Workshop 2: Gen AI Amazon Q

Welcome! This workshop is designed to introduce generative AI concepts to Nonprofit organizations through hands-on exercises.
This workshop is for technical audiences working with Nonprofit organizations. Technical audiences could include architects, software developers, product managers, and anyone else interested in building generative AI applications. While a background in software development or AWS services can be helpful, it is not required for this workshop. No data science experience is required, either.
By the end of the labs, you will have a good foundation for building Generative AI chatbots using Amazon Q for Business, Amazon Bedrock, and Amazon Lex.

Generative AI and Nonprofits

Recent innovations in generative AI (GenAI) have made cutting-edge technology more affordable and accessible than ever for nonprofit organizations. In addition, the widespread adoption of generative AI tools has introduced a broad range of use cases that help nonprofits advance their missions, gain more valuable insights from their data, and automate repetitive tasks. Some of the most common use cases include:

- Public-facing chatbots to deliver information about an organization conversationally.
- Internal chatbots to help employees quickly extract meaningful insights from your organization's data without having to write code or database queries. This could include donor and outreach data, volunteer data, budgeting, reporting, strategy and more.
- Multi-media content generation for your outreach that is customized to match your organization's tone and style.

With Amazon Bedrock's pay-as-you-go pricing model, nonprofits pay only for what they actually use. This means that even the smallest organizations can start leveraging generative AI today and seamlessly scale to an enterprise level when needed.

Workshop Overview

This workshop includes low to no-code, as well as full-code, approaches to developing GenAI applications on AWS. The primary use case developed throughout this workshop is a chatbot for interfacing with large language models (LLMs) that accesses custom datasets to answer questions.
Throughout the workshop we will be developing a GenAI application for a fictional food bank.
The workshop is split into three sections, each with low to full-code approaches for working with GenAI chatbots.

1. Getting started with GenAI

    - No-code: Amazon Q for Business
    - Full-code: Amazon Bedrock with langChain and streamlit
2. Working with vector stores
    - Low-code: Knowledge Bases for Amazon Bedrock , Amazon Kendra
    - Full-code: Amazon Opensearch Serverless
3. Hosting a front end for GenAI
    - No-code: Amazon Lex with built-in QnAIntent
    - Full-code: Amazon Lex with AWS Lambda and Amazon bedrock

Overview of Amazon Bedrock and Streamlit

Amazon Bedrock

is a fully managed service for using foundation models. It allows you to access models from Amazon and third parties with a single set of APIs for both text generation and image generation.

Streamlit

allows us to quickly create web front ends for our Python code, without needing front-end development skills. Streamlit is great for creating proofs-of-concepts that can be presented to a wide audience of both technical and non-technical people.

Supported Regions

This workshop is primarily hosted in us-west-2 (Oregon) with Amazon Q for Business components hosed in us-east-1 (N. Virginia) to avoid capacity conflicts at large events.

Time to complete

1. Getting started with GenAI: 1 hour
2. Working with vector stores: 1-2 hours
3. Hosting a front end for GenAI: 1 hour

Pursuing all the low-code options within each section will take around 2 hours total.
Pursuing all the full-code options within each section will take 2-3 hours total.

Navigating the workshop

	Getting started	Working with vector stores	Hosting a front end
No-code	Amazon Q for Business		Amazon Lex integration with Amazon Kendra or Bedrock Knowledge Bases
Low-code		- RAG chatbot with Amazon Kendra	
		- RAG chatbot with Bedrock Knowledge Bases	
Full-code	Streamlit RAG chatbot	RAG Chatbot with Amazon OpenSearch Serverless	Amazon Lex with AWS Lambda fulfillment

Table of contents

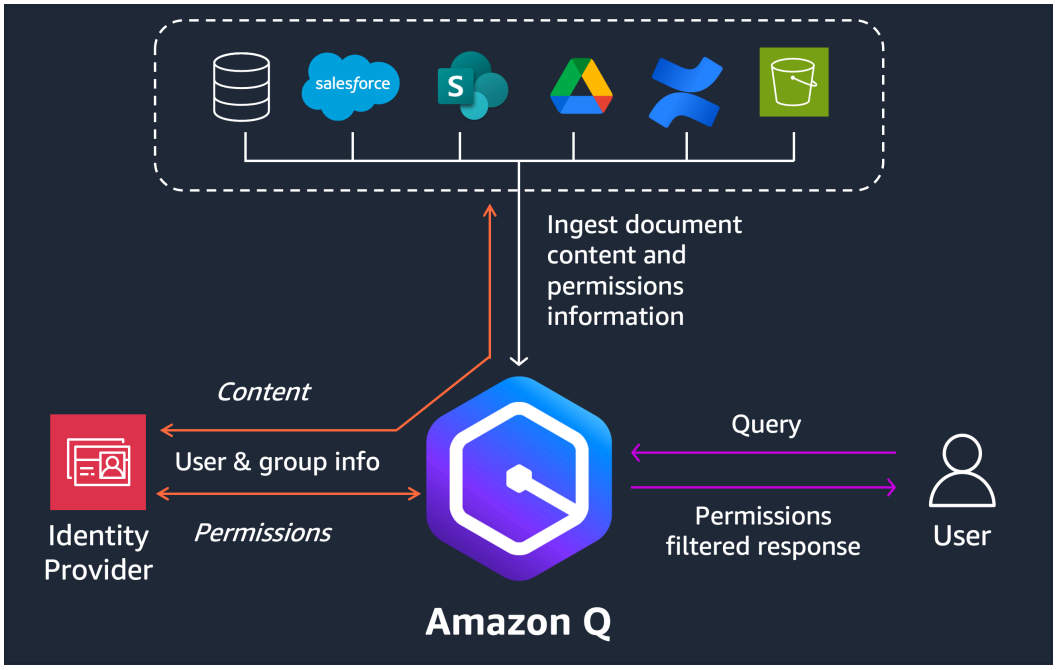
Amazon Q Business

Overview

Amazon Q Business application is a generative AI assistant that boosts employee productivity and transforms the way that you get work done. You can

quickly find accurate answers from your enterprise content, backed up with citations and references. You can brainstorm new ideas, generate content, or create summaries using Amazon Q application based on your enterprise data. Amazon Q application ensures that users access enterprise content securely according to their permissions. You can quickly deploy Amazon Q application with built-in connectors to popular enterprise repositories.

High-level architecture of Amazon Q



Control flow of a user request in Amazon Q application

1. User makes a request.
2. Amazon Q application retrieves semantically most relevant information to the user's request from the ingested enterprise content, to which the user has access permissions.
3. Amazon Q sends the user request along with the retrieved information as context to the underlying large language model (LLM).
4. LLM returns a succinct response to the user’s request based on the context.
5. The response from LLM is sent back to the user.

Steps to configure and deploy an Amazon Q application

With Amazon Q application, you create an Application, configure a Retriever and then use the Preview Web Experience to interact with your application from the console, or deploy a web experience using SAML 2.0 compliant identity provider of your choice such as Okta, and provide the URL to your users, so that they can authenticate and use the Amazon Q application you just created.

1. **Create an Application:** You can use the step by step workflow in the AWS Management Console for Amazon Q application to start creating an application.
2. **Customize Web Experience:** The step by step workflow to create an application will also guide you to provide customization for your Web

Experience.

3. **Configure Retriever:** The next step is to make a choice about whether you are planning to use the native retriever provided by Amazon Q or you plan to use a pre-existing Amazon Kendra index, in the same AWS region as the Amazon Q application, with Kendra retriever.
    - **Configure native retriever:** In this step, you can simply upload documents from your local machine to Amazon Q application, configure a data source with sample documents, or use one of the native data source connectors to connect to the data sources where the enterprise documents are located. If you configured a data source connector, initiate data source sync to index the documents from the data source.
    - **Configure Kendra retriever:** In this step, you configure the existing Kendra index as a retriever.
4. **Deploy web experience:** In this step you will deploy the web experience for your Amazon Q application using SAML 2.0 compliant identity provider (IdP) and provide the URL to your users, so that they can authenticate and use the Amazon Q application you just created.

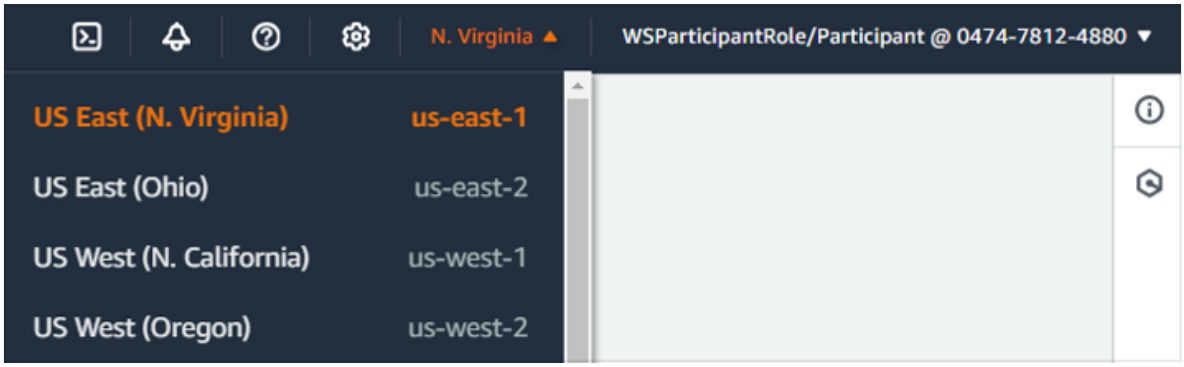
What you will build today

In this lab, you will configure an Amazon Q Business application, use its native retriever to to ingest data from Amazon S3, and deploy the web experience for this application using AWS IAM Identity Center (IDC) as identity provider (IdP). You will then use the deployed web experience logged in as a test user to experience how the responses are generated securely, based only on the content that has been ingested from Amazon S3.

Before Getting Started

Important - Region Selection

Ensure that you're in the us-east-1 region (N. Virginia) when setting up IAM Identity Center.



Enabling AWS IAM Identity Center

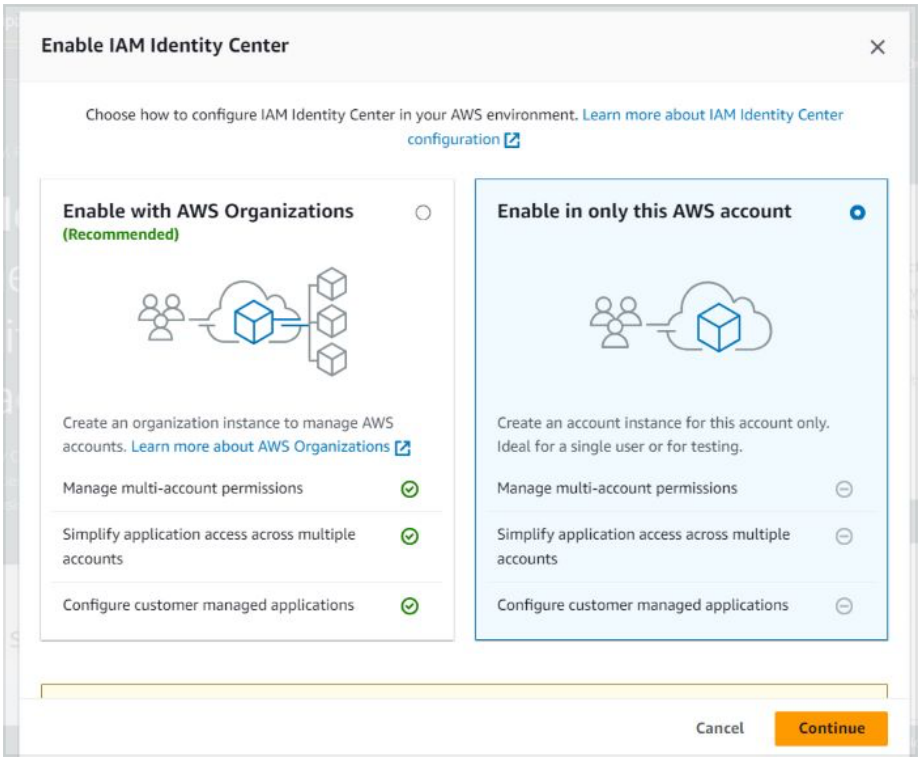
1. Open the AWS Management Console for IAM Identity Center

2. Select **Enable** from the landing page

3.

Are you in the right region?
Ensure that you're in the us-east-1 (N. Virginia) region before proceeding.

3. Choose **Enable in only this AWS account** and select **Continue**



4.

