

# Dive deep into vector data stores using Amazon Bedrock Knowledge Bases

by Vishwa Gupta, Abhishek Madan, Ginni Malik, Isaac Privitera, and Satish Sarapuri | on 11 OCT 2024 | in [Amazon Aurora](#), [Amazon Bedrock](#), [Amazon OpenSearch Service](#), [Technical How-to](#) | [Permalink](#) | [Comments](#) | [Share](#)

Customers across all industries are experimenting with generative AI to accelerate and improve business outcomes. Generative AI is used in various use cases, such as content creation, personalization, intelligent assistants, questions and answers, summarization, automation, cost-efficiencies, productivity improvement assistants, customization, innovation, and more.

Generative AI solutions often use Retrieval Augmented Generation (RAG) architectures, which augment external knowledge sources for improving content quality, context understanding, creativity, domain-adaptability, personalization, transparency, and explainability.

This post dives deep into [Amazon Bedrock Knowledge Bases](#), which helps with the storage and retrieval of data in vector databases for RAG-based workflows, with the objective to improve large language model (LLM) responses for inference involving an organization's datasets.

## Benefits of vector data stores

Several challenges arise when handling complex scenarios dealing with data like data volumes, multi-dimensionality, multi-modality, and other interfacing complexities. For example:

- Data such as images, text, and audio need to be represented in a structured and efficient manner
- Understanding the semantic similarity between data points is essential in generative AI tasks like natural language processing (NLP), image recognition, and recommendation systems
- As the volume of data continues to grow rapidly, scalability becomes a significant challenge
- Traditional databases may struggle to efficiently handle the computational demands of generative AI tasks, such as training complex models or performing inference on large datasets
- Generative AI applications frequently require searching and retrieving similar items or patterns within datasets, such as finding similar images or recommending relevant content
- Generative AI solutions often involve integrating multiple components and technologies, such as deep learning frameworks, data processing pipelines, and deployment environments

Vector databases serve as a foundation in addressing these data needs for generative AI solutions, enabling efficient representation, semantic understanding, scalability, interoperability, search and retrieval, and model deployment. They contribute to the effectiveness and feasibility of generative AI applications across various domains. Vector databases offer the following capabilities:

- Provide a means to represent data in a structured and efficient manner, enabling computational processing and manipulation

- Enable the measurement of semantic similarity by encoding data into vector representations, allowing for comparison and analysis
- Handle large-scale datasets efficiently, enabling processing and analysis of vast amounts of information in a scalable manner
- Provide a common interface for storing and accessing data representations, facilitating interoperability between different components of the AI system
- Support efficient search and retrieval operations, enabling quick and accurate exploration of large datasets

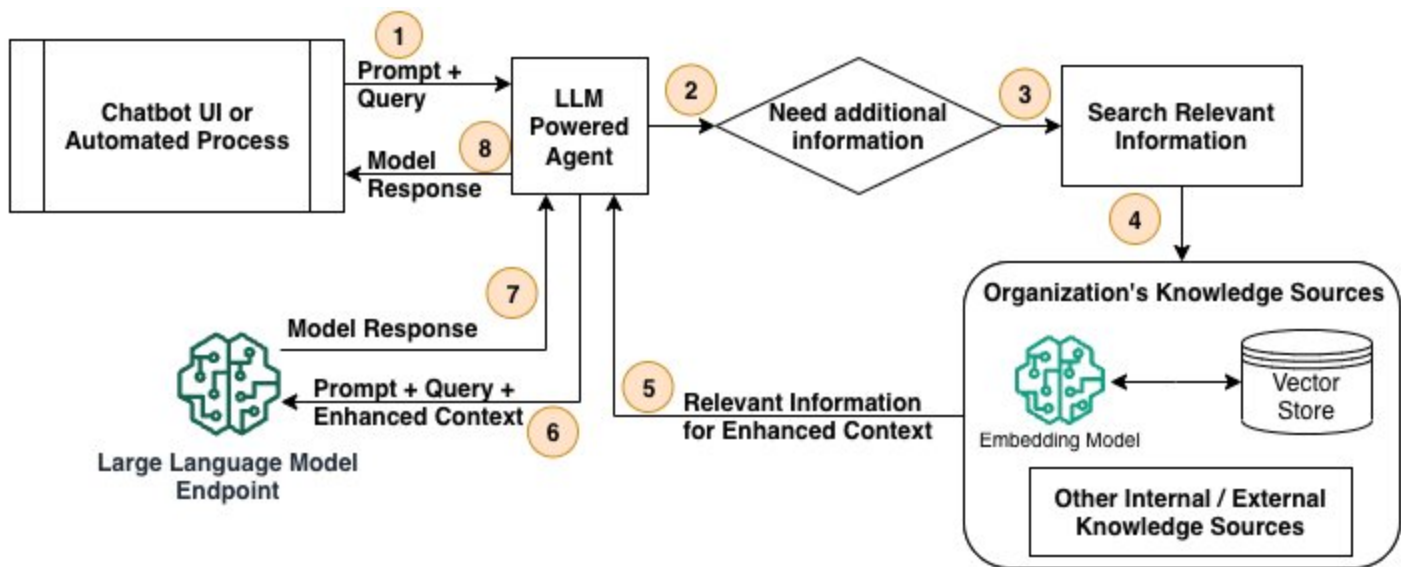
To help implement generative AI-based applications securely at scale, AWS provides [Amazon Bedrock](#), a fully managed service that enables deploying generative AI applications that use high-performing LLMs from leading AI startups and Amazon. With the Amazon Bedrock serverless experience, you can experiment with and evaluate top foundation models (FMs) for your use cases, privately customize them with your data using techniques such as fine-tuning and RAG, and build agents that run tasks using enterprise systems and data sources.

In this post, we dive deep into the vector database options available as part of Amazon Bedrock Knowledge Bases and the applicable use cases, and look at working code examples. Amazon Bedrock Knowledge Bases enables faster time to market by abstracting from the heavy lifting of building pipelines and providing you with an out-of-the-box RAG solution to reduce the build time for your application.

## Knowledge base systems with RAG

RAG optimizes LLM responses by referencing authoritative knowledge bases outside of its training data sources before generating a response. Out of the box, LLMs are trained on vast volumes of data and use billions of parameters to generate original output for tasks like answering questions, translating languages, and completing sentences. RAG extends the existing powerful capabilities of LLMs to specific domains or an organization's internal knowledge base, all without the need to retrain the model. It's a cost-effective approach to improving an LLM's output so it remains relevant, accurate, and useful in various contexts.

The following diagram depicts the high-level steps of a RAG process to access an organization's internal or external knowledge stores and pass the data to the LLM.



The workflow consists of the following steps:

1. Either a user through a chatbot UI or an automated process issues a prompt and requests a response from the LLM-based application.
2. An LLM-powered agent, which is responsible for orchestrating steps to respond to the request, checks if additional information is needed from knowledge sources.
3. The agent decides which knowledge source to use.
4. The agent invokes the process to retrieve information from the knowledge source.
5. The relevant information (enhanced context) from the knowledge source is returned to the agent.
6. The agent adds the enhanced context from the knowledge source to the prompt and passes it to the LLM endpoint for the response.
7. The LLM response is passed back to the agent.
8. The agent returns the LLM response to the chatbot UI or the automated process.

