Knowledge Bases and RAG

Overview

One of the key issues with LLM's is their ability to hallucinate, so returning answers grounded in facts becomes a top priority for question answering (QA), Contextual Chatbot and other use-cases. QA is an important task that involves extracting answers to factual queries posed in natural language. Typically, a QA system processes a query against a knowledge base containing structured or unstructured data and generates a response with accurate information. Ensuring high accuracy is key to developing a useful, reliable and trustworthy question answering system, especially for enterprise use cases.

Generative AI models like Titan, Claude, and Jurassic use probability distributions to generate responses to questions. These models are trained on vast amounts of text data, which allows them to predict what comes next in a sequence or what word might follow a particular word. However, these models are not able to provide accurate or deterministic answers to every question because there is always some degree of uncertainty in the data.

In this module, you will walk you through how to implement the QA pattern with Bedrock. Additionally, you will prepare embeddings to be loaded in the vector database (in this example, in notebook memory).

Relevance

Organisations need to query domain specific and proprietary data and use the information to answer questions, including data on which the model has not been trained. The challenge here is the limit on the amount of contextual information used in a given scenario, because models have a limit on the size of the prompt they can handle.

You can overcome this challange using the Retrieval Augmented Generation (RAG) technique. RAG combines using embeddings to index the corpus of documents to build a knowledge base and using a large language model to extract information from a subset of documents in the knowledge base.

As a preparation step for RAG, the documents comprising the knowledge base are split into chunks of a fixed size and passed to an embedding model to obtain the embedding vector. The size of each chunk is limited by the maximum input size of the chosen embedding model. The embedding together with the original chunk of the document and additional metadata are stored in a vector database. The vector database is optimized to efficiently perform similarity searches between vectors.

Target Audience

- Customers with data stores that may be private or frequently changing
- Customer service agents
- Anyone who needs to improve question search with accurate or specific documentation

Challenges

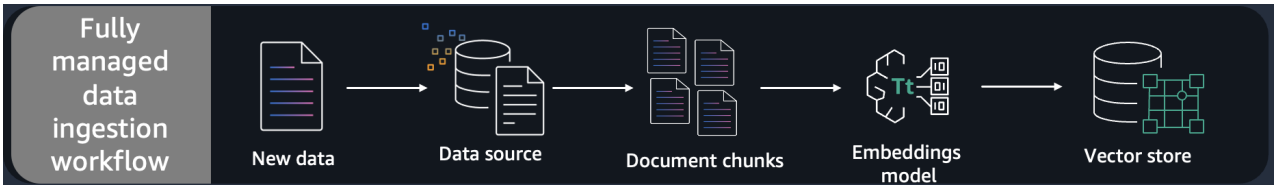
The challenges are getting relevant answers from a trained model and avoiding hallucination, as well as dealing with large amounts of data and model context limitations.

Introduction to Knowledge Bases for Amazon Bedrock

Knowledge Bases for Amazon Bedrock With Knowledge Bases for Amazon Bedrock, you can give FMs and agents contextual information from your company's private data sources for Retrieval Augmented Generation (RAG) to deliver more relevant, accurate, and customized responses

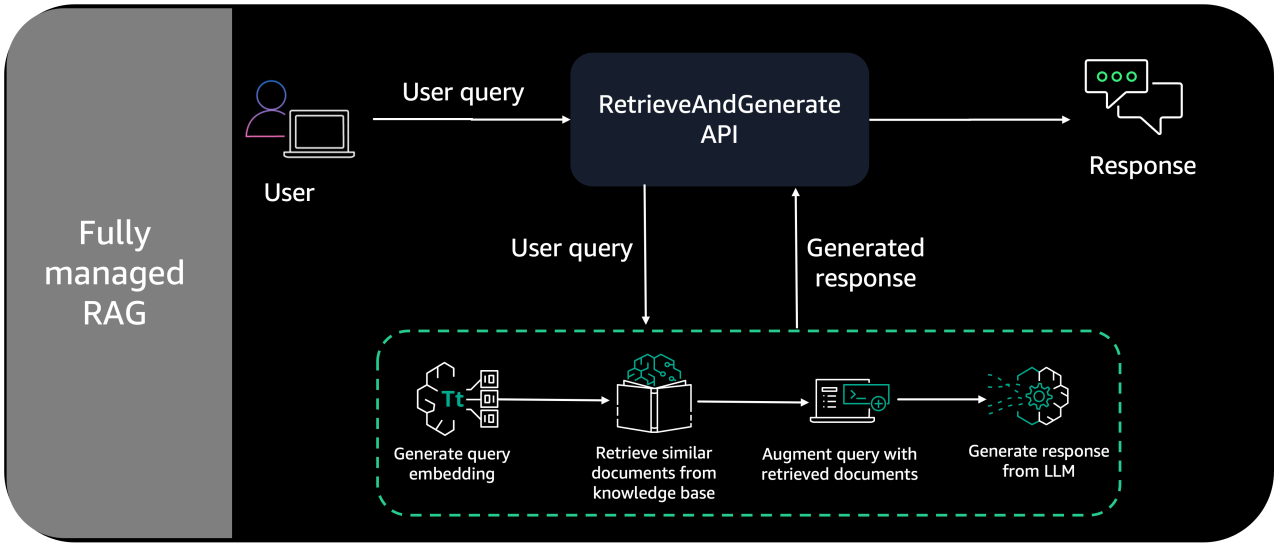
Knowledge Bases for Amazon Bedrock is a fully managed capability that helps you implement the entire RAG workflow from ingestion to retrieval and prompt augmentation without having to build custom integrations to data sources and manage data flows. Session context management is built in, so your app can readily support multi-turn conversations.