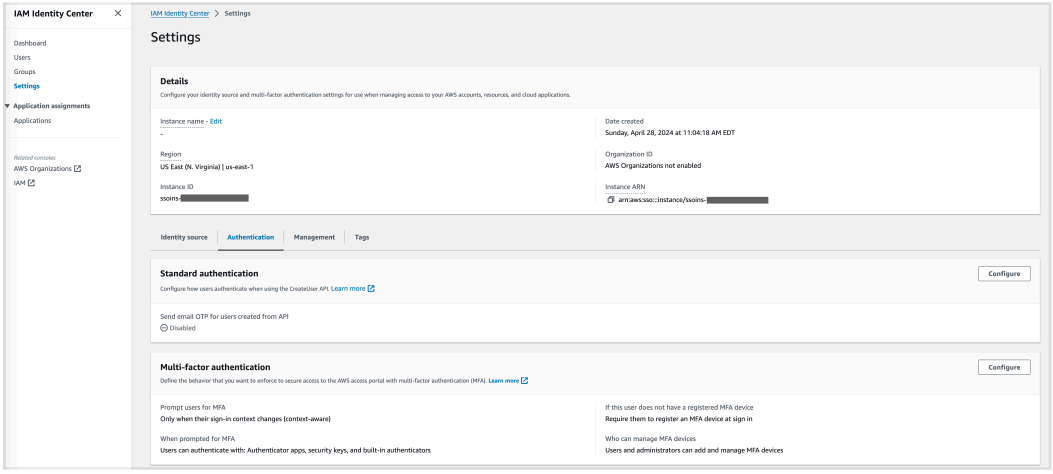
5. Do not add any new tags and choose **Enable** to set up IAM Identity Center

Did you accidentally enable in us-west-2?

Disable MFA Authentication

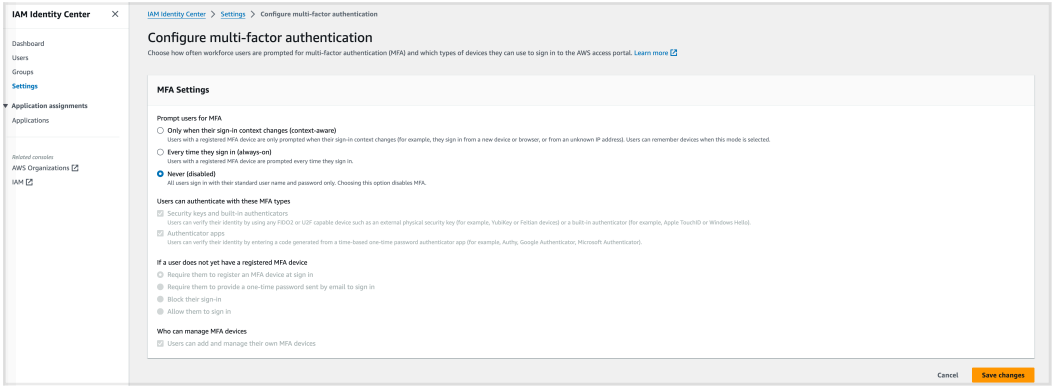
We strongly recommend that you adhere to the best practice of enabling MFA for user authentication in your own AWS accounts. For this workshop, we choose to disable MFA to simplify the process.

1. To disable MFA, navigate to **Settings**, select **Authentication** tab.



2.

3. Under **Multi-factor authentication**, select **Configure**. Under **Prompt users for MFA**, select **Never (disabled)**, and click **Save changes**.



4.

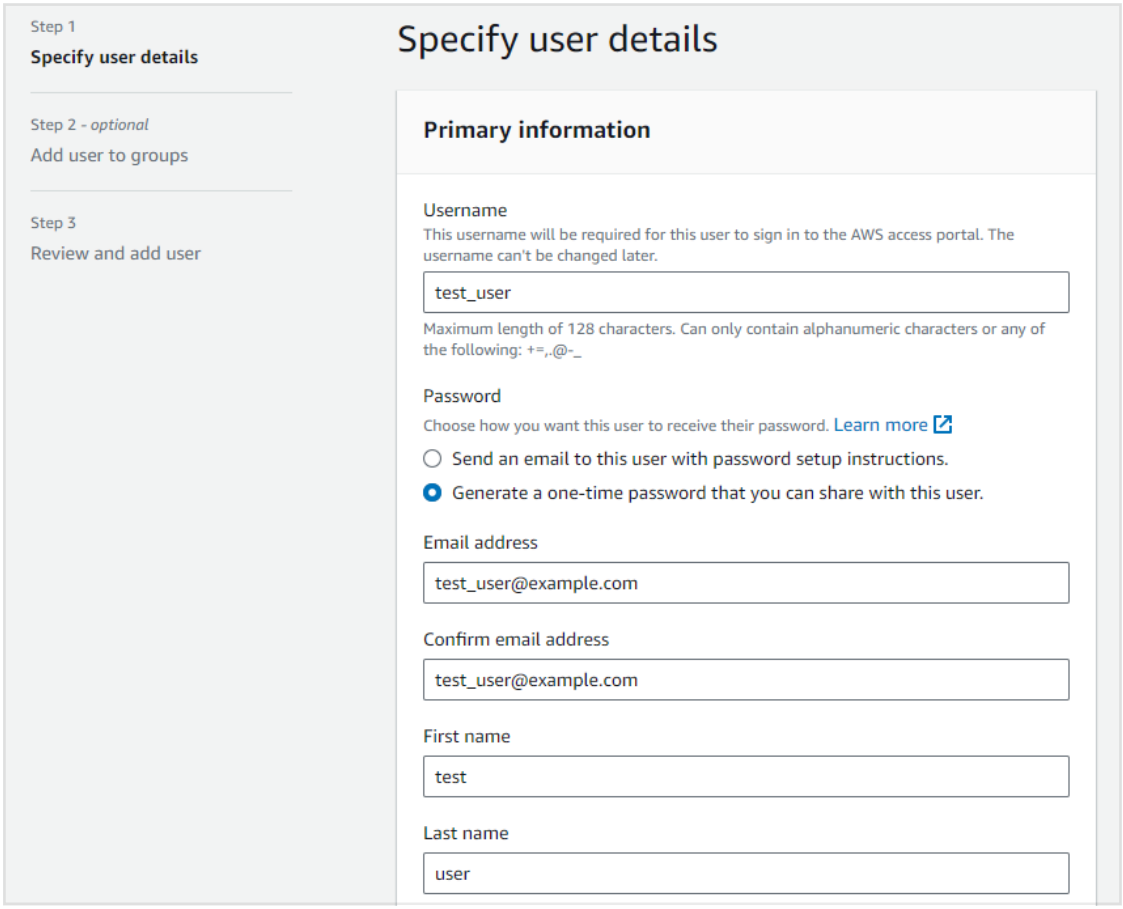
Creating Users

1. On AWS Identity Center console
   , select **Users** from left navigation pane.
2. Select **Add user**.
3. On **Specify user details** page, provide **Primary information**. In this example, we create a user called **test_user**.

- test_user
- test_user@example.com

4. Choose **Generate a one-time password that you can share with**

this user to simplify the password setup.



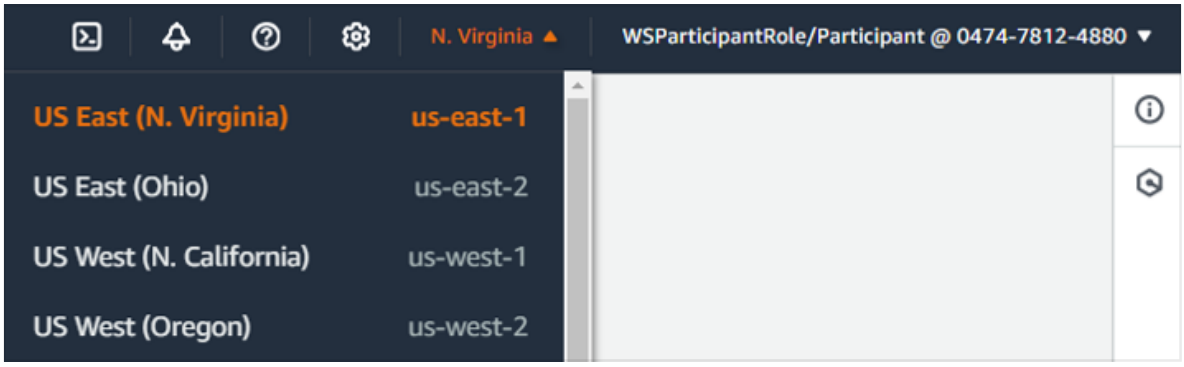
5.

6. Leave the optional fields empty, and select **Next**.
7. Select **Next** on **Add users to groups - optional** page.
8. Confirm the user details then select **Add user**.
9. Copy the one-time password, you will need this to log into the Q Business application in a later section.

Congratulations!
You have completed IAM Identity Center setup to start using Amazon Q Business Application.

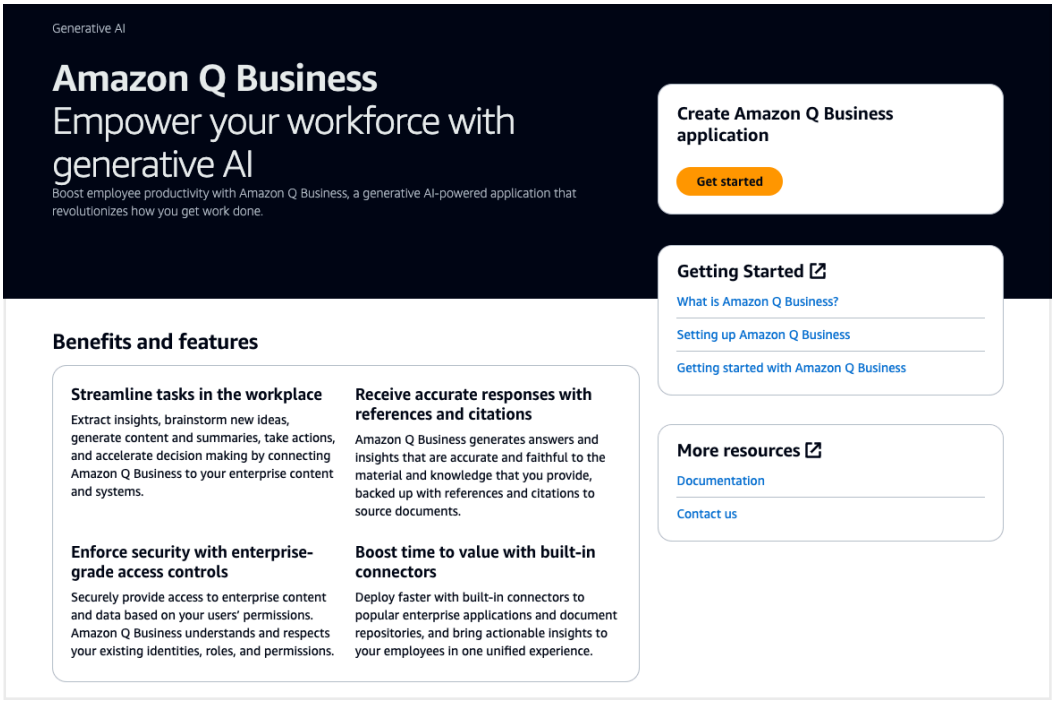
Configure Amazon Q Business application

Important - Region Selection
Ensure that you're in the us-east-1 region (N. Virginia) when setting up Amazon Q for Business.

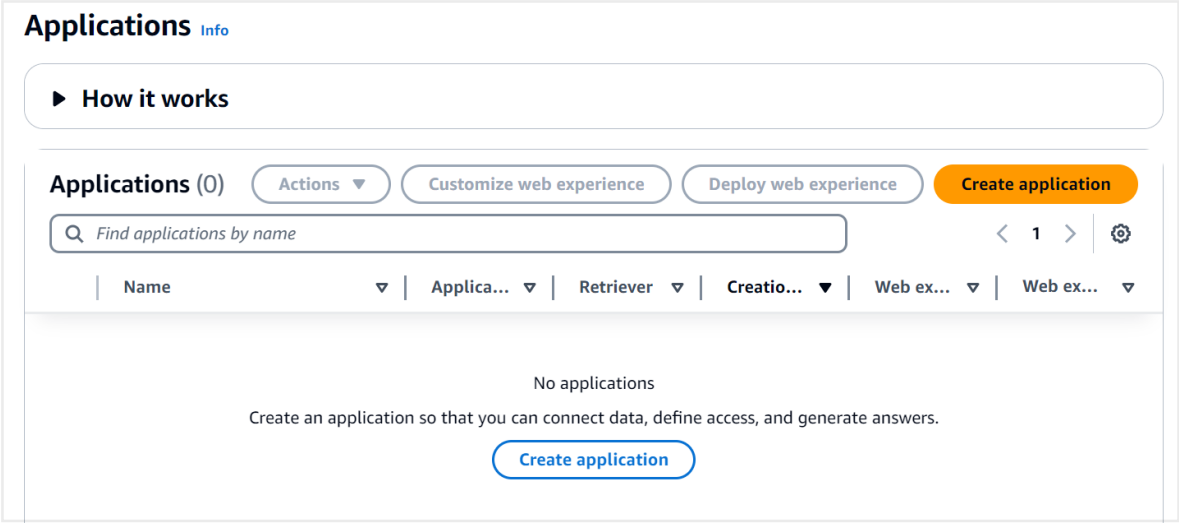


Creating an Amazon Q Business Application

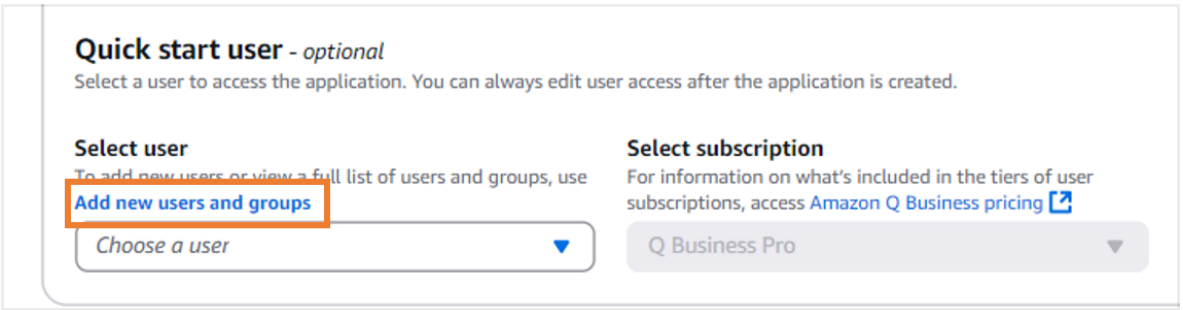
1. Open AWS Management Console for Amazon Q Business and click on **Get started**.



