Build a RAG based question answer solution using Amazon Bedrock Knowledge Base, Amazon OpenSearch Service Serverless with vector search and LangChain

*Amit Arora*



One of the most common applications of generative AI and large language models (LLMs) in an enterprise environment is answering questions based on the enterprise’s knowledge corpus. [Amazon Lex](https://aws.amazon.com/lex/) provides the framework for building [AI based chatbots](https://aws.amazon.com/solutions/retail/ai-for-chatbots). Pre-trained foundation models (FMs) perform well at natural language understanding (NLU) tasks such summarization, text generation and question answering on a broad variety of topics but either struggle to provide accurate (without hallucinations) answers or completely fail at answering questions about content that they haven’t seen as part of their training data. Furthermore, FMs are trained with a point in time snapshot of data and have no inherent ability to access fresh data at inference time; without this ability they might provide responses that are potentially incorrect or inadequate.

A commonly used approach to address this problem is to use a technique called Retrieval Augmented Generation (RAG). In the RAG-based approach we convert the user question into vector embeddings using an LLM and then do a similarity search for these embeddings in a pre-populated vector database holding the embeddings for the enterprise knowledge corpus. A small number of similar documents (typically three) is added as context along with the user question to the “prompt” provided to another LLM and then that LLM generates an answer to the user question using information provided as context in the prompt. RAG models were introduced by [Lewis et al.](https://arxiv.org/abs/2005.11401) in 2020 as a model where parametric memory is a pre-trained seq2seq model and the non-parametric memory is a dense vector index of Wikipedia, accessed with a pre-trained neural retriever. To understand the overall structure of a RAG-based approach, refer to [Build a powerful question answering bot with Amazon SageMaker, Amazon OpenSearch Service, Streamlit, and LangChain](https://aws.amazon.com/blogs/machine-learning/build-a-powerful-question-answering-bot-with-amazon-sagemaker-amazon-opensearch-service-streamlit-and-langchain/).

In this post we provide a step-by-step guide with all the building blocks for creating an enterprise ready RAG application such as a question answering solution. We use LLMs available through Amazon Bedrock for the embeddings model (Amazon Titan Text Embeddings v2), the text generation model (Anthropic Claude v2) and the Amazon Bedrock Knowledge Base for this solution. The text corpus representing an enterprise knowledge base is stored as HTML files in Amazon S3 and is ingested in the form of text embeddings into an index in a Amazon OpenSearch Service Serverless collection using Bedrock knowledge base agent in a fully-managed serverless fashion.

We provide an AWS Cloud Formation template to stand up all the resources required for building this solution. We then demonstrate how to use [LangChain](https://www.langchain.com) to interface with the Bedrock and [opensearch-py](https://pypi.org/project/opensearch-py/) to interface with OpenSearch Service Serverless and build a RAG based question answer workflow.

## Solution overview

We use a subset of [SageMaker docs](https://sagemaker.readthedocs.io) as the knowledge corpus for this post. The data is available in the form of HTML files in an S3 bucket, a Bedrock Knowledge Base Agent then reads these files, converts them into smaller chunks, encodes these chunks into vectors (embeddings) and then ingests these embeddings into an OpenSearch Service Serverless collection index. We implement the RAG functionality in a notebook, a set of SageMaker related questions is asked of the Claude model without providing any additional context and then the same questions are asked again but this time with context based on similar documents retrieved from OpenSearch Service Serverless i.e. using the RAG approach. We demonstrate the responses generated without RAG could be factually inaccurate whereas the RAG based responses are accurate and more useful.

All the code for this post is available in the [GitHub repo](https://github.com/aws-samples/bedrock-kb-rag/tree/main/blogs/rag).

