's `insert_one` method. The `result` dictionary inserted into MongoDB looks like this:

```
{
  "answer_similarity": 0.8873,
  "answer_correctness": 0.5922,
  "timestamp": 2024-04-07T23:27:30.655+00:00
}
```

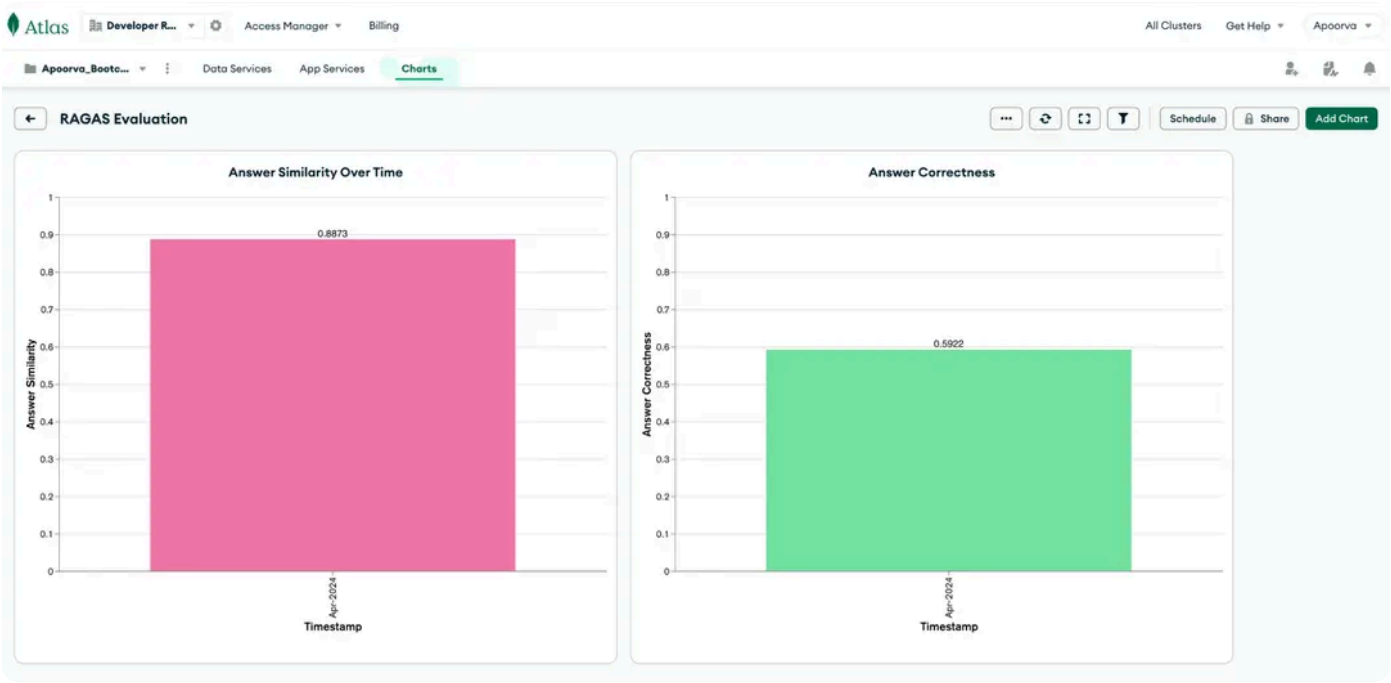
We can now create a dashboard in Atlas Charts to visualize the data in the metrics collection:



Once the dashboard is created, click the Add Chart button and select the metrics collection as the data source for the chart. Drag and drop fields to include, choose a chart type, add a title and description for the chart, and save it to the dashboard:



Here's what our sample dashboard looks like:



Similarly, once your application is in production, you can create a dashboard for any feedback metrics you collect.

Conclusion

In this tutorial, we looked into some of the challenges with evaluating LLM applications, followed by a detailed, step-by-step workflow for evaluating an LLM application, including persisting and tracking evaluation results over time. While we used RAG as our example for evaluation, the concepts and techniques shown in this tutorial can be extended to other LLM applications, including agents.

Now that you have a good foundation on how to evaluate RAG applications, you can take it up as a challenge to evaluate RAG systems from some of our other tutorials:

- Building a RAG System With Google's Gemma, Hugging Face, and MongoDB
- Building a RAG System Using Claude Opus and MongoDB

If you have further questions about LLM evaluations, please reach out to us in our Generative AI community forums and stay tuned for the next tutorial in the RAG

series. Previous tutorials from the series can be found below:

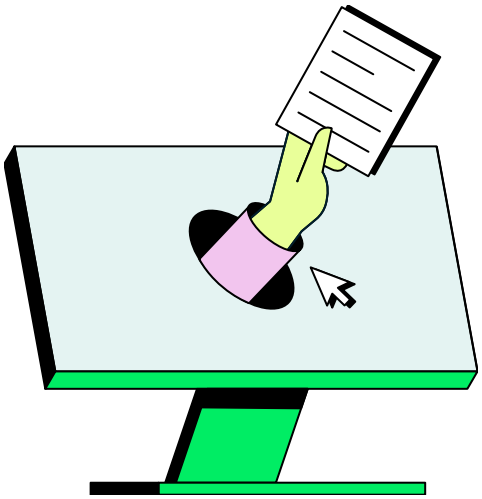
- Part 1: How to Choose the Right Embedding Model for Your RAG Application

References

If you would like to learn more about evaluating LLM applications, check out the following references:

- https://docs.ragas.io/en/latest/getstarted/index.html
- Yan, Ziyou. (Oct 2023). AI Engineer Summit - Building Blocks for LLM Systems & Products. eugeneyan.com. https://eugeneyan.com/speaking/ai-eng-summit/
- Yan, Ziyou. (Mar 2024). LLM Task-Specific Evals that Do & Don't Work. eugeneyan.com. https://eugeneyan.com/writing/evals/
- Yan, Ziyou. (Jul 2023). Patterns for Building LLM-based Systems & Products. eugeneyan.com. https://eugeneyan.com/writing/llm-patterns/
- https://aiconference.com/speakers/jerry-liu/
- https://www.databricks.com/blog/LLM-auto-eval-best-practices-RAG
- https://huggingface.co/learn/cookbook/en/rag_evaluation
- Llamaindex evals framework

Top Comments in Forums



There are no comments on this article yet.

Start the Conversation

Rate this tutorial

Related

TUTORIAL

How to Build an Animated Timeline Chart with the MongoDB Charts Embedding SDK

Dec 13, 2023 | 6 min read

TUTORIAL

Sentiment Chef Agent App with Google Cloud and MongoDB Atlas

Mar 14, 2025 | 16 min read

TUTORIAL

Building Generative AI Applications Using MongoDB: Harnessing the Power of Atlas Vector Search and Open Source Models

Mar 12, 2025 | 10 min read

TUTORIAL

Leveraging OpenAI and MongoDB Atlas for Improved Search Functionality

Sep 18, 2024 | 5 min read

Request a Tutorial

English

About

Careers

Support

Contact Us