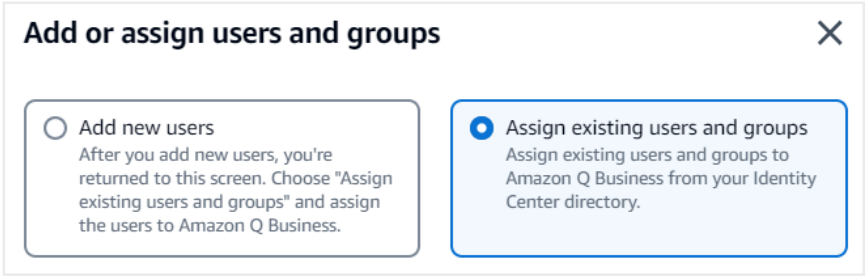
2. Select **Create application**



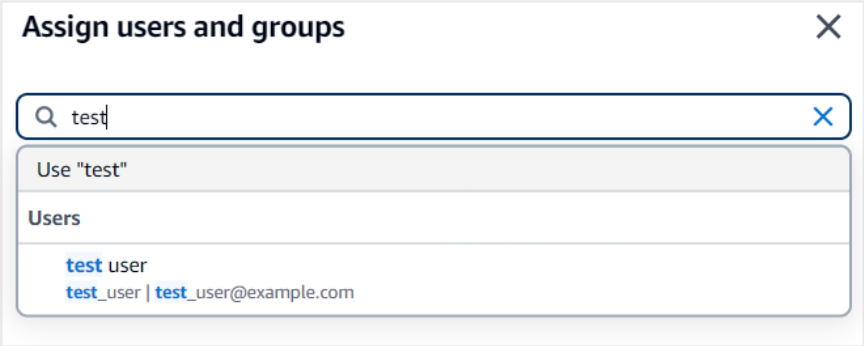
3. Choose a simple application name (exp. MyApplication) and scroll down to the **Quick start user** section. You'll notice that the IAM Identity Center instance you configured in the previous section is pre-populated.
4. Select the **Add new users and groups** link. We will choose the test user we created in our IAM Identity Center previously.



5. Choose **Existing users and groups** given that we have already created a test user in IAM Identity Center. Choose **Next**.

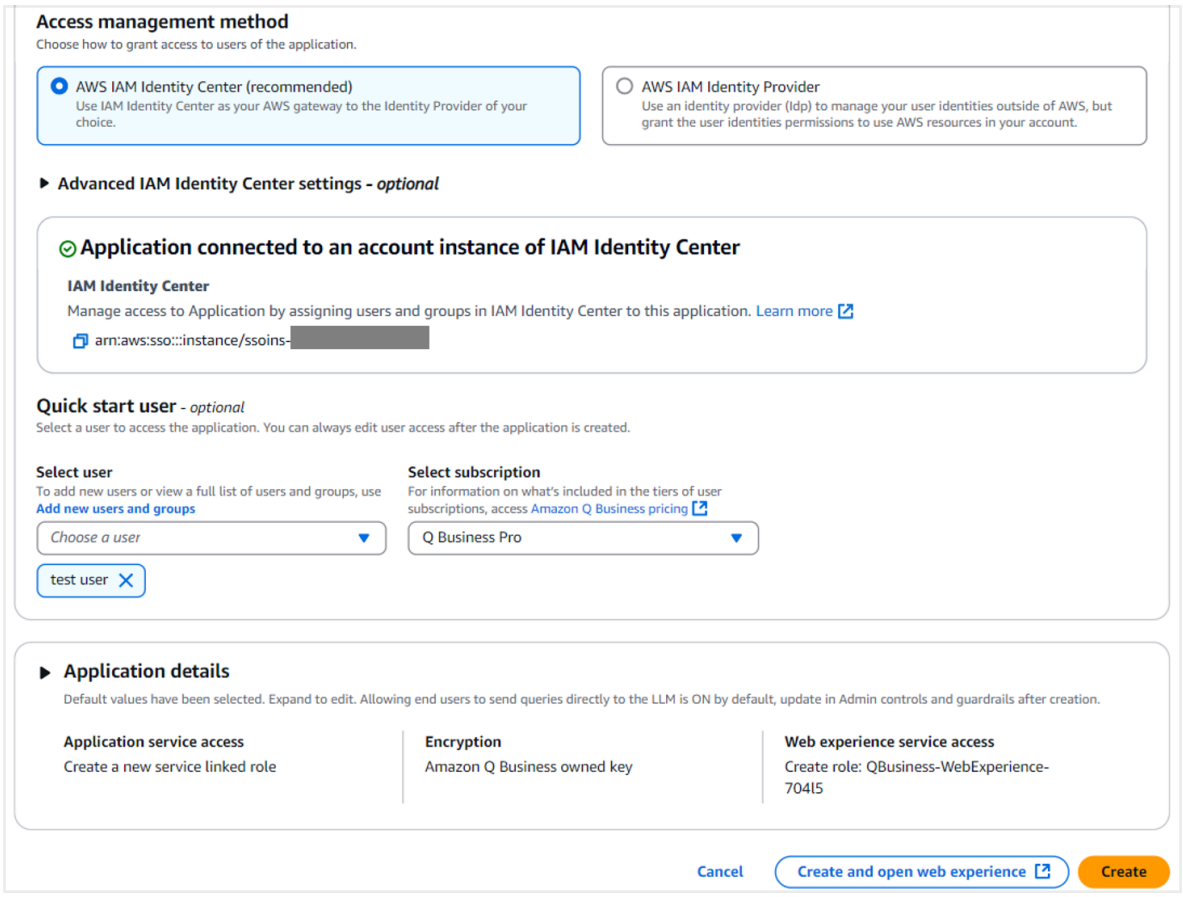


6. Choose **Get started**.
7. Select the **test_user** that was previously created in the IAM Identity Center. Choose **Assign**.



8. Confirm the user details in the **Quick start user** section. Ensure the **Select subscription** aligned to the user is **Q Business Pro**. This allows the user to use advanced features of the Amazon Q Business application.
9. Leave other settings as default, and choose **Create**. This may take 1-2 minutes to create the aligned IAM roles.

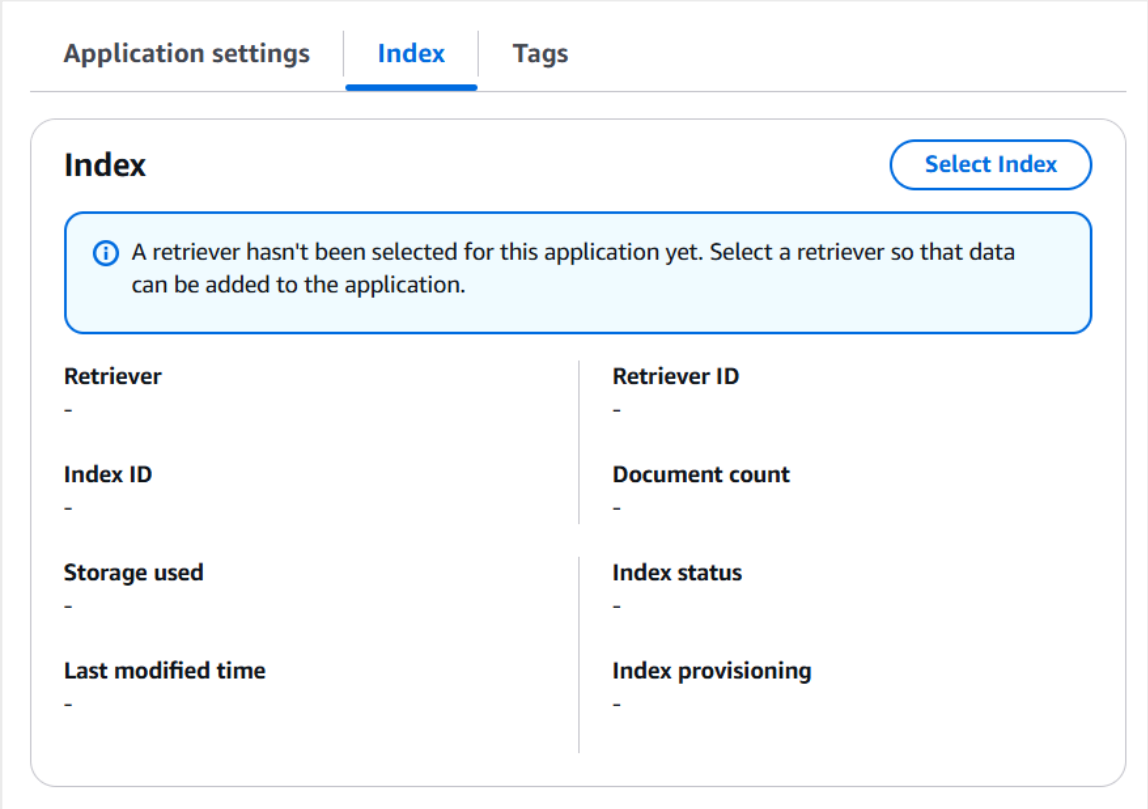
   Do not leave the page until the IAM roles have been completed or the application will not be created properly.



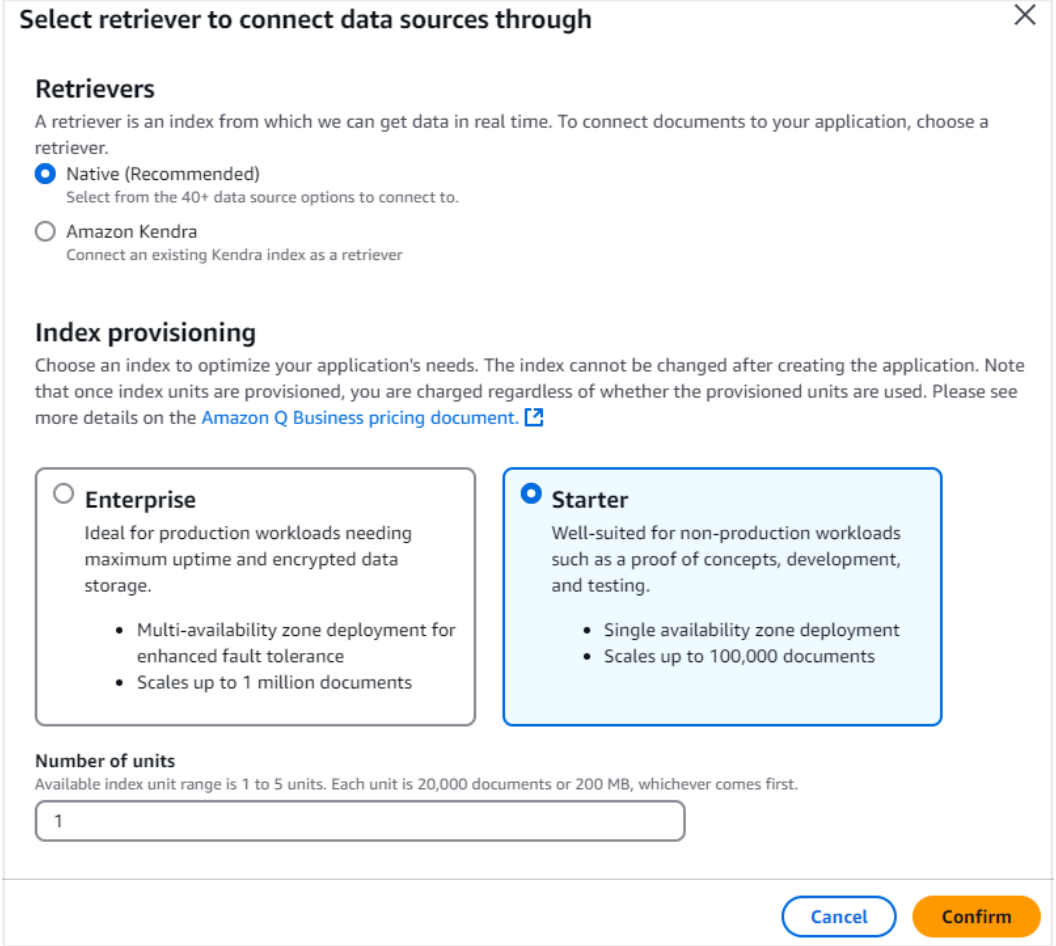
Adding a data source

Once your Q application has been completed, we need to select a retriever that can store the indexed content from our data source.