The following figure represents the high-level architecture of the proposed solution.

|  |
| --- |
| Figure 1: Architecture |

Step-by-step explanation:

1. The user provides a question via the Jupyter notebook.
2. The question is converted into embedding using Bedrock via the Titan embeddings v2 model.
3. The embedding is used to find similar documents from an OpenSearch Service Serverless index.
4. The similar documents long with the user question are used to create a “prompt”.
5. The prompt is provided to Bedrock to generate a response using the Claude v2 model.
6. The response along with the context is printed out in a notebook cell.

As illustrated in the architecture diagram, we use the following AWS services:

* [Bedrock](https://aws.amazon.com/bedrock/) for access to the LLMs for embedding and text generation as well as for the knowledge base agent.
* [OpenSearch Service Serverless with vector search](https://aws.amazon.com/opensearch-service/serverless-vector-engine/) for storing the embeddings of the enterprise knowledge corpus and doing similarity search with user questions.
* [S3](https://aws.amazon.com/pm/serv-s3/) for storing the raw knowledge corpus data (HTML files).
* [AWS Identity and Access Management](https://aws.amazon.com/iam/) roles and policies for access management.
* [AWS CloudFormation](https://aws.amazon.com/cloudformation/) for creating the entire solution stack through infrastructure as code.

In terms of open-source packages used in this solution, we use [LangChain](https://python.langchain.com/en/latest/index.html) for interfacing with Bedrock and [opensearch-py](https://pypi.org/project/opensearch-py/) to interface with OpenSearch Service Serverless.

The workflow for instantiating the solution presented in this post in your own AWS account is as follows:

1. Run the CloudFormation template provided with this post in your account. This will create all the necessary infrastructure resources needed for this solution:
   1. OpenSearch Service Serverless collection
   2. SageMaker Notebook
   3. IAM roles
2. Create a vector index in the OpenSearch Service Serverless collection. This is done through the OpenSearch Service Serverless console.
3. Create a knowledge base in Bedrock and synch data from the S3 bucket to the OpenSearch Service Serverless collection index. This is done through the Bedrock console.
4. Run the [rag\_w\_bedrock\_and\_aoss.ipynb](./rag_w_bedrock_and_aoss.ipynb) notebook in the SageMaker notebook to ask questions based on the data ingested in OpenSearch Service Serverless collection index.

These steps are discussed in detail in the following sections.

### Prerequisites

To implement the solution provided in this post, you should have an [AWS account](https://signin.aws.amazon.com/signin?redirect_uri=https%3A%2F%2Fportal.aws.amazon.com%2Fbilling%2Fsignup%2Fresume&client_id=signup) and awareness about LLMs, OpenSearch Service and Bedrock.

#### Use AWS Cloud Formation to create the solution stack

Choose **Launch Stack** for the Region you want to deploy resources to. All parameters needed by the CloudFormation template have default values already filled in, except for ARN of the IAM role with which you are currently logged into your AWS account which you’d have to provide. Make a note of the OpenSearch Service collection ARN, we use this in subsequent steps. **This template takes about 5 minutes to complete**.

| AWS Region | Link |
| --- | --- |
| us-east-1 (N. Virginia) |  |
| us-west-2 (Oregon) |  |

After the stack is created successfully, navigate to the stack’s Outputs tab on the AWS CloudFormation console and note the values for CollectionARN and AOSSVectorIndexName. We use those in the subsequent steps.

|  |
| --- |
| Figure 2: Cloud Formation Stack Outputs |

#### Create an OpenSearch Service Serverless vector index

The CloudFormation stack creates an OpenSearch Service Serverless collection, the next step is to create a vector index. This is done through the OpenSearch Service Serverless console as described below.