## Use cases for vector databases for RAG

In the context of RAG architectures, the external knowledge can come from relational databases, search and document stores, or other data stores. However, simply storing and searching through this external data using traditional methods (such as keyword search or inverted indexes) can be inefficient and might not capture the true semantic relationships between data points. Vector databases are recommended for RAG use cases because they enable similarity search and dense vector representations.

The following are some scenarios where loading data into a vector database can be advantageous for RAG use cases:

- **Large knowledge bases** – When dealing with extensive knowledge bases containing millions or billions of documents or passages, vector databases can provide efficient similarity search capabilities.

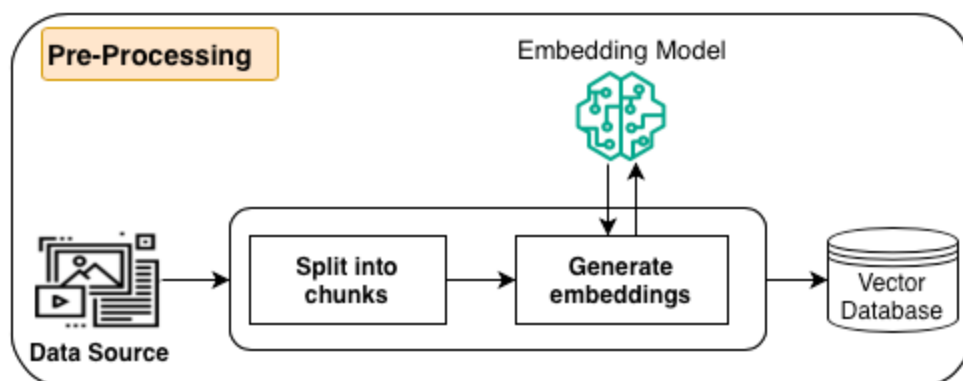
- **Unstructured or semi-structured data** – Vector databases are particularly well-suited for handling unstructured or semi-structured data, such as text documents, webpages, or natural language content. By converting the textual data into dense vector representations, vector databases can effectively capture the semantic relationships between documents or passages, enabling more accurate retrieval.
- **Multilingual knowledge bases** – In RAG systems that need to handle knowledge bases spanning multiple languages, vector databases can be advantageous. By using multilingual language models or cross-lingual embeddings, vector databases can facilitate effective retrieval across different languages, enabling cross-lingual knowledge transfer.
- **Semantic search and relevance ranking** – Vector databases excel at semantic search and relevance ranking tasks. By representing documents or passages as dense vectors, the retrieval component can use vector similarity measures to identify the most semantically relevant content.
- **Personalized and context-aware retrieval** – Vector databases can support personalized and context-aware retrieval in RAG systems. By incorporating user profiles, preferences, or contextual information into the vector representations, the retrieval component can prioritize and surface the most relevant content for a specific user or context.

Although vector databases offer advantages in these scenarios, their implementation and effectiveness may depend on factors such as the specific vector embedding techniques used, the quality and representation of the data, and the computational resources available for indexing and retrieval operations. With Amazon Bedrock Knowledge Bases, you can give FMs and agents contextual information from your company's private data sources for RAG to deliver more relevant, accurate, and customized responses.

## Amazon Bedrock Knowledge Bases with RAG

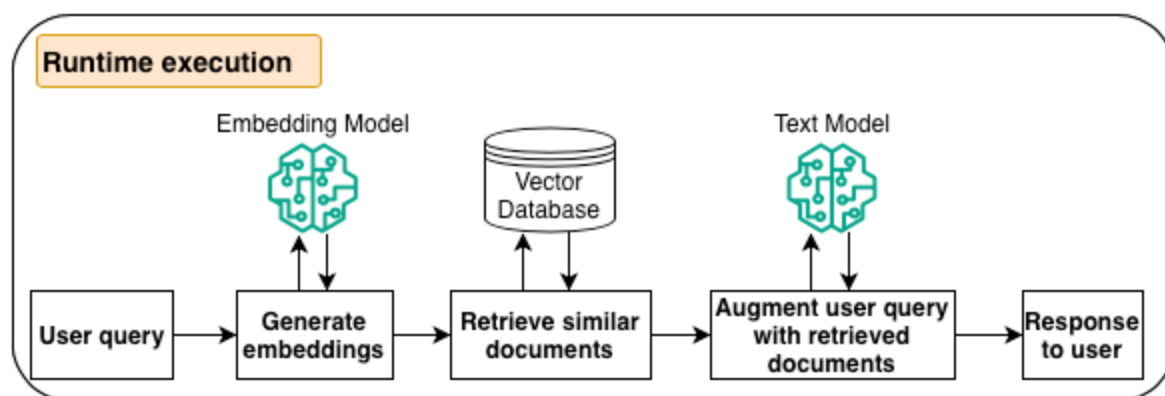
Amazon Bedrock Knowledge Bases is a fully managed capability that helps with the implementation of the entire RAG workflow, from ingestion to retrieval and prompt augmentation, without having to build custom integrations to data sources and manage data flows. Knowledge bases are essential for various use cases, such as customer support, product documentation, internal knowledge sharing, and decision-making systems. A RAG workflow with knowledge bases has two main steps: data preprocessing and runtime execution.

The following diagram illustrates the data preprocessing workflow.



As part of preprocessing, information (structured data, unstructured data, or documents) from data sources is first split into manageable chunks. The chunks are converted to embeddings using embeddings models available in Amazon Bedrock. Lastly, the embeddings are written into a vector database index while maintaining a mapping to the original document. These embeddings are used to determine semantic similarity between queries and text from the data sources. All these steps are managed by Amazon Bedrock.

The following diagram illustrates the workflow for the runtime execution.



During the inference phase of the LLM, when the agent determines that it needs additional information, it reaches out to knowledge bases. The process converts the user query into vector embeddings using an Amazon Bedrock embeddings model, queries the vector database index to find semantically similar chunks to the user's query, converts the retrieved chunks to text and augments the user query, and then responds back to the agent.

Embeddings models are needed in the preprocessing phase to store data in vector databases and during the runtime execution phase to generate embeddings for the user query to search the vector database index. Embeddings models map high-dimensional and sparse data like text into dense vector representations to be efficiently stored and processed by vector databases, and encode the semantic meaning and relationships of data into the vector space to enable meaningful similarity searches. These models support mapping different data types like text, images, audio, and video into the same vector space to enable multi-modal queries and analysis. Amazon Bedrock Knowledge Bases provides industry-leading [embeddings models](#) to enable use cases such as semantic search, RAG, classification, and clustering, to name a few, and provides multilingual support as well.

## Vector database options with Amazon Bedrock Knowledge Bases

At the time of writing this post, Amazon Bedrock Knowledge Bases provides five integration options: the [Vector Engine for Amazon OpenSearch Serverless](#), [Amazon Aurora](#), MongoDB Atlas, Pinecone, and Redis Enterprise Cloud, with more vector database options to come. In this post, we discuss use cases, features, and steps to set up and retrieve information using these vector databases. Amazon Bedrock makes it straightforward to adopt any of these choices by providing a common set of APIs, industry-leading embedding models, security, governance, and observability.