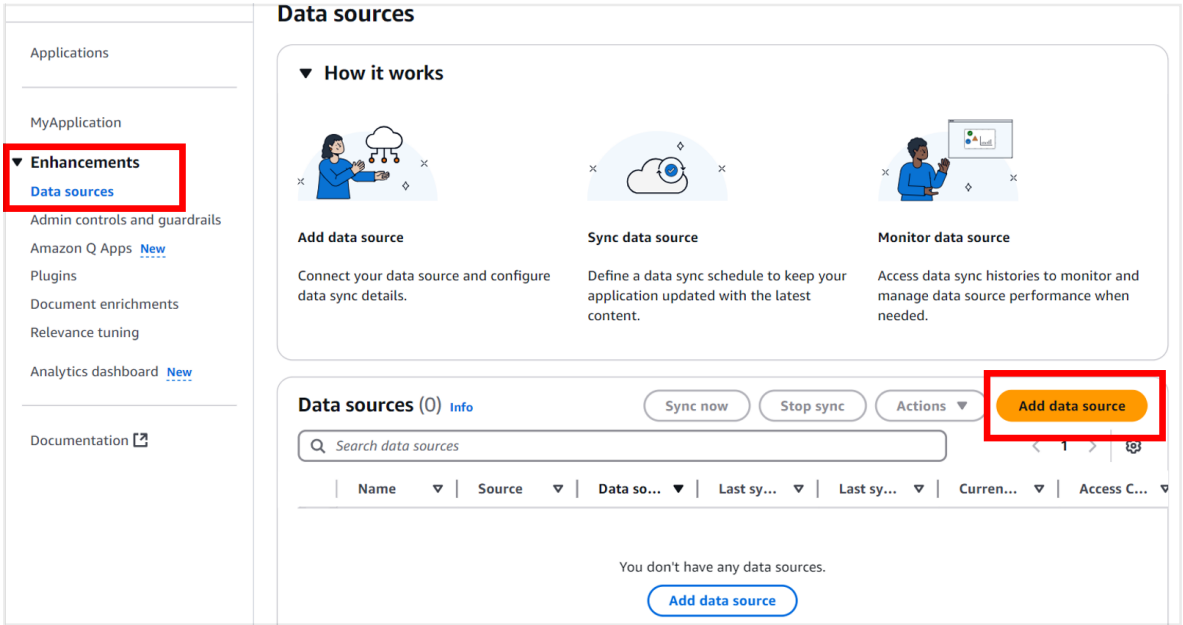
1. From your Q application home page, select the **Index** tab and choose **Select Index**.



2. Select the native retreiver. This will use Amazon Q Business's internal indexing service.
3. Select the **Starter** option for index provisioning given our small number of documents. For larger document repositories, consider increasing the number of indexing units. For this workshop, we will choose 1 index unit.
4. Choose **Confirm**

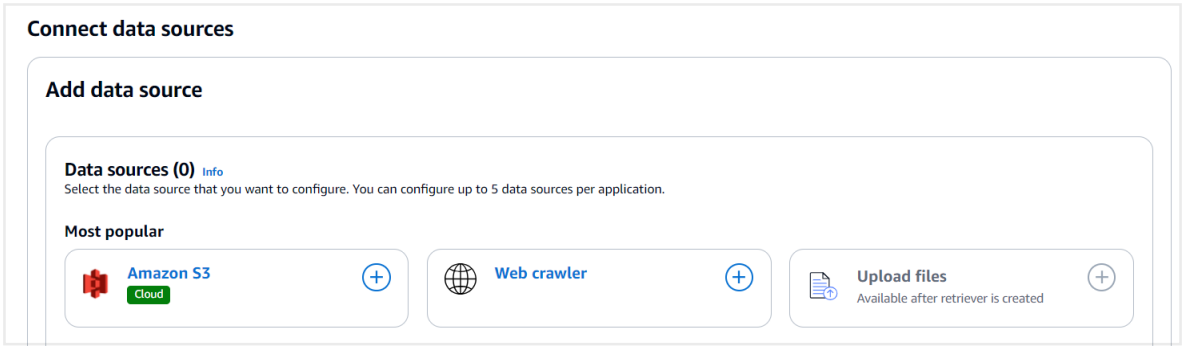


5. Next, we can add data sources to our application. Select **Data sources** from the Amazon Q Business menu under **Enhancements** then choose **Add data source**.

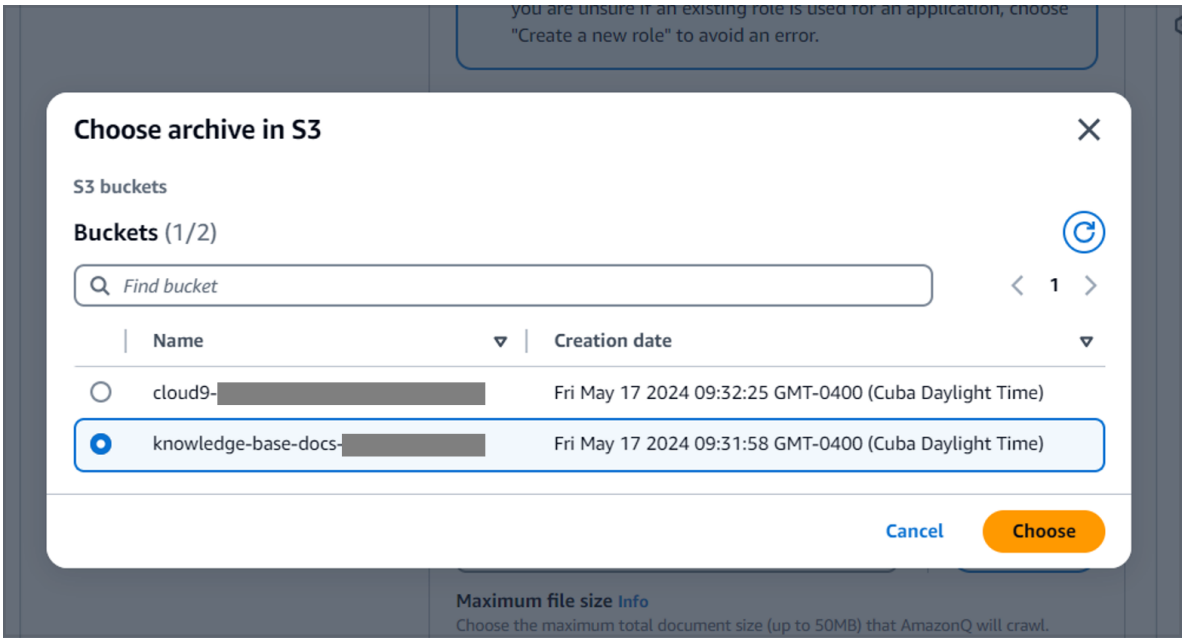


6. An Amazon S3 bucket with several PDF documents related to our food bank has been created on your behalf. Select Amazon S3 as our data

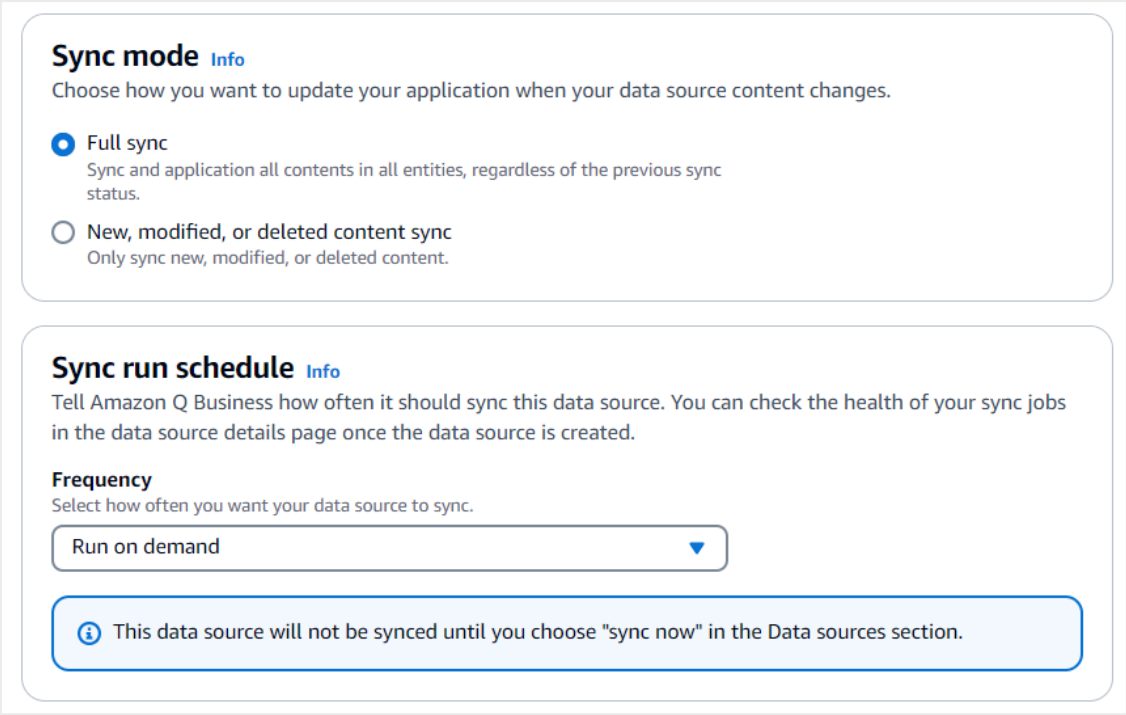
source.



7. Choose a simple name for the data source (exp. s3-docs)
8. Under **IAM role** choose **Create a new service role (recommended)**. Leave the role name as is.
9. Under **Sync scope**, browse to the Amazon S3 bucket that begins with **knowledge-base-docs** and select **Choose**. This bucket has been populated with sample documents relating to food bank impact and strategic planning.



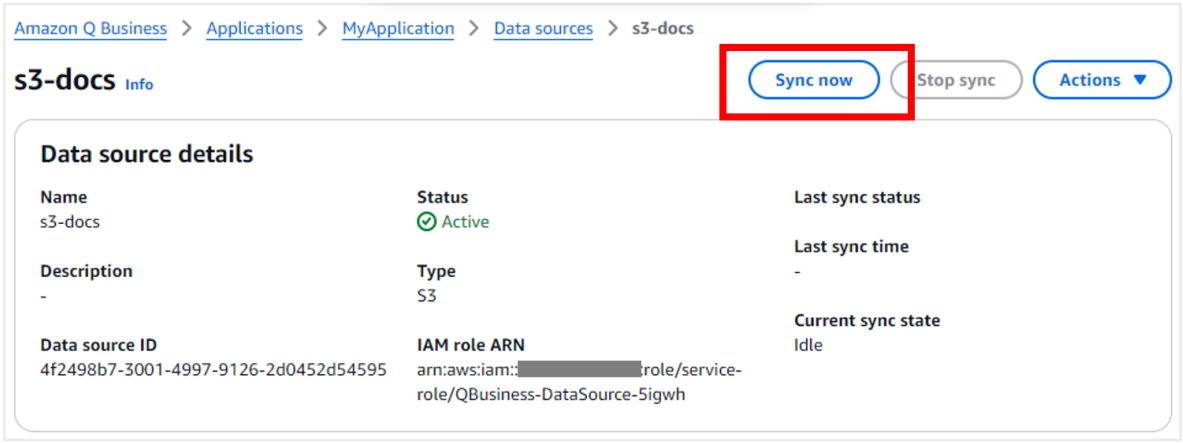
10. Maintain sync mode as **Full Sync** and choose **Run on demand** for the Frequency of the Sync run Schedule.



11. Choose **Add data source** and wait 30 seconds for the IAM role to be created.
12. After a 1-2 minutes, your data source will transiton to a status of **Active**.

Sync the data source

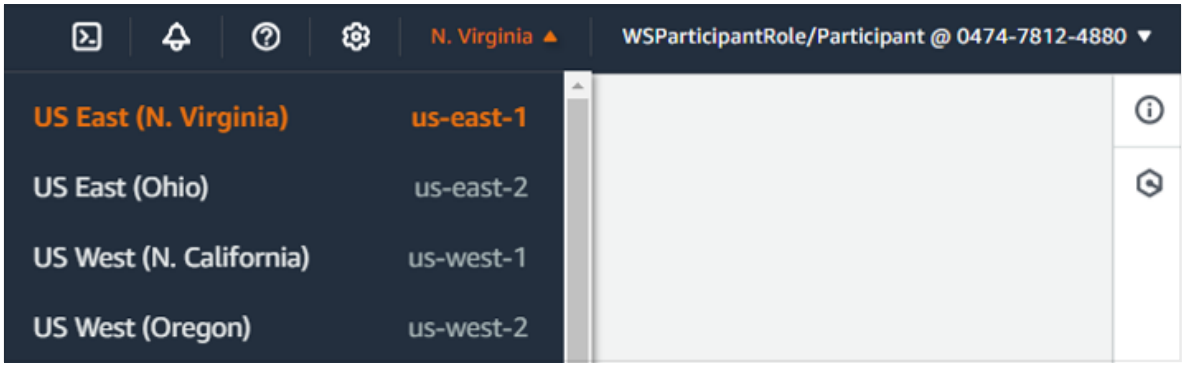
1. From the recently created data source page, select **Sync now** to begin the process of indexing your source data to the native retriever of Amazon Q Business. For our sample dataset, this will take 7-10 minutes. You are welcome to explore other labs while waiting for the indexing to complete.