1. Navigate to OpenSearch Service console and click on Collections. The sagemaker-kb collection created by the CloudFormation stack will be listed there.

|  |
| --- |
| * Figure 3: SageMaker Knowledge Base Collection |

1. Click on the sagemaker-kb link to create a vector index for storing the embeddings from the documents in S3.

|  |
| --- |
| * Figure 4: SageMaker Knowledge Base Vector Index |

1. Set the vector index name as sagemaker-readthedocs-io, vector field name as vector dimensions as 1536, and distance metric as Euclidean. **It is required that you set these parameters exactly as mentioned here because the Bedrock Knowledge Base Agent is going to use these same values**.

|  |
| --- |
| * Figure 5: SageMaker Knowledge Base Vector Index Parameters |

1. Once created the vector index is listed as part of the collection.

|  |
| --- |
| * Figure 6: SageMaker Knowledge Base Vector Index Created |

#### Create a Bedrock knowledge base

Once the OpenSearch Service Serverless collection and vector index have been created, it is time to setup the Bedrock knowledge base.

1. Navigate to the Bedrock Console and click on Knowledge Base and click on the Created Knowledge Base button.

|  |
| --- |
| * Figure 7: Bedrock Knowledge Base |

1. Fill out the details for creating the knowledge base as shown in the screenshots below.

|  |
| --- |
| * Figure 8: Bedrock Knowledge Base |

1. Select the S3 bucket.

|  |
| --- |
| * Figure 9: Bedrock Knowledge Base S3 bucket |

1. The Titan embeddings model is automatically selected.

|  |
| --- |
| * Figure 10: Bedrock Knowledge Base embeddings model |

1. Select Amazon OpenSearch Service Serverless from the vector database options available.

|  |
| --- |
| * Figure 11: Bedrock Knowledge Base OpenSearch Service Serverless |

1. Review and create the knowledge base by clicking the Create knowledge base button.

|  |
| --- |
| * Figure 12: Bedrock Knowledge Base Review & Create |

1. The knowledge base should be created now.

|  |
| --- |
| * Figure 13: Bedrock Knowledge Base create complete |

##### Sync the Bedrock knowledge base

Once the Bedrock knowledge base is created we are now ready to sync the data (raw documents) in S3 to embeddings in the OpenSearch Service Serverless collection vector index.

1. Start the Sync by pressing the Sync button, the button label changes to Syncing.

|  |
| --- |
| * Figure 14: Bedrock Knowledge Base sync |

1. Once the Sync completes the status changes to Ready.

|  |
| --- |
| * Figure 15: Bedrock Knowledge Base sync completed |

#### Run the RAG notebook

Now we are all set to ask some questions off our newly created knowledge base. The CloudFormation template creates a SageMaker Notebook that contains the code to demonstrate this.

1. Navigate to SageMaker Notebooks and find the notebook named bedrock-kb-rag-workshop and click on Open Jupyter Lab.

|  |
| --- |
| * Figure 16: RAG with Bedrock KB notebook |

1. Open a new Terminal from File -> New -> Terminal and run the following commands to install the Bedrock SDK in a new conda kernel called bedrock\_py39.

* chmod +x /home/ec2-user/SageMaker/bedrock-kb-rag-workshop/setup\_bedrock\_conda.sh  
  /home/ec2-user/SageMaker/bedrock-kb-rag-workshop/setup\_bedrock\_conda.sh

1. Wait for one minute after completing the previous step and now click on the rag\_w\_bedrock\_and\_aoss.ipynb to open the notebook. *Confirm that the notebook is using the newly created bedrock\_py39 kernel, otherwise the code will not work. In case the kernel is not set to bedrock\_py39 then refresh the page and this time the bedrock\_py39 kernel would be selected*.
2. The notebook code demonstrates use of Bedrock, LangChain and opensearch-py packages for implementing the RAG technique for question answering.
3. We access the models available via Bedrock using the Bedrock and BedrockEmbeddings classes from the LangChain package.