## Role of metadata while indexing data in vector databases

Metadata plays a crucial role when loading documents into a vector data store in Amazon Bedrock. It provides additional context and information about the documents, which can be used for various purposes, such as filtering, sorting, and enhancing search capabilities.

The following are some key uses of metadata when loading documents into a vector data store:

- **Document identification** – Metadata can include unique identifiers for each document, such as document IDs, URLs, or file names. These identifiers can be used to uniquely reference and retrieve specific documents from the vector data store.
- **Content categorization** – Metadata can provide information about the content or category of a document, such as the subject matter, domain, or topic. This information can be used to organize and filter documents based on specific categories or domains.
- **Document attributes** – Metadata can store additional attributes related to the document, such as the author, publication date, language, or other relevant information. These attributes can be used for filtering, sorting, or faceted search within the vector data store.
- **Access control** – Metadata can include information about access permissions or security levels associated with a document. This information can be used to control access to sensitive or restricted documents within the vector data store.
- **Relevance scoring** – Metadata can be used to enhance the relevance scoring of search results. For example, if a user searches for documents within a specific date range or authored by a particular individual, the metadata can be used to prioritize and rank the most relevant documents.
- **Data enrichment** – Metadata can be used to enrich the vector representations of documents by incorporating additional contextual information. This can potentially improve the accuracy and quality of search results.
- **Data lineage and auditing** – Metadata can provide information about the provenance and lineage of documents, such as the source system, data ingestion pipeline, or other transformations applied to the data. This information can be valuable for data governance, auditing, and compliance purposes.

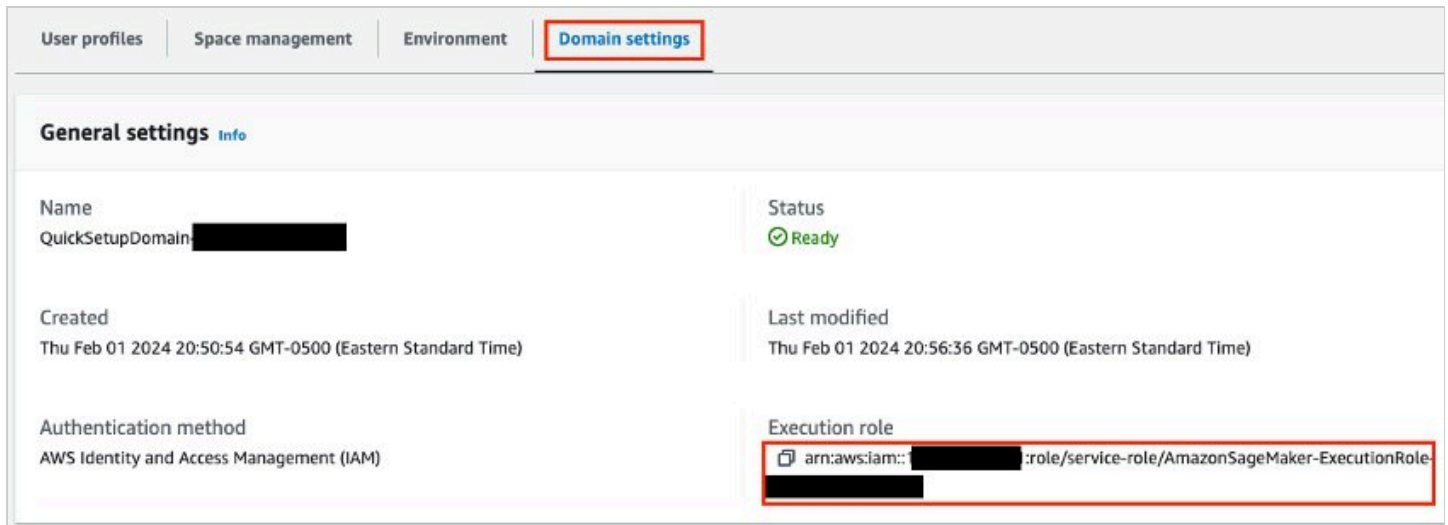
## Prerequisites

Complete the steps in this section to set up the prerequisite resources and configurations.

### Configure Amazon SageMaker Studio

The first step is to set up an [Amazon SageMaker Studio](#) notebook to run the code for this post. You can set up the notebook in any AWS Region where Amazon Bedrock Knowledge Bases is [available](#).

1. Complete the [prerequisites](#) to set up [Amazon SageMaker](#).
2. Complete the [quick setup](#) or [custom setup](#) to enable your SageMaker Studio domain and user profile.  
You also need an [AWS Identity and Access Management](#) (IAM) role assigned to the SageMaker Studio domain. You can identify the role on the SageMaker console. On the Domains page, open your domain. The IAM role ARN is listed on the Domain settings tab.



The role needs permissions for IAM, [Amazon Relational Database Service](#) (Amazon RDS), Amazon Bedrock, [AWS Secrets Manager](#), [Amazon Simple Storage Service](#) (Amazon S3), and [Amazon OpenSearch Serverless](#).

3. Modify the role permissions to add the following policies:

- a. `IAMFullAccess`
- b. `AmazonRDSFullAccess`
- c. `AmazonBedrockFullAccess`
- d. `SecretsManagerReadWrite`
- e. `AmazonRDSDataFullAccess`
- f. `AmazonS3FullAccess`

g. The following inline policy:

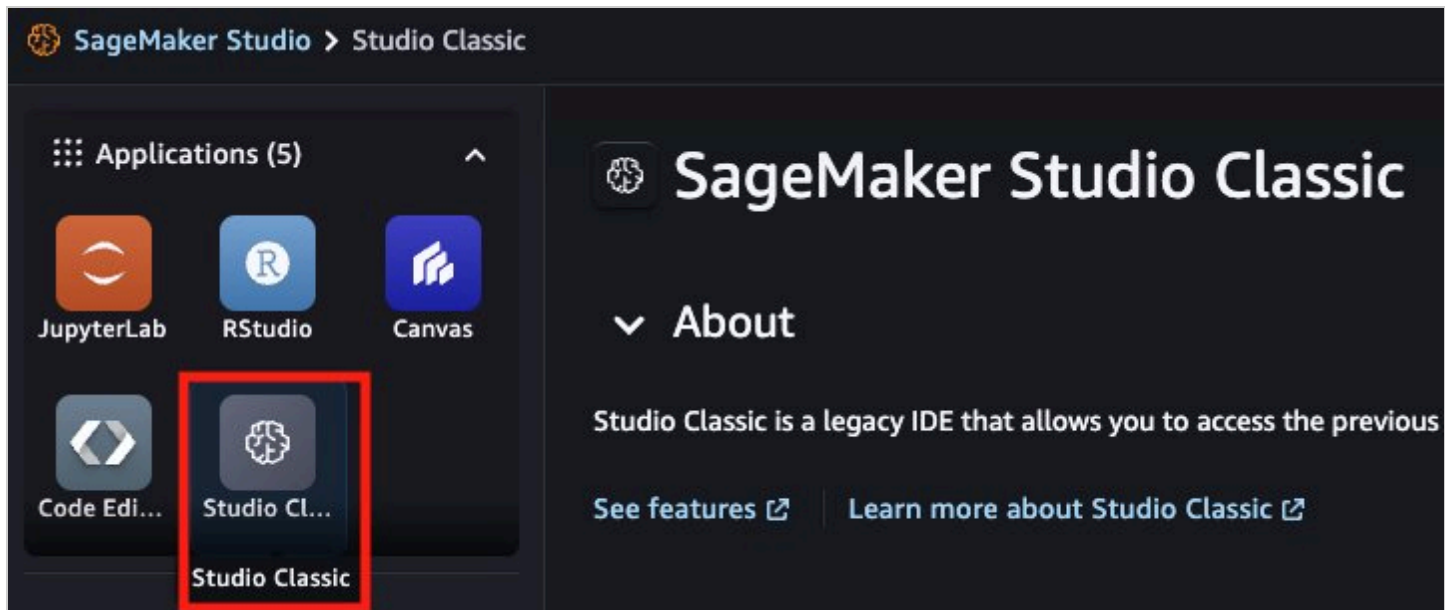
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "OpenSearchServeless",
      "Effect": "Allow",
      "Action": "aoss:*",
      "Resource": "*"
    }
  ]
}
```

4. On the SageMaker console, choose Studio in the navigation pane.

## 5. Choose your user profile and choose Open Studio.



This will open a new browser tab for SageMaker Studio Classic.



## 6. Run the SageMaker Studio application.

Name	Application	Status	Type	Last modified	Action
default-...	Studio Classic	Stopped	Private	0 seconds ago	Run

1 results   Results are cached   Refresh   Go to page 1   Page 1 of 1

## 7. When the application is running, choose Open.

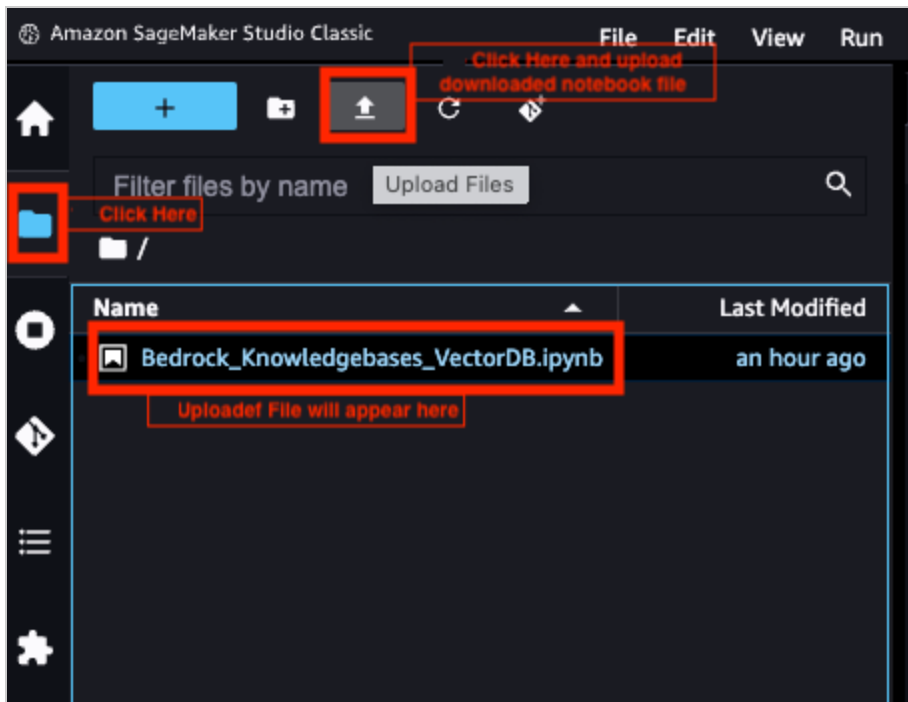
Name	Application	Status	Type	Last modified	Action
default-20240503t090598	Studio Classic	Running	Private	55 seconds ago	Stop   Open

1 results   Results are cached   Refresh   Go to page 1   Page 1 of 1

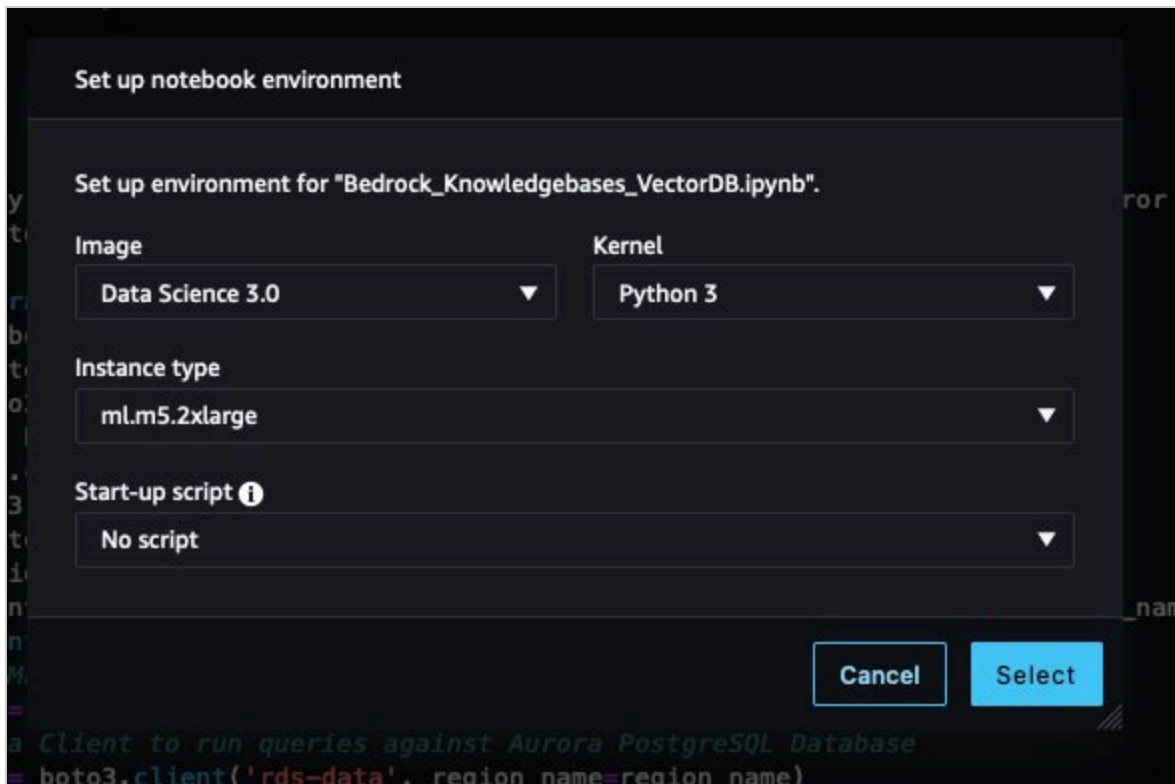
JupyterLab will open in a new tab.

8. Download the [notebook file](#) to use in this post.

9. Choose the file icon in the navigation pane, then choose the upload icon, and upload the notebook file.



10. Leave the image, kernel, and instance type as default and choose Select.



## Request Amazon Bedrock model access

Complete the following steps to request access to the embeddings model in Amazon Bedrock:

1. On the Amazon Bedrock console, choose **Model** access in the navigation pane.

2. Choose **Enable specific models**.
3. Select the **Titan Text Embeddings V2** model.
4. Choose **Next** and complete the access request.

**Amazon Bedrock** <

Amazon Bedrock > **Model access**

### What is Model access?

To use Bedrock, account users with the correct [IAM Permissions](#) must have the correct [Model Terms](#) for Bedrock FMs.

**Enable all models** **Enable specific models**

Visit [Amazon Bedrock Quotas](#) for a quick guide to the default quotas.

### Base models (33)

Not seeing a model you're interested in? Check out all supported models

Find model

Model	Access	Category	EULA
<input type="checkbox"/> Titan Multimodal Embeddings G1	Available to request	Embedding	<a href="#">EULA</a>
<input checked="" type="checkbox"/> <b>Titan Text Embeddings V2</b>	Available to request	Embedding	<a href="#">EULA</a>
<input type="checkbox"/> ▼ Anthropic (2)	0/2 access granted		
<input type="checkbox"/> Claude 3 Sonnet	Available to request	Text & Vision	<a href="#">EULA</a>
<input type="checkbox"/> Claude 3 Haiku	Available to request	Text & Vision	<a href="#">EULA</a>
<input type="checkbox"/> ▼ Cohere (2)	0/2 access granted		
<input type="checkbox"/> Embed English	Available to request	Embedding	<a href="#">EULA</a>
<input type="checkbox"/> Embed Multilingual	Available to request	Embedding	<a href="#">EULA</a>
<input type="checkbox"/> ▼ Meta (2)	0/2 access granted		
<input type="checkbox"/> Llama 3 8B Instruct	Available to request	Text	<a href="#">EULA</a>
<input type="checkbox"/> Llama 3 70B Instruct	Available to request	Text	<a href="#">EULA</a>
<input type="checkbox"/> ▼ Mistral AI (3)	0/3 access granted		
<input type="checkbox"/> Mistral 7B Instruct	Available to request	Text	<a href="#">EULA</a>
<input type="checkbox"/> Mixtral 8x7B Instruct	Available to request	Text	<a href="#">EULA</a>
<input type="checkbox"/> Mistral Large (24.02)	Available to request	Text	<a href="#">EULA</a>

Cancel **Next**

## Import dependencies

Open the notebook file `Bedrock_Knowledgebases_VectorDB.ipynb` and run Step 1 to import dependencies for this post and create Boto3 clients:

```
!pip install opensearch-py
!pip install retrying
```

```

from urllib.request import urlretrieve
import json
import os
import boto3
import random
import time
from opensearchpy import OpenSearch, RequestsHttpConnection, AWSV4SignerAuth, Request
credentials = boto3.Session().get_credentials()
service = 'aoss'
suffix = random.randrange(200, 900)
boto3_session = boto3.session.Session()
region_name = boto3_session.region_name
iam_client = boto3_session.client('iam')
account_number = boto3.client('sts').get_caller_identity().get('Account')
identity = boto3.client('sts').get_caller_identity()['Arn']

```

## Create an S3 bucket

You can use the following code to create an S3 bucket to store the source data for your vector database, or use an existing bucket. If you create a new bucket, make sure to follow your organization's best practices and guidelines.

```

if region_name in ('af-south-1', 'ap-east-1', 'ap-northeast-1', 'ap-northeast-2', 'ap-northeast-3', 'ap-south-1', 'ap-south-2', 'ap-southeast-1', 'ap-southeast-2', 'ap-southeast-3', 'eu-central-1', 'eu-central-2', 'eu-north-1', 'eu-south-1', 'eu-south-2', 'eu-west-1', 'eu-west-2', 'eu-west-3', 'me-south-1', 'sa-east-1'):
    # Create the bucket
    response = s3_client.create_bucket(
        Bucket=bucket_name,
        CreateBucketConfiguration={
            'LocationConstraint': region_name
        }
    )
    # Print the response and validate that value for HTTPStatusCode is 200
    print(response)
else:
    # Create the bucket
    response = s3_client.create_bucket(
        Bucket=bucket_name
    )
    # Print the response and validate that value for HTTPStatusCode is 200
    print(response)

```

## Set up sample data

Use the following code to set up the sample data for this post, which will be the input for the vector database:

```

# Create metadata file for 2019
metadata_2019_json = json.dumps(metadata_2019)

with open(f"{local_data_path}AMZN-2019-Shareholder-Letter.pdf.metadata.json", "w") as f:
    f.write(str(metadata_2019_json))

f.close()

# Upload files to Amazon S3
def uploadDirectory(path, bucket_name):
    for root, dirs, files in os.walk(path):
        for file in files:
            key = data_s3_prefix + '/' + file
            s3_client.upload_file(os.path.join(root, file), bucket_name, key)

uploadDirectory(local_data_path, data_s3_bucket)

# Delete files from local directory
!rm -r ./data/

```

## Configure the IAM role for Amazon Bedrock

Use the following code to define the function to create the IAM role for Amazon Bedrock, and the functions to attach policies related to [Amazon OpenSearch Service](#) and Aurora:

```

        "aws:ResourceAccount": f"{account_number}"
    }
}

]
}

assume_role_policy_document = {
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}

```

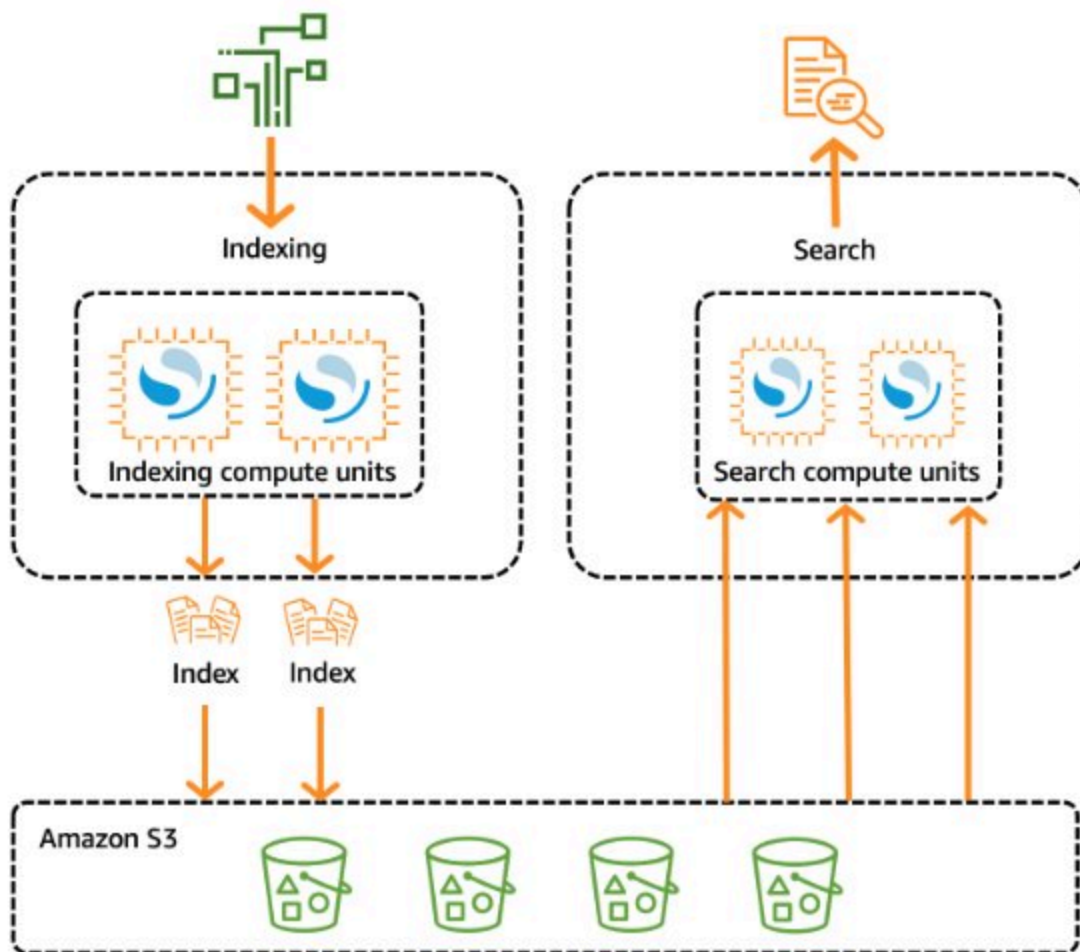
Use the following code to create the IAM role for Amazon Bedrock, which you'll use while creating the knowledge base:

```
bedrock_kb_execution_role = create_bedrock_execution_role(bucket_name=data_s3_bucket)
bedrock_kb_execution_role_arn = bedrock_kb_execution_role['Role']['Arn']
```

## Integrate with OpenSearch Serverless

The Vector Engine for Amazon OpenSearch Serverless is an on-demand [serverless](#) configuration for OpenSearch Service. Because it's serverless, it removes the operational complexities of provisioning, configuring, and tuning your OpenSearch clusters. With OpenSearch Serverless, you can search and analyze a large volume of data without having to worry about the underlying infrastructure and data management.

The following diagram illustrates the OpenSearch Serverless architecture. OpenSearch Serverless compute capacity for data ingestion, searching, and querying is measured in OpenSearch Compute Units (OCUs).



The vector search collection type in OpenSearch Serverless provides a similarity search capability that is scalable and high performing. This makes it a popular option for a vector database when using Amazon Bedrock Knowledge Bases, because it makes it straightforward to build modern machine learning (ML) augmented search experiences and generative AI applications without having to manage the underlying vector database infrastructure. Use cases

for OpenSearch Serverless vector search collections include image searches, document searches, music retrieval, product recommendations, video searches, location-based searches, fraud detection, and anomaly detection. The vector engine provides distance metrics such as Euclidean distance, cosine similarity, and dot product similarity. You can store fields with various data types for metadata, such as numbers, Booleans, dates, keywords, and geopoints. You can also store fields with text for descriptive information to add more context to stored vectors. Collocating the data types reduces complexity, increases maintainability, and avoids data duplication, version compatibility challenges, and licensing issues.

The following code snippets set up an OpenSearch Serverless vector database and integrate it with a knowledge base in Amazon Bedrock:

### 1. Create an OpenSearch Serverless vector collection.

```
# create security, network and data access policies within OSS
encryption_policy, network_policy, access_policy = create_policies_in_oss(vector_s
aoss_client=aoss_client,
bedrock_kb_execution_role_arn=bedrock_kb_execution_role_arn)

# Create OpenSearch Serverless Vector Collection
collection = aoss_client.create_collection(name=oss_vector_store_name, type='VECTOR

# Get the OpenSearch serverless collection URL
collection_id = collection['createCollectionDetail']['id']
host = collection_id + '.' + region_name + '.aoss.amazonaws.com'
print(host)

# wait for collection creation
# This can take couple of minutes to finish
response = aoss_client.batch_get_collection(names=[oss_vector_store_name])
# Periodically check collection status
while (response['collectionDetails'][0]['status']) == 'CREATING':
```

### 2. Create an index in the collection; this index will be managed by Amazon Bedrock Knowledge Bases:

```
body_json = {
    "settings": {
        "index.knn": "true",
        "number_of_shards": 1,
        "knn.algo_param.ef_search": 512,
        "number_of_replicas": 0,
    },
    "mappings": {
        "properties": {
```

```

"vector": {
  "type": "knn_vector",
  "dimension": 1024,
  "method": {
    "name": "hnsw",
    "engine": "faiss",
    "space_type": "l2"
  },
},
}

```

### 3. Create a knowledge base in Amazon Bedrock pointing to the OpenSearch Serverless vector collection and index

```

opensearchServerlessConfiguration = {
  "collectionArn": collection["createCollectionDetail"]['arn'],
  "vectorIndexName": oss_index_name,
  "fieldMapping": {
    "vectorField": "vector",
    "textField": "text",
    "metadataField": "text-metadata"
  }
}

```

# The embedding model used by Bedrock to embed ingested documents, and realtime pr  
embeddingModelArn = f"arn:aws:bedrock:{region\_name}::foundation-model/amazon.titan

```

name = f"kb-os-shareholder-letter-{suffix}"
description = "Amazon shareholder letter knowledge base."
roleArn = bedrock_kb_execution_role_arn

```

# Create a KnowledgeBase

### 4. Create a data source for the knowledge base:

```

# Ingest strategy - How to ingest data from the data source
chunkingStrategyConfiguration = {
  "chunkingStrategy": "FIXED_SIZE",
  "fixedSizeChunkingConfiguration": {
    "maxTokens": 512,
    "overlapPercentage": 20
  }
}

```

# The data source to ingest documents from, into the OpenSearch serverless knowled