Congratulations!
You have configured your Amazon Q Business Application! Next we will test the application in the web experience.

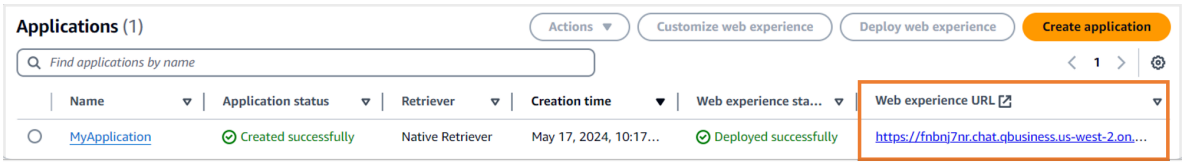
Test Amazon Q Business application

Important - Region Selection
Ensure that you're in the us-east-1 region (N. Virginia) when testing your Q for Business application.

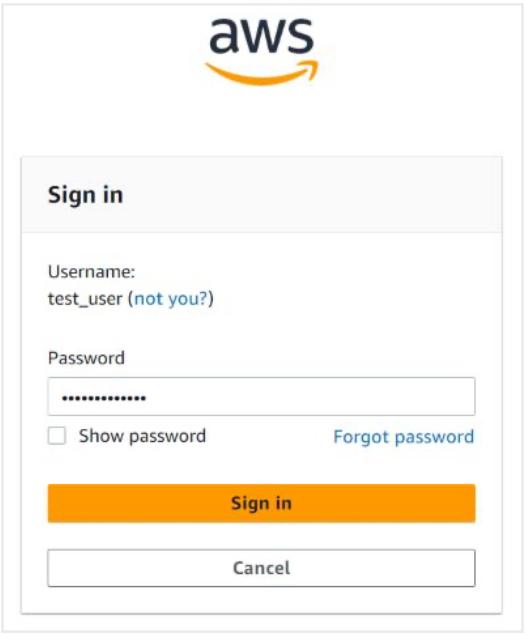


Testing an Amazon Q Business Application

1. Open AWS Management Console for Amazon Q Business and scroll to your available applications.
2. Click into the **Web Experience URL** next to your recently created application (exp. MyApplication).



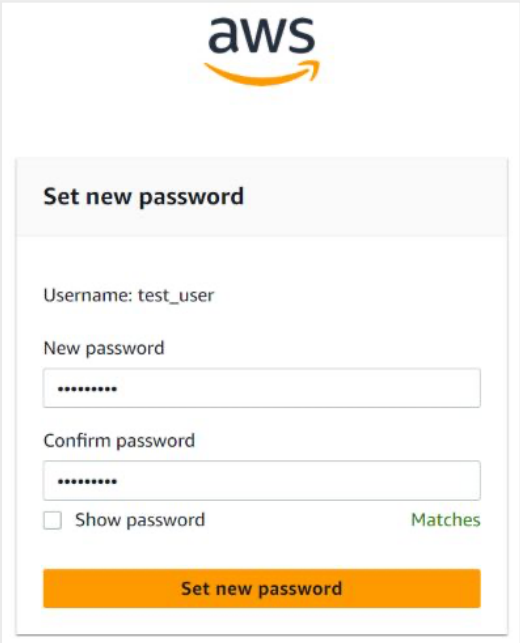
3. From the login screen, enter the name of your originally created IAM Identity Center user (exp. test_user) and the one-time password that was created for your user.



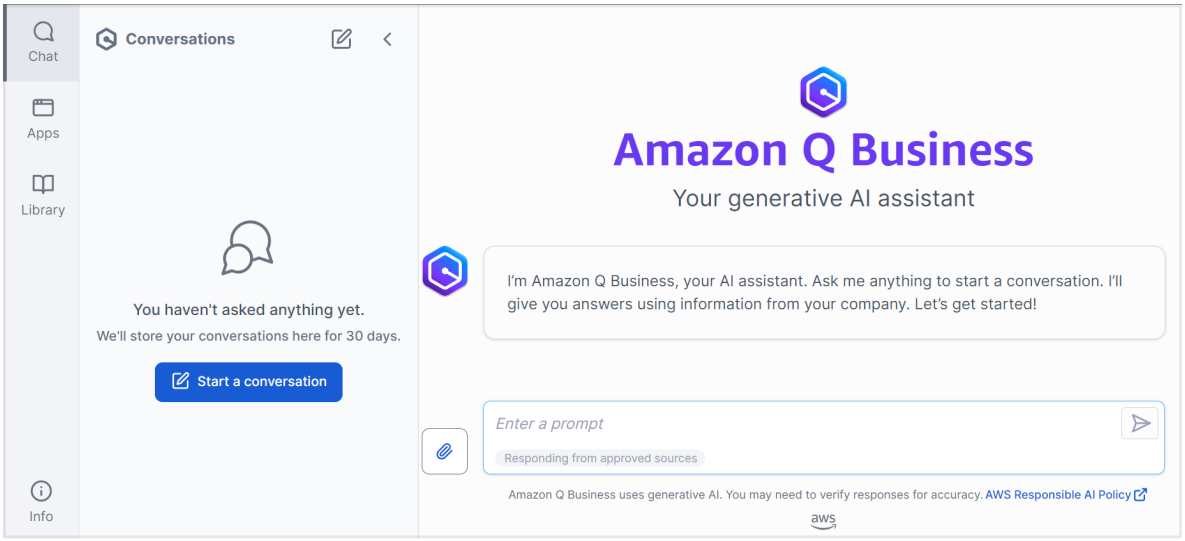
4.

Re-generate password
If you have misplaced the one-time password that was generated when you created the original user, simply navigate back to AWS Identity Center
, click into your user from the **Users** menu, and choose **Reset password**. Select **Generate a one-time password that you can share with this user** and copy the password for the login.

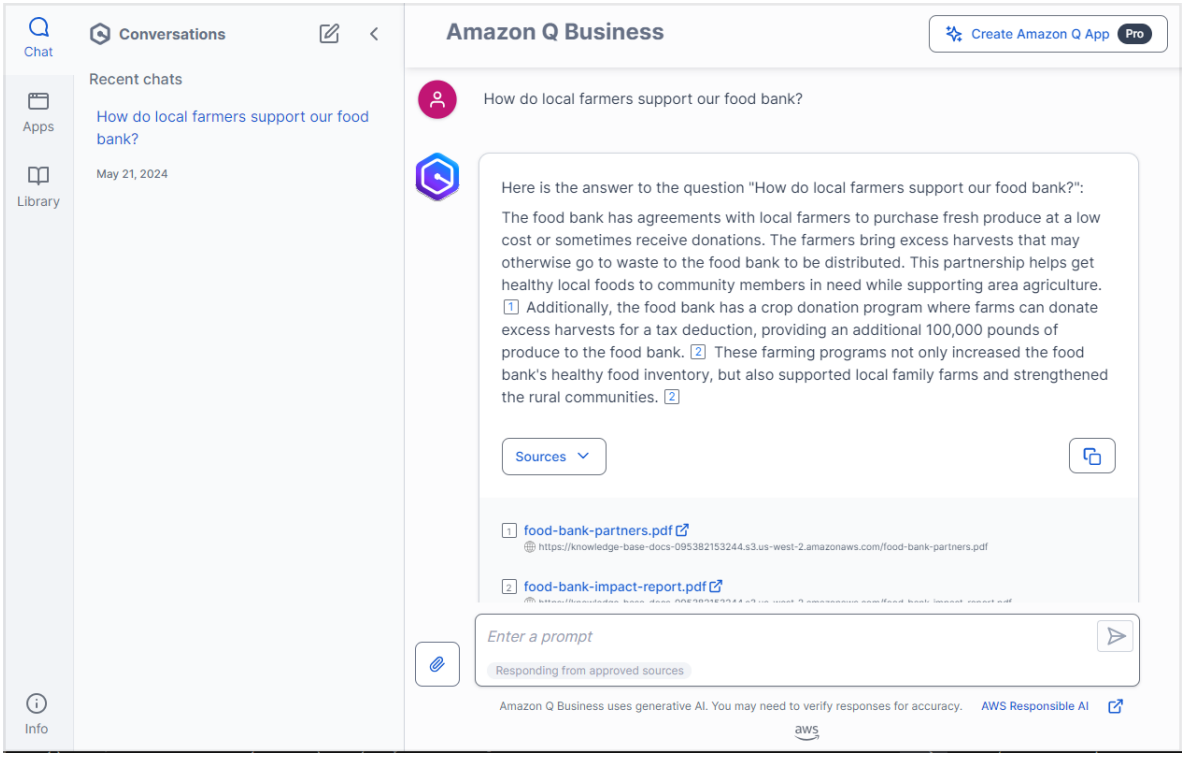
5. After logging in with your one-time password, create a new password for future logins.



5. You will be logged into the standalone Amazon Q business application.



6. From the chat window, ask questions of our indexed dataset. The chat history will be saved in the side bar and stored for future sessions.
    - How do local farmers support our food bank?
    - Summarize our food distribution statistics from 2022.
    - What is our volunteer retention strategy?



Congratulations!
You have successfully logged into and tested your Amazon Q Business Application! This concludes the Amazon Q business portion of the workshop.
Previous

AWS access portal URL: https://d-9067dc6a26.awsapps.com/start, Username: test_user, One-time password: CVb0p#?1$PNn4iaaHLrQif@@*JxL?ty #3BorgyUxq)8bx
New Password: Leaked@pass1

Full-code

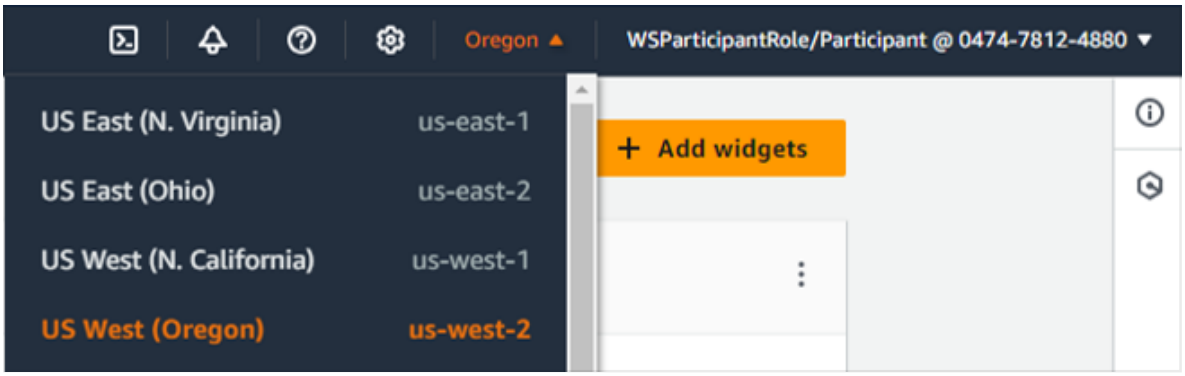
Overview

1. Getting started with GenAI
    - Full-code: Amazon Bedrock with langChain and streamlit

Launch AWS Cloud9

Important - Region Selection
Ensure that you're in the us-west-2 region (Oregon) when setting up your

Cloud9 environment.



Please complete the Running at an AWS-facilitated event steps before starting this lab.
We will be using AWS Cloud9

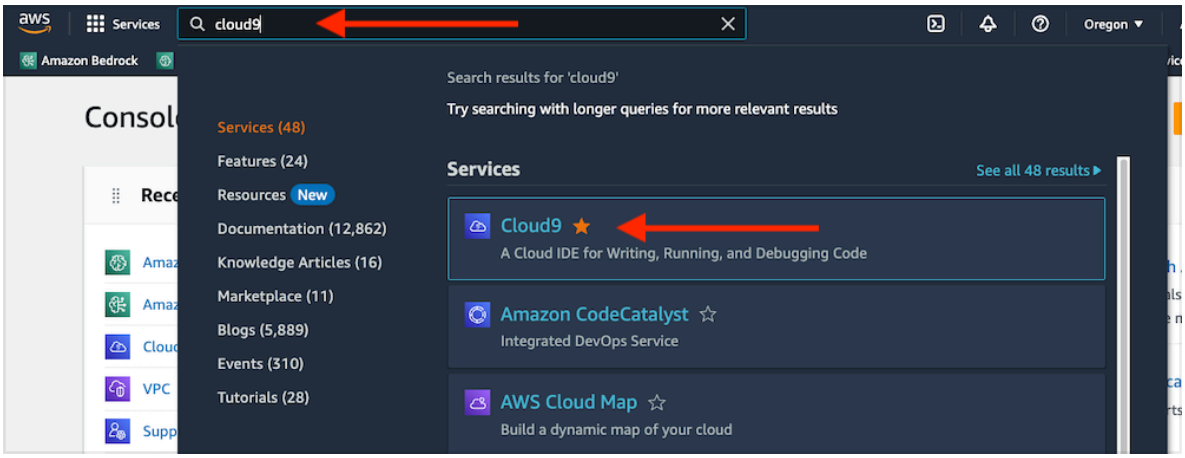
as our integrated development environment for this workshop. AWS Cloud9 is one option for building applications with Amazon Bedrock - you can also use your own development tools (VS Code, PyCharm, etc.), Amazon SageMaker Studio

, or Jupyter Notebooks.
Below we will configure an AWS Cloud9 environment

in order to build and run generative AI applications. An environment is a web-based integrated development environment for editing code and running terminal commands.