* # we will use Anthropic Claude for text generation  
  claude\_llm = Bedrock(model\_id= "anthropic.claude-v2")  
  claude\_llm.model\_kwargs = dict(temperature=0.5, max\_tokens\_to\_sample=300, top\_k=250, top\_p=1, stop\_sequences=[])  
    
  # we will be using the Titan Embeddings Model to generate our Embeddings.  
  embeddings = BedrockEmbeddings(model\_id = "amazon.titan-embed-g1-text-02")

1. Interface to OpenSearch Service Serverless is through the opensearch-py package.

* # Functions to talk to OpenSearch  
    
  # Define queries for OpenSearch  
  def query\_docs(query: str, embeddings: BedrockEmbeddings, aoss\_client: OpenSearch, index: str, k: int = 3) -> Dict:  
   """  
   Convert the query into embedding and then find similar documents from OpenSearch Service Serverless  
   """  
    
   # embedding  
   query\_embedding = embeddings.embed\_query(query)  
    
   # query to lookup OpenSearch kNN vector. Can add any metadata fields based filtering  
   # here as part of this query.  
   query\_qna = {  
   "size": k,  
   "query": {  
   "knn": {  
   "vector": {  
   "vector": query\_embedding,  
   "k": k  
   }  
   }  
   }  
   }  
    
   # OpenSearch API call  
   relevant\_documents = aoss\_client.search(  
   body = query\_qna,  
   index = index  
   )  
   return relevant\_documents

1. We combine the prompt and the documents retrieved from OpenSearch Service Serverless as follows.

* def create\_context\_for\_query(q: str, embeddings: BedrockEmbeddings, aoss\_client: OpenSearch, vector\_index: str) -> str:  
   """  
   Create a context out of the similar docs retrieved from the vector database  
   by concatenating the text from the similar documents.  
   """  
   print(f"query -> {q}")  
   aoss\_response = query\_docs(q, embeddings, aoss\_client, vector\_index)  
   context = ""  
   for r in aoss\_response['hits']['hits']:  
   s = r['\_source']  
   print(f"{s['metadata']}\n{s['text']}")  
   context += f"{s['text']}\n"  
   print("----------------")  
   return context

1. Combining everything, the RAG workflow works as shown below.

* # 1. Start with the query  
  q = "What versions of XGBoost are supported by Amazon SageMaker?"  
    
  # 2. Create the context by finding similar documents from the knowledge base  
  context = create\_context\_for\_query(q, embeddings, client, aoss\_vector\_index)  
    
  # 3. Now create a prompt by combining the query and the context  
  prompt = PROMPT\_TEMPLATE.format(context, q)  
    
  # 4. Provide the prompt to the LLM to generate an answer to the query based on context provided  
  response = claude\_llm(prompt)

1. Here is an example of a sample question answered first with just the question in the prompt i.e. without providing any additional context. The answer without context is inaccurate.

|  |
| --- |
| * Figure 17: Answer with prompt alone |

1. We then ask the same question but this time with the additional context retrieved from the knowledge base included in the prompt. Now the inaccuracy in the earlier response is addressed and we also have attribution as to the source of this answer (notice the underlined text for the filename and the actual answer)!

|  |
| --- |
| * Figure 18: Answer with prompt and context |

## Clean up

To avoid incurring future charges, delete the resources. You can do this by first deleting all the files from the S3 bucket created by the CloudFormation template and then deleting the CloudFormation stack.

## Conclusion

In this post, we showed how to create an enterprise ready RAG solution using a combination of AWS services and open-source Python packages.

We encourage you to learn more by exploring [Amazon Titan](https://aws.amazon.com/bedrock/titan/) models, [Amazon Bedrock](https://aws.amazon.com/bedrock/), and [OpenSearch Service](https://aws.amazon.com/opensearch-service/) and building a solution using the sample implementation provided in this post and a dataset relevant to your business. If you have questions or suggestions, leave a comment.

## Author bio

Amit Arora is an AI and ML Specialist Architect at Amazon Web Services, helping enterprise customers use cloud-based machine learning services to rapidly scale their innovations. He is also an adjunct lecturer in the MS data science and analytics program at Georgetown University in Washington D.C.