This example implements a knowledge base by specifying a data source such as Amazon S3, select an embedding model, such as Amazon Titan Embeddings to convert the data into vector embeddings, and a destination vector database such as Amazon OpenSearch Service Serverless (AOSS) to store the vector data. Bedrock takes care of creating, storing, managing, and updating your embeddings in the vector database.

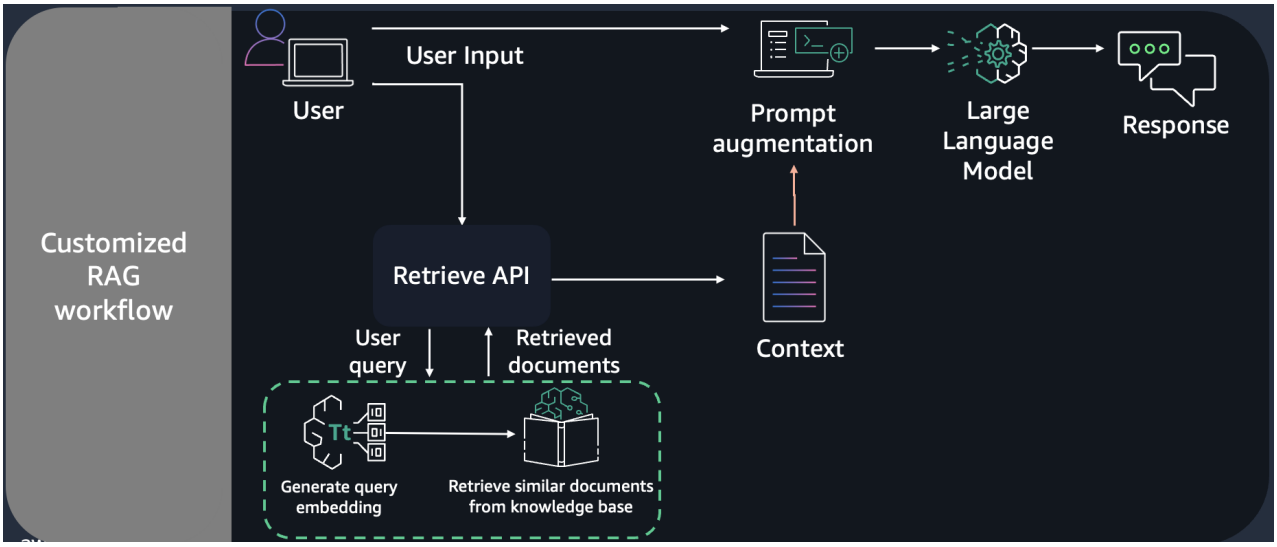


This example implements the RAG architectural pattern. RAG retrieves data from outside the language model (non-parametric) and augments prompts by adding relevant retrieved data as context. Here, you are performing RAG effectively on the knowledge base created previously or using console.