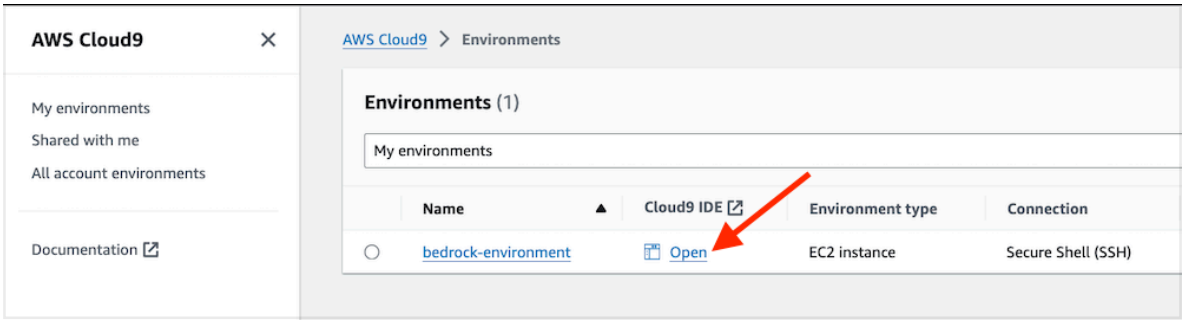
AWS Cloud9 setup instructions

1. In the AWS console, search for **Cloud9**.
    - Select **Cloud9** from the search results.

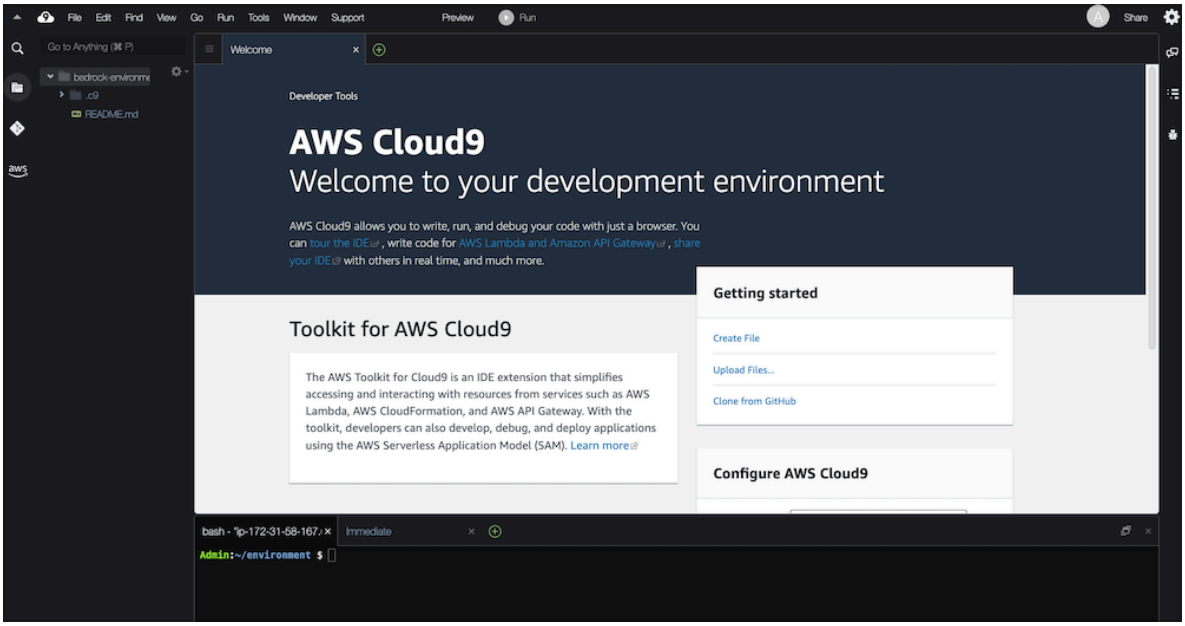


2. In the Environments list, click the **Open** link. This will launch the AWS

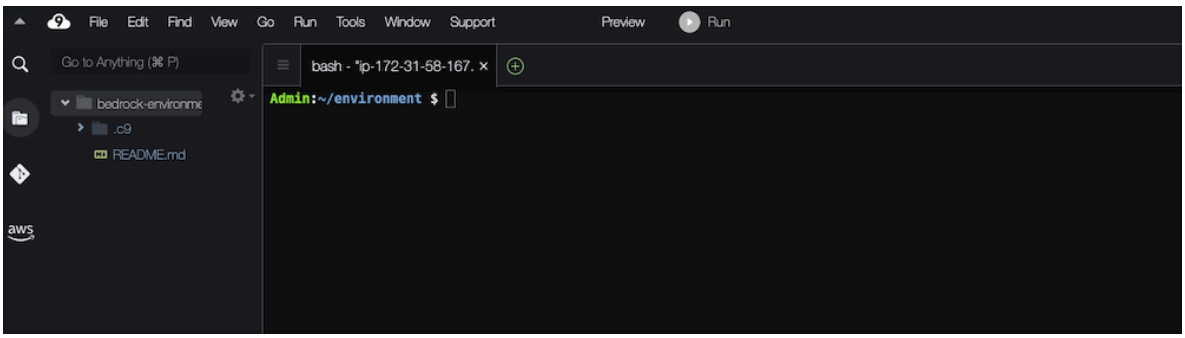
Cloud9 IDE in a new tab.



3. Confirm that the AWS Cloud9 environment loaded properly.
    - You can close the **Welcome** tab
    - You can drag tabs around to the position you want them in.



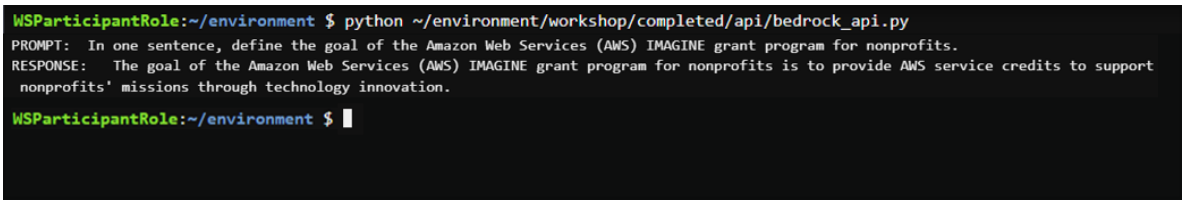
In this example, the bash terminal tab was dragged up to the top tab strip, and the bottom-aligned panel was closed:



4. Verify configuration by pasting and running the following into the AWS Cloud9 terminal:

cd ~/environment/
python ~/environment/workshop/completed/api/bedrock_api.py

If everything is working properly, you should see a response describing the AWS IMAGINE conference:



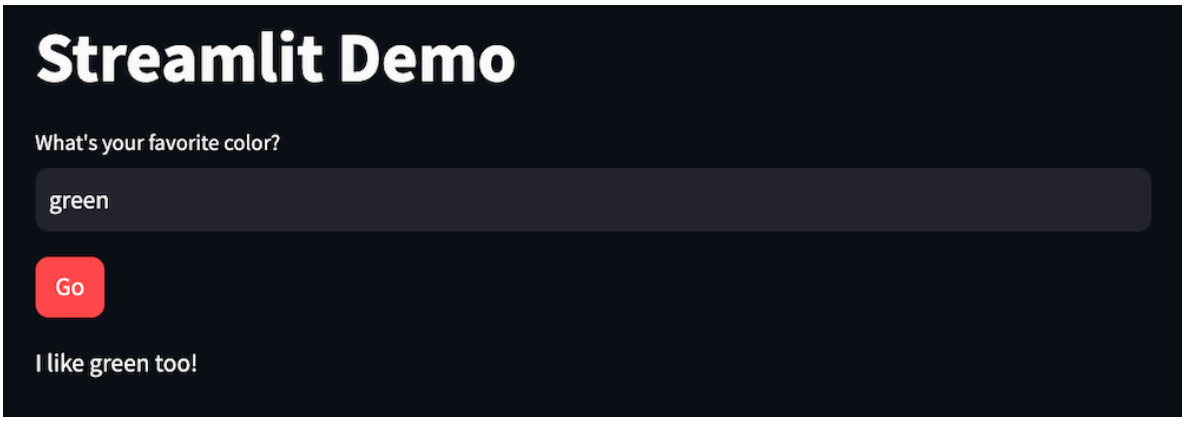
Congratulations!
You have successfully launched AWS Cloud9!

Intro to Streamlit

Please complete the Running at an AWS-facilitated event steps before starting this lab.

Lab introduction

Final product:



Streamlit is an open-source Python framework for building front-end applications to demo machine learning applications. Streamlit provides a library of commands for displaying web elements. You can see the list of controls in the Streamlit API reference

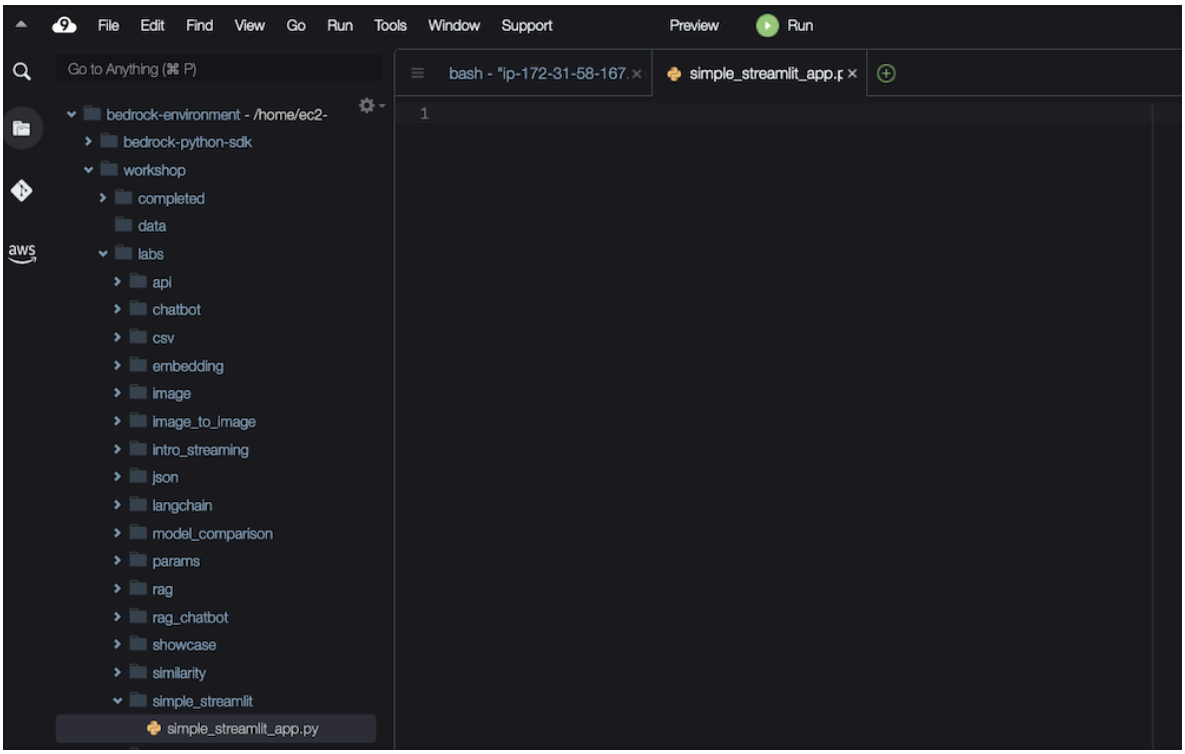
.
Streamlit allows you to build simple and attractive user interfaces with a relatively small amount of Python code. For back-end developers, this means you can create demo applications for your code, without having to learn the

various programming languages, frameworks, and hosting platforms for front-end development. Even for front-end developers, it allows you to quickly build a proof of concept to validate your approach.
Streamlit is well suited for creating generative AI prototypes. Throughout this workshop, you will have the opportunity to try a wide variety of Streamlit's capabilities that greatly simplify the demo-building process.
You run your streamlit applications from the command line using the streamlit run command. You can find a more in-depth introduction to Streamlit at its Get started page

.

Create the Streamlit app

1. Navigate to the **workshop/labs/simple_streamlit** folder, and open the file **simple_streamlit_app.py**



2. This script was created for you to inspect and test a sample streamlit application. In this section, you will familiarize yourself with the code and prepare to create your own streamlit app in later sections.
- This statement allows us to use Streamlit elements and functions. import streamlit as st #all streamlit commands will be available through the "st" alias

-
-

- Here we are setting the page title on the actual page and the title shown in the browser tab.
  st.set_page_config(page_title="Streamlit Demo") #HTML title
- st.title("Streamlit Demo") #page title
-
-

- We are creating an input text box and button to get a color from the user.
  color_text = st.text_input("What's your favorite color?") #display a text box
- go_button = st.button("Go", type="primary") #display a primary button
-
-

- We use the if block below to handle the button click. We then format the submitted color and display it using Streamlit's write function.
  if go_button: #code in this if block will be run when the button is clicked
-
- st.write(f"I like {color_text} too!") #display the response content
-
-

3. Save the file

Great! Now you are ready to run the streamlit app!