```
s3Configuration = {
    "bucketArn": f"arn:aws:s3:::{data_s3_bucket}",
    "inclusionPrefixes": [f"{data_s3_prefix}"] # you can use this if you want to c
}
# Create a DataSource in KnowledgeBase
create_ds_response = bedrock_agent_client.create_data_source(
    name = f'{name}-{bucket_name}',
```

5. Start an ingestion job for the knowledge base pointing to OpenSearch Serverless to generate vector embeddings for data in Amazon S3:

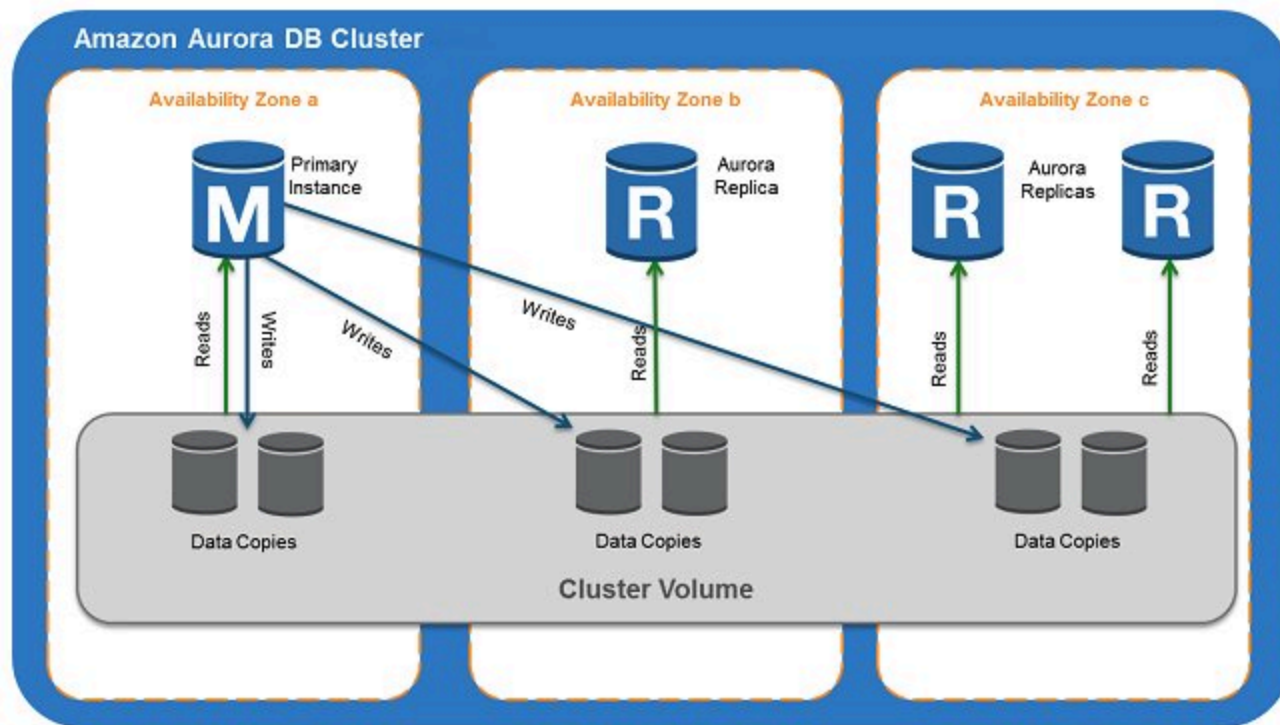
```
ingest_jobs=[]
# Start an ingestion job
try:
    start_job_response = bedrock_agent_client.start_ingestion_job(knowledgeBaseId
    job = start_job_response["ingestionJob"]
    print(f"ingestion job started successfully\n")

    while(job['status'] != 'COMPLETE' ):
        get_job_response = bedrock_agent_client.get_ingestion_job(
            knowledgeBaseId = kb['knowledgeBaseId'],
            dataSourceId = ds["dataSourceId"],
            ingestionJobId = job["ingestionJobId"]
        )
        job = get_job_response["ingestionJob"]

    time.sleep(30)
    print(f"job completed successfully\n")
```

## Integrate with Aurora pgvector

Aurora provides [pgvector](#) integration, which is an open source extension for PostgreSQL that adds the ability to store and search over ML-generated vector embeddings. This enables you to use Aurora for generative AI RAG-based use cases by storing vectors with the rest of the data. The following diagram illustrates the sample architecture.



Use cases for Aurora pgvector include applications that have requirements for ACID compliance, point-in-time recovery, joins, and more. The following is a sample code snippet to configure Aurora with your knowledge base in Amazon Bedrock:

1. Create an Aurora DB instance (this code creates a managed DB instance, but you can create a serverless instance as well). Identify the security group ID and subnet IDs for your VPC before running the following step and provide the appropriate values in the `vpc_security_group_ids` and `SubnetIds` variables:

```
# Define database instance parameters
db_instance_identifier = aurora_vector_db_instance
db_cluster_identifier = aurora_vector_db_cluster
engine = 'aurora-postgresql'
db_name = aurora_database_name
db_instance_class = 'db.r6g.2xlarge'
master_username = 'postgres'
# Get Security Group Id(s), for replicating Blogpost steps it can be one associated
vpc_security_group_ids = ['sg-XXXXXXX']
subnet_group_name = 'vectordbsubnetgroup'

response = rds.create_db_subnet_group(
    DBSubnetGroupName=subnet_group_name,
    DBSubnetGroupDescription='Subnet Group for Blogpost Aurora PostgreSQL Database
# Get Subnet IDs, for replicating Blogpost steps it can be one associated with
SubnetIds=[
    'subnet-XXXXXXX',
    'subnet-XXXXXXX',
```

- On the Amazon RDS console, confirm the Aurora database status shows as Available.

**Databases (2)**

Filter by databases

	DB identifier	Status	Role
<input type="radio"/>	<a href="#">aurora-shareholder-letter-<span style="background-color: black; color: black;">XXXXXXXXXX</span></a>	Available	Regional cluster
<input type="radio"/>	<a href="#">aurora-shareholder-letter-instance-<span style="background-color: black; color: black;">XXXXXXXXXX</span></a>	Available	Writer instance

- Create the vector extension, schema, and vector table in the Aurora database:

```
##Get Amazon Aurora Database Secret Manager ARN created internally while creating
describe_db_clusters_response = rds.describe_db_clusters(
    DBClusterIdentifier=db_cluster_identifier,
    IncludeShared=False
)

aurora_db_secret_arn = describe_db_clusters_response['DBClusters'][0]['MasterUsersSecretArn']
db_cluster_arn = describe_db_clusters_response['DBClusters'][0]['DBClusterArn']

# Enable HTTP Endpoint for Amazon Aurora Database instance
response = rds.enable_http_endpoint(
    ResourceArn=db_cluster_arn
)

# Create Vector Extension in Aurora PostgreSQL Database which will be used in table
vector_extension_create_response = rds_data_client.execute_statement(
    resourceArn=db_cluster_arn,
```

- Create a knowledge base in Amazon Bedrock pointing to the Aurora database and table:

```
# Attached RDS related permissions to the Bedrock Knowledgebase role

create_rds_policy_attach_bedrock_execution_role(db_cluster_arn, aurora_db_secret_arn)

# Define RDS Configuration for Knowledge bases
rdsConfiguration = {
```

```

'credentialsSecretArn': aurora_db_secret_arn,
'databaseName': db_name,
'fieldMapping': {
    'metadataField': 'metadata',
    'primaryKeyField': 'id',
    'textField': 'chunks',
    'vectorField': 'embedding'
},
'resourceArn': db_cluster_arn,
'tableName': 'bedrock_integration.share_holder_letter_kb'
}

```

## 5. Create a data source for the knowledge base:

```

# Ingest strategy - How to ingest data from the data source
chunkingStrategyConfiguration = {
    "chunkingStrategy": "FIXED_SIZE",
    "fixedSizeChunkingConfiguration": {
        "maxTokens": 512,
        "overlapPercentage": 20
    }
}

# The data source to ingest documents from, into the OpenSearch serverless knowledge base
s3Configuration = {
    "bucketArn": f"arn:aws:s3:::{data_s3_bucket}",
    "inclusionPrefixes": [f"{data_s3_prefix}"] # you can use this if you want to c
}

# Create a DataSource in KnowledgeBase
create_ds_response = bedrock_agent_client.create_data_source(
    name = f'{name}-{data_s3_bucket}',
    description = description,

```

## 6. Start an ingestion job for your knowledge base pointing to the Aurora pgvector table to generate vector embeddings for data in Amazon S3:

```

ingest_jobs=[]
# Start an ingestion job
try:
    start_job_response = bedrock_agent_client.start_ingestion_job(knowledgeBaseId)
    job = start_job_response["ingestionJob"]
    print(f"job started successfully\n")

```