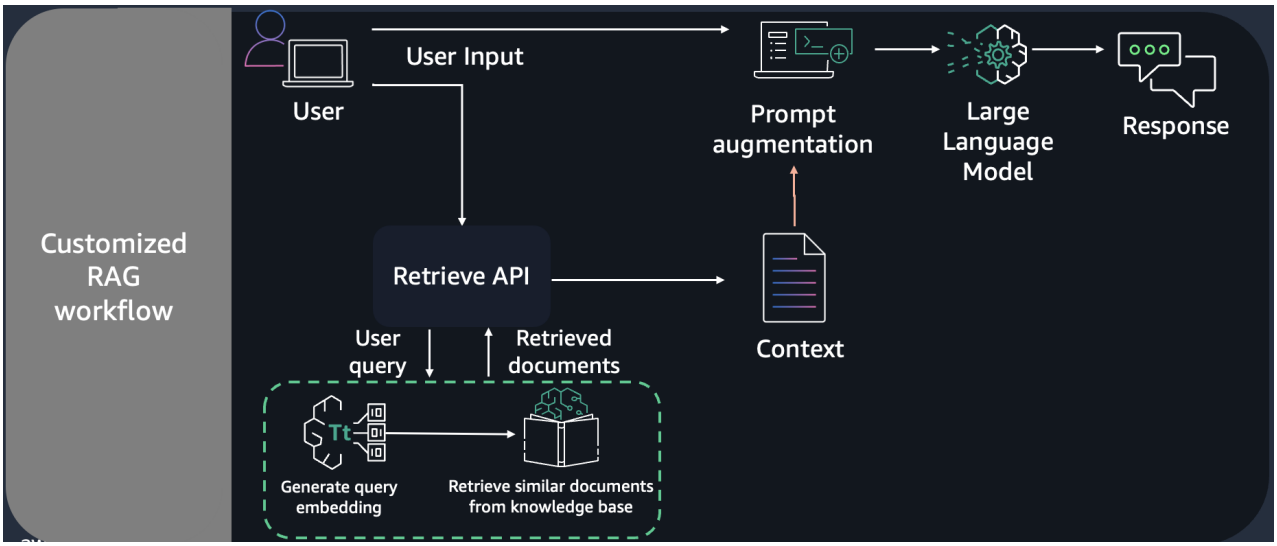
In first architectural pattern, you will use the RetreiveAndGenerate API provided by Knowledge Bases for Amazon Bedrock which converts user queries into embeddings, searches the knowledge base, get the relevant results, augment the prompt and then invoking a LLM to generate the response.



In second pattern architecture, you will use the Retreive API provided by Knowledge Bases for Amazon Bedrock which converts user queries into embeddings, searches the knowledge base, get the relevant results, giving you more control to build custom workflows on top of the semantic search results. The output of the Retrieve API includes the the retrieved text chunks, the location type and URI of the source data, as well as the relevance scores of the retrievals. You will then use the text chunks being generated and augment it with the original prompt and pass it through the anthropic.claude-v2 model using prompt engineering patterns based on your use case.



You will also demonstrate the use of open source tools like Lang Chain to implement the RAG pattern. RAG retrieves data from outside the language model (non-parametric) and augments prompts by adding relevant retrieved data as context. Here, you are performing RAG effectively on the knowledge base created previously or using console/sdk. In this architecture, you will use the Retreive API provided by Knowledge Bases for Amazon Bedrock which converts user queries into embeddings, searches the knowledge base, get the relevant results, giving you more control to build custom workflows on top of the semantic search results. The output of the Retrieve API includes the the retrieved text chunks, the location type and URI of the source data, as well as the relevance scores of the retrievals. You will then use the RetrievalQAChain provided by LangChain, add RetreiverAPI as a retriever in the chain. This chain will then automatically augment the text chunks being generated with the original prompt and pass it through the anthropic.claude-instant-v1 model.



Sub-patterns

In this lab, you will explore multiple **RAG** patterns:

1. Ingest and Retrieve Pattern - The **first** example will show how to create and ingest data using OpenSearch and then leverage RetrieveAndGenerate API of KnowledgeBase to search the embeddings and return contextual results. Behind the scenes, RetrieveAndGenerate API converts queries into embeddings, searches the knowledge base, and then augments the foundation model prompt with the search results as context information and returns the FM-generated response to the question. For multi-turn conversations, Knowledge Bases manage short-term memory of the conversation to provide more contextual results.
2. Citations and Retrieve API provided by Knowledge Bases - The **Second** pattern shows how to use Retrieve API provided by Knowledge Bases for Amazon Bedrock. Note that `citation` is extremely important to validate `faithfulness` and `relevancy` of the vector databases. The LLM can use the citation in conjunction with the right prompt template to generate response that is grounded in the facts provided by KnowledgeBase.

3. Question Answering (Q&A) with Retrieve API provided by Knowledge Bases - The **Third** pattern shows how to build question answering applications using Retrieve API provided by Knowledge Bases for Amazon Bedrock. The LLM can leverage this API to gather relevant context, which, combined with an appropriate prompt template, allows generatation of responses grounded in the facts sourced from the KnowledgeBase.
4. Question Answering with Retrieve API provided by Knowledge Bases with Claude V2 - This pattern will show the same pattern as above but with Claude V2, which needs a different prompt template.

> **Important**
> If you are running this workshop in an AWS created event, you will not have access to the Claude V3 models

Previous Next