Run the Streamlit app

1. Select the **bash terminal** in AWS Cloud9 and change directory. If you don't see a bash terminal open, select **Window > New Terminal** from the Cloud9 toolbar.

cd ~/environment/workshop/labs/simple_streamlit

Just want to run the app?

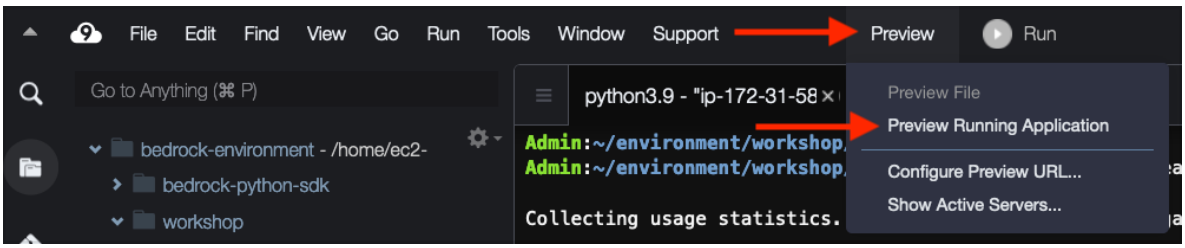
Expand here & run this command instead

2. Run the streamlit command from the terminal.

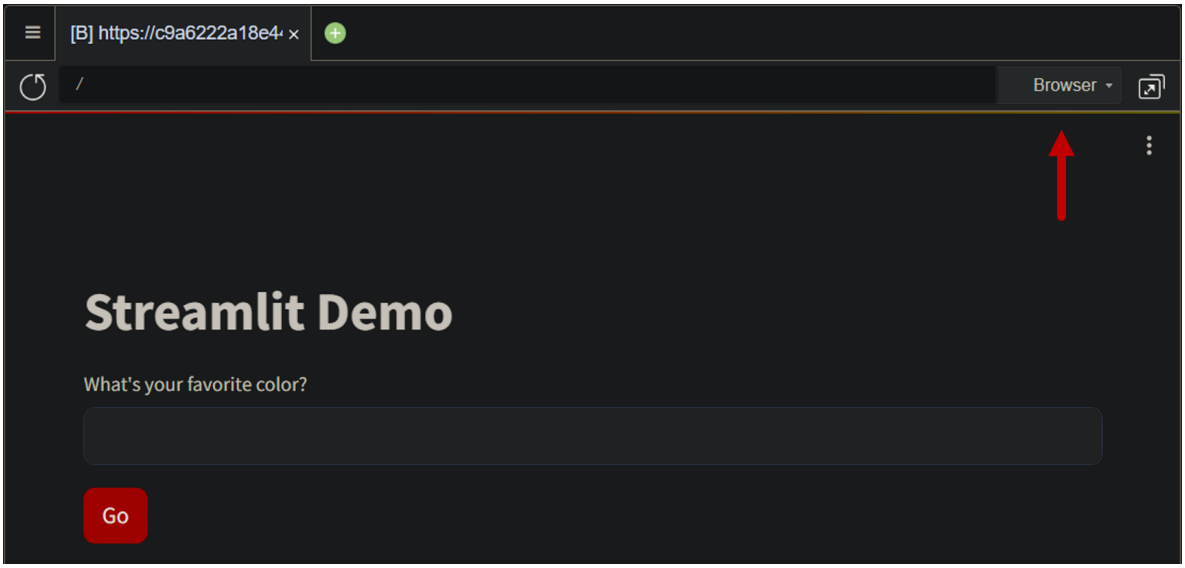
```
streamlit run simple_streamlit_app.py --server.port 8080
```

Ignore the Network URL and External URL links displayed by the Streamlit command. Instead, we will use AWS Cloud9's preview feature.

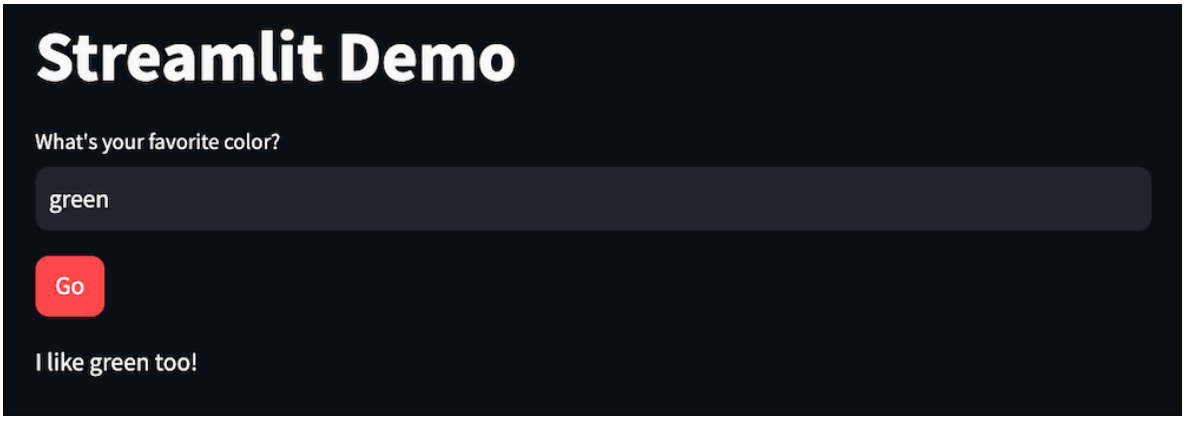
3. In AWS Cloud9, select **Preview** -> **Preview Running Application**.



You should see a web page like below. Ensure the **Browser** option is selected at the top right.



4. Enter a color and see the results.



5. Close the preview tab in AWS Cloud9. Return to the terminal and press Control-C to exit the application.

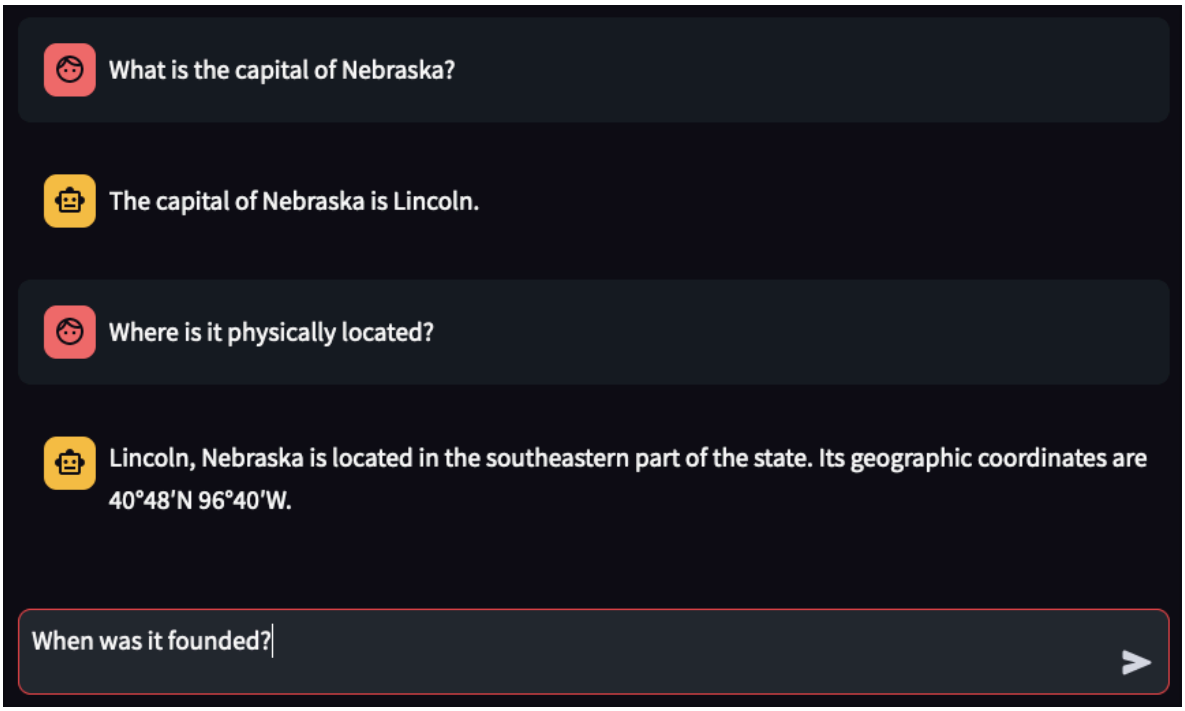
Congratulations!
You have successfully built a simple Streamlit app!

Chatbot

Please complete the Running at an AWS-facilitated event steps before starting this lab.

Lab introduction

Final product:



In this lab, we will build a simple chatbot with Amazon Bedrock and Streamlit. We will use the Amazon Bedrock Converse API

which simplifies development of chat experiences using models invoked through Bedrock.

Amazon Bedrock (and LLMs in general) don't have any concept of state or memory. Any chat history has to be tracked externally and then passed into the model with each new message. The Converse API supports conversational history in a structured way as part of the API request, reducing the complexity for multi-turn conversations.

An important difference between this lab and the retrieval-augmented generation chatbot lab: the responses from this chatbot are based purely on the underlying foundation model, without any supporting data source. So the chatbot's messages can potentially include made-up responses (hallucination). In a later lab, we will create a more powerful chatbot that incorporates the retrieval-augmented generation pattern to return more accurate responses.

You can build the application code by copying the code snippets below and pasting into the indicated Python file.

Just want to run the app?

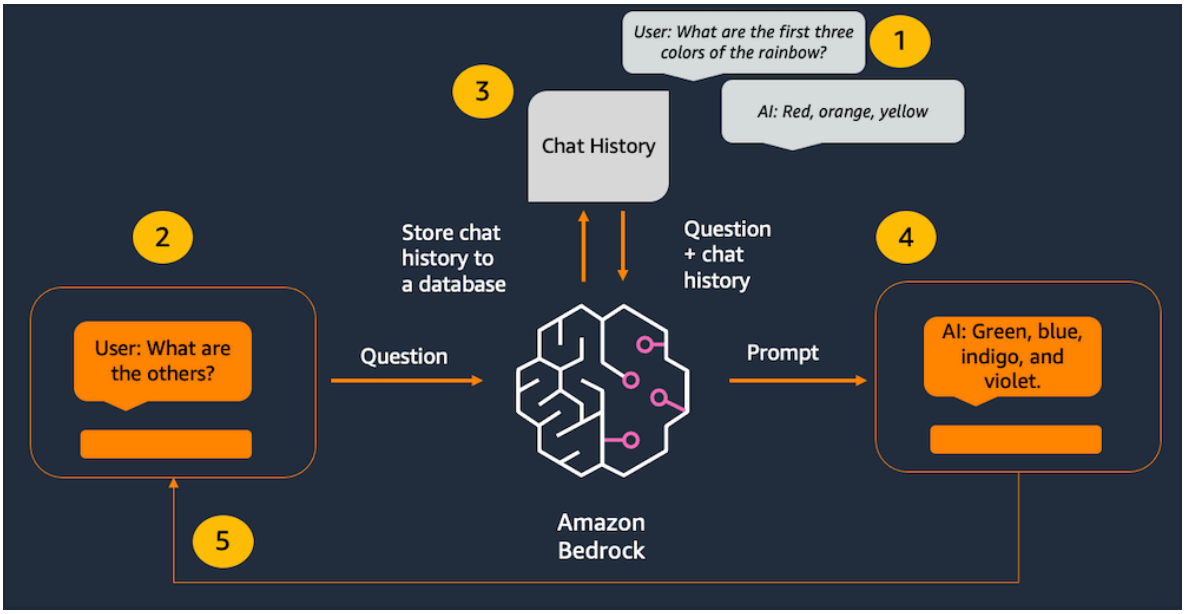
You can jump ahead to run a pre-made application.

Use cases

The chatbot pattern is good for the following use cases:

- Simple interactive user conversation, without the use of any specialized knowledge or data.

Architecture



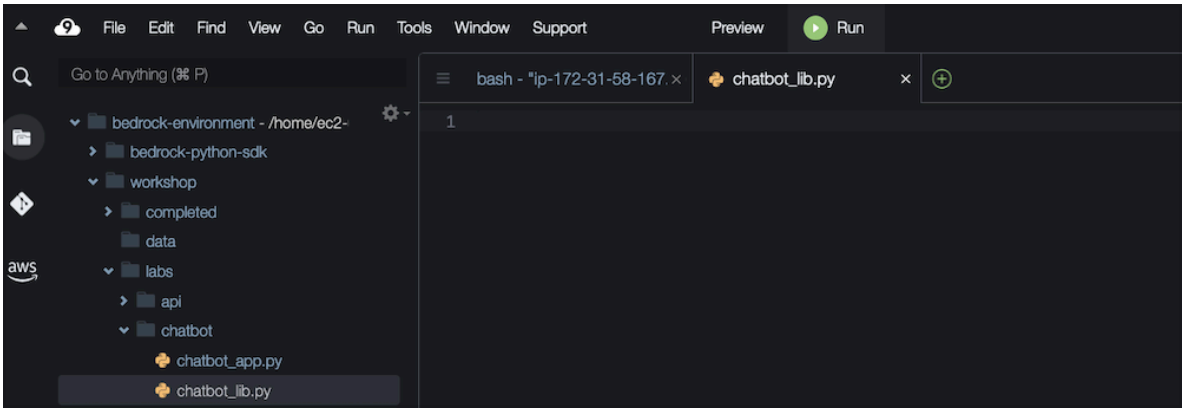
1. Past interactions are tracked in the chat memory object.
2. The user enters a new message.
3. The chat history is retrieved from the memory object and added before the new message.
4. The combined history & new message are sent to the model.
5. The model's response is displayed to the user.