```
while(job['status']!='COMPLETE' ):
    get_job_response = bedrock_agent_client.get_ingestion_job(
        knowledgeBaseId = kb['knowledgeBaseId'],
        dataSourceId = ds["dataSourceId"],
        ingestionJobId = job["ingestionJobId"]
    )
    job = get_job_response["ingestionJob"]

time.sleep(30)
print(f"job completed successfully\n")
```

## Integrate with MongoDB Atlas

MongoDB Atlas Vector Search, when integrated with Amazon Bedrock, can serve as a robust and scalable knowledge base to build generative AI applications and implement RAG workflows. By using the flexible document data model of MongoDB Atlas, organizations can represent and query complex knowledge entities and their relationships with Amazon Bedrock. The combination of MongoDB Atlas and Amazon Bedrock provides a powerful solution for building and maintaining a centralized knowledge repository.

To use MongoDB, you can create a cluster and vector search index. The native vector search capabilities embedded in an operational database simplify building sophisticated RAG implementations. MongoDB allows you to store, index, and query vector embeddings of your data without the need for a separate bolt-on vector database.

There are three pricing options available for MongoDB Atlas through [AWS Marketplace: MongoDB Atlas \(pay-as-you-go\)](#), [MongoDB Atlas Enterprise](#), and [MongoDB Atlas for Government](#). Refer to the [MongoDB Atlas Vector Search documentation](#) to set up a MongoDB vector database and add it to your knowledge base.

## Integrate with Pinecone

Pinecone is a type of vector database from [Pinecone Systems Inc.](#) With Amazon Bedrock Knowledge Bases, you can integrate your enterprise data into Amazon Bedrock using Pinecone as the fully managed vector database to build generative AI applications. Pinecone is highly performant; it can speed through data in milliseconds. You can use its metadata filters and sparse-dense index support for top-notch relevance, achieving quick, accurate, and grounded results across diverse search tasks. Pinecone is enterprise ready; you can launch and scale your AI solution without needing to maintain infrastructure, monitor services, or troubleshoot algorithms. Pinecone adheres to the security and operational requirements of enterprises.

There are two pricing options available for Pinecone in AWS Marketplace: [Pinecone Vector Database – Pay As You Go Pricing](#) (serverless) and [Pinecone Vector Database – Annual Commit](#) (managed). Refer to the [Pinecone documentation](#) to set up a Pinecone vector database and add it to your knowledge base.

## Integrate with Redis Enterprise Cloud

Redis Enterprise Cloud enables you to set up, manage, and scale a distributed in-memory data store or cache environment in the cloud to help applications meet low latency requirements. Vector search is one of the solution options available in Redis Enterprise Cloud, which solves for low latency use cases related to RAG, semantic caching, document search, and more. Amazon Bedrock natively integrates with Redis Enterprise Cloud vector search.

There are two pricing options available for Redis Enterprise Cloud through AWS Marketplace: [Redis Cloud Pay As You Go Pricing](#) and [Redis Cloud – Annual Commits](#). Refer to the [Redis Enterprise Cloud documentation](#) to set up vector search and add it to your knowledge base.

## Interact with Amazon Bedrock knowledge bases

Amazon Bedrock provides a common set of APIs to interact with knowledge bases:

- **Retrieve API** – Queries the knowledge base and retrieves information from it. This is a Bedrock Knowledge Base specific API, it helps with use cases where only vector-based searching of documents is needed without model inferences.
- **Retrieve and Generate API** – Queries the knowledge base and uses an LLM to generate responses based on the retrieved results.

The following code snippets show how to use the Retrieve API from the OpenSearch Serverless vector database's index and the Aurora pgvector table:

1. Retrieve data from the OpenSearch Serverless vector database's index:

```
query = "What is Amazon's doing in the field of generative AI?"

relevant_documents_os = bedrock_agent_runtime_client.retrieve(
    retrievalQuery= {
        'text': query
    },
    knowledgeBaseId=kb['knowledgeBaseId'],
    retrievalConfiguration= {
        'vectorSearchConfiguration': {
            'numberOfResults': 3 # will fetch top 3 documents which matches closely
        }
    }
)
relevant_documents_os["retrievalResults"]
```

2. Retrieve data from the Aurora pgvector table:

```
query = "What is Amazon's doing in the field of generative AI?"

relevant_documents_rds = bedrock_agent_runtime_client.retrieve(
    retrievalQuery= {
        'text': query
    },
    knowledgeBaseId=rds_kb['knowledgeBaseId'],
    retrievalConfiguration= {
        'vectorSearchConfiguration': {
            'numberOfResults': 3 # will fetch top 3 documents which matches closely
        }
    }
)

relevant_documents_rds["retrievalResults"]
```

## Clean up

When you're done with this solution, clean up the resources you created:

- Amazon Bedrock knowledge bases for OpenSearch Serverless and Aurora
- OpenSearch Serverless collection
- Aurora DB instance
- S3 bucket
- SageMaker Studio domain
- Amazon Bedrock service role
- SageMaker Studio domain role

## Conclusion

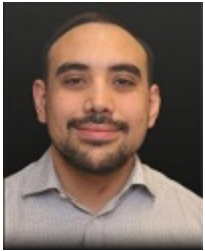
In this post, we provided a high-level introduction to generative AI use cases and the use of RAG workflows to augment your organization's internal or external knowledge stores. We discussed the importance of vector databases and RAG architectures to enable similarity search and why dense vector representations are beneficial. We also went over Amazon Bedrock Knowledge Bases, which provides common APIs, industry-leading governance, observability, and security to enable vector databases using different options like AWS native and partner products through AWS Marketplace. We also dived deep into a few of the vector database options with code examples to explain the implementation steps.

Try out the code examples in this post to implement your own RAG solution using Amazon Bedrock Knowledge Bases, and share your feedback and questions in the comments section.

## About the Authors



**Vishwa Gupta** is a Senior Data Architect with AWS Professional Services. He helps customers implement generative AI, machine learning, and analytics solutions. Outside of work, he enjoys spending time with family, traveling, and trying new foods.



**Isaac Privitera** is a Principal Data Scientist with the AWS Generative AI Innovation Center, where he develops bespoke generative AI-based solutions to address customers' business problems. His primary focus lies in building responsible AI systems, using techniques such as RAG, multi-agent systems, and model fine-tuning. When not immersed in the world of AI, Isaac can be found on the golf course, enjoying a football game, or hiking trails with his loyal canine companion, Barry.



**Abhishek Madan** is a Senior GenAI Strategist with the AWS Generative AI Innovation Center. He helps internal teams and customers in scaling generative AI, machine learning, and analytics solutions. Outside of work, he enjoys playing adventure sports and spending time with family.



**Ginni Malik** is a Senior Data & ML Engineer with AWS Professional Services. She assists customers by architecting enterprise data lake and ML solutions to scale their data analytics in the cloud.



**Satish Sarapuri** is a Sr. Data Architect, Data Lake at AWS. He helps enterprise-level customers build high-performance, highly available, cost-effective, resilient, and secure generative AI, data mesh, data lake, and analytics platform solutions on AWS through which customers can make data-driven decisions to gain impactful outcomes for their business, and helps them on their digital and data transformation journey. In his spare time, he enjoys spending time with his family and playing tennis.



Like



Share

---

## Comments

[Log in to comment](#)