This application consists of two files: one for the Streamlit front end, and one for the supporting library to make calls to Bedrock.

Create the library script

First we will create the supporting library to connect the Streamlit front end to the Bedrock back end.

1. Navigate to the **workshop/labs/chatbot** folder, and open the file **chatbot_lib.py**



2. Add the import statements and create a Bedrock client.

```
import boto3
import os

bedrock = boto3.client('bedrock-runtime') # Creates a Bedrock client
```

3. Create a memory list to store conversation history. We will append messages into this list and submit through the Converse API request later on.

```
# Instantiate a messages list to store the conversation history. This will preserve context along with the latest message.

messages = []
```

4. Add this function to call bedrock.
    - We're creating a function we can call from the Streamlit front end application. This function defines model parameters like temperature and top_k, then submits the user's query to the Claude 3 Sonnet model using the Converse API. The response is returned to the front end.

```
def invoke_model(message):

    # Define the system prompts to guide the model's behavior and role.
    system_prompts = [{"text": "You are a helpful assistant. Keep your answers short and succinct."}]

    # Format the user's message as a dictionary with role and content
    message = {"role": "user", "content": [{"text": message}]}

    # Append the formatted user message to the list of messages.
```

```
    messages.append(message)

    # Set the temperature for the model inference, controlling the randomness of
the responses.
    temperature = 0.5

    # Set the top_k parameter for the model inference, determining how many of
the top predictions to consider.
    top_k = 200

    # Call the converse method of the Bedrock client object to get a response
from the model.
    response = bedrock.converse(
        modelId="anthropic.claude-3-sonnet-20240229-v1:0",
        messages=messages,
        system=system_prompts,
        inferenceConfig={"temperature": temperature},
        additionalModelRequestFields= {"top_k": top_k}
    )

    # Extract the output message from the response.
    output_message = response['output']['message']

    # Append the output message to the list of messages.
    messages.append(output_message)

    # Print the message output to review how the conversation history builds
after each turn
    print("Message History: ", messages)
    print("##############################")

    # Return the text of the model's response to the frontend
    return output_message['content'][0]['text']
```

5. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Nice! You are done with the backing library. Now we will create the front-end application.

Create the Streamlit front-end app

1. In the same folder as your lib file, open the file **chatbot_app.py**

2. Import the streamlit library and import the invoke_model function from our library script.

    - These statements allow us to use Streamlit elements and call functions in the backing library script.

```
import streamlit as st
from chatbot_lib import invoke_model
```

3. Add the page title and configuration.
    - Here we are setting the page title on the actual page and the title shown in the browser tab.

```
st.set_page_config(page_title="Chatbot") #HTML title
st.title("Amazon Bedrock Chatbot with the Converse API") #page title
```

4. Add the message history to the session cache.
    - This allows us to maintain a unique chat memory per user session. Otherwise, the chatbot won't be able to remember past messages to display on the front end UI.
    - In Streamlit, session state is tracked server-side. If the browser tab is closed, or the application is stopped, the session and its chat history will be lost. In a real-world application, you would want to track the chat history in a database like Amazon DynamoDB

```
# configuring values for session state

if "messages" not in st.session_state:

    st.session_state.messages = []
```

5. Add the for loop to render previous chat messages in the front end UI.
    - Re-render previous messages based on the "messages" session state object.

```
#Re-render the chat history (Streamlit re-runs this script, so need this to preserve previous chat messages)

for message in st.session_state.messages:

    with st.chat_message(message["role"]):
        st.markdown(message["content"])
```

6. Add the input elements.

- We use the if block below to handle the user input. See the in-line comments below for more details.

```
# evaluating st.chat_input and determining if a question has been input.

if question := st.chat_input("Ask me about anything..."):

    # with the user icon, write the question to the front end
    with st.chat_message("user"):
        st.markdown(question)

    # append the question and the role (user) as a message to the session state
    st.session_state.messages.append({"role": "user", "content": question})

    # respond as the assistant with the answer
    with st.chat_message("assistant"):

        # making sure there are no messages present when generating the answer
        message_placeholder = st.empty()

        # passing the user question into the function from our library script and
returning our response
        answer = invoke_model(question)

        # writing the answer to the front end
        message_placeholder.markdown(f"{answer}")

    st.session_state.messages.append({"role": "assistant","content": answer})
```

7. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Magnificent! Now you are ready to run the application!

Run the Streamlit app

1. Select the **bash terminal** in AWS Cloud9 and change directory. If you don't see a bash terminal open, select **Window > New Terminal** from the Cloud9 toolbar.

```
cd ~/environment/workshop/labs/chatbot
```

Just want to run the app?

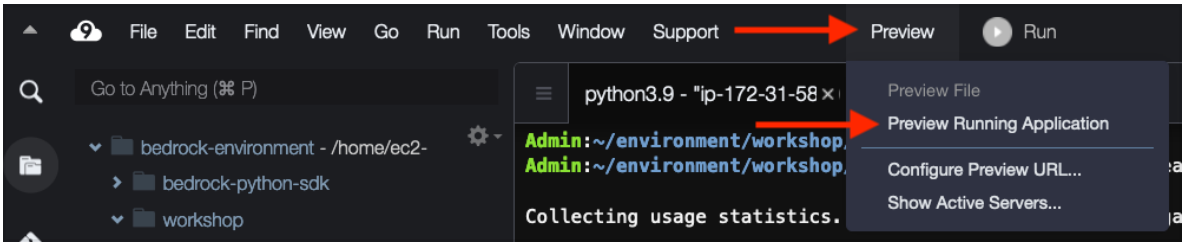
Expand here & run this command instead

2. Run the streamlit command from the terminal.

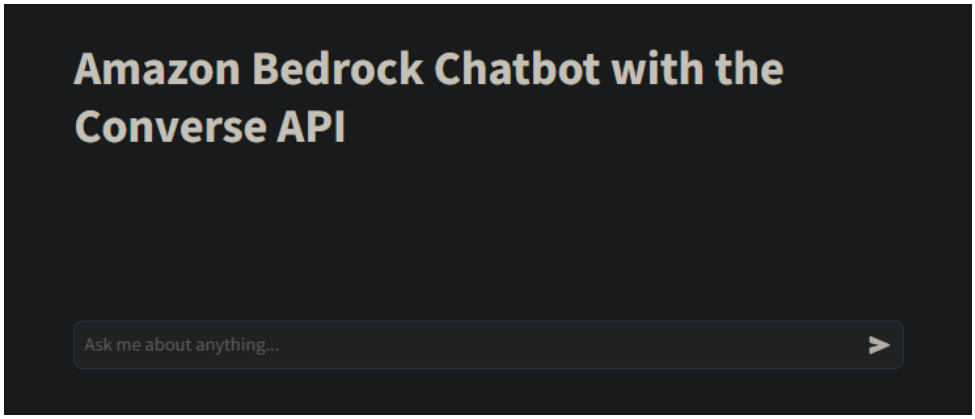
```
streamlit run chatbot_app.py --server.port 8080
```

Ignore the Network URL and External URL links displayed by the Streamlit command. Instead, we will use AWS Cloud9's preview feature.

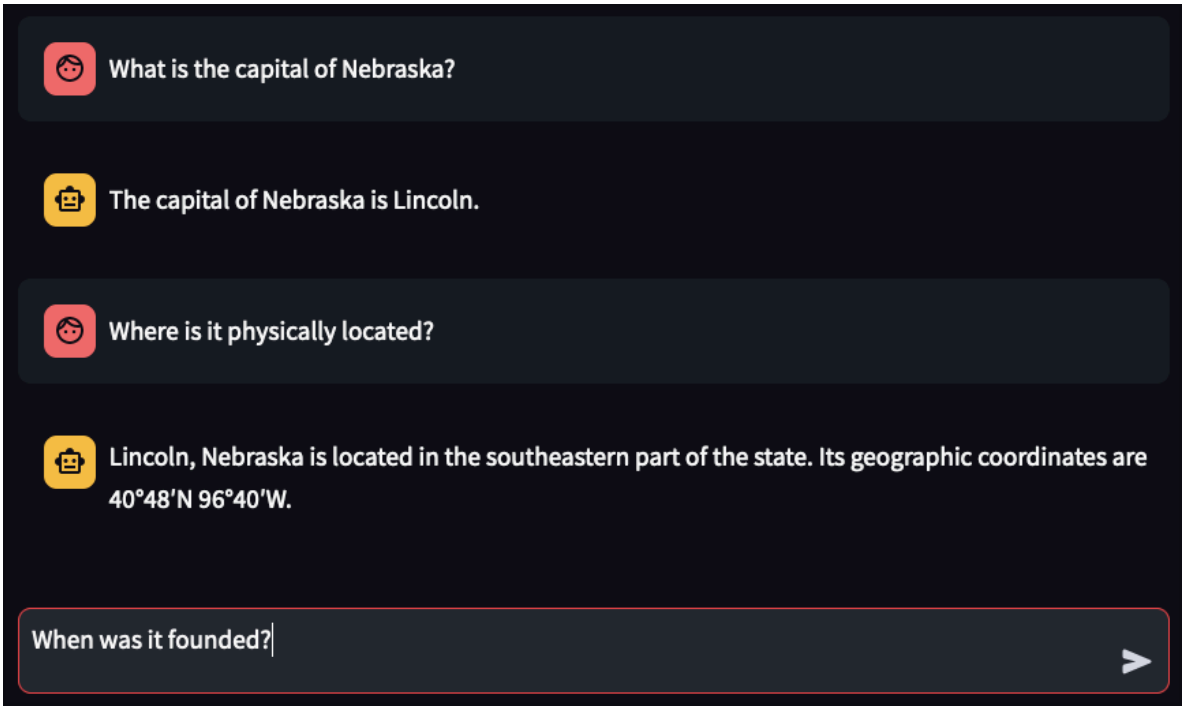
3. In AWS Cloud9, select **Preview** -> **Preview Running Application**.



You should see a web page like below:



4. Try out some prompts and see the results. You will see the chat history build incrementally from the bash terminal output.
    - What is the capital of Nebraska?
    - When was it founded?
    - what is the population?



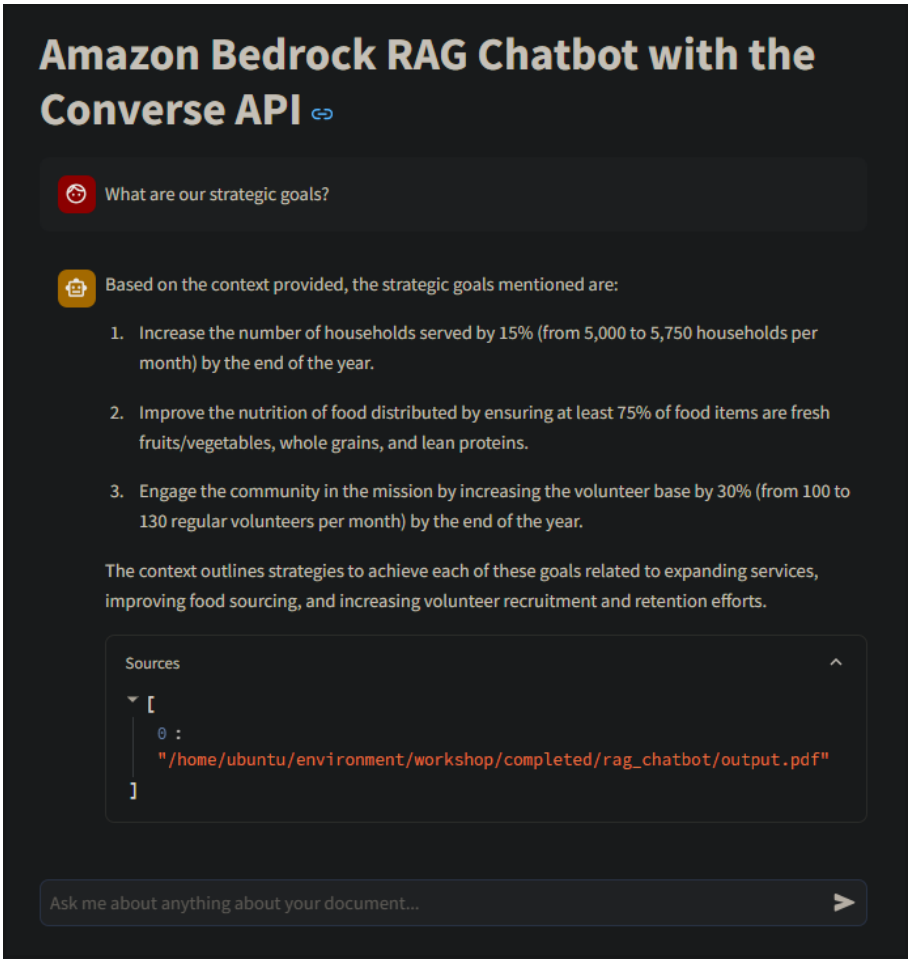
5. Close the preview tab in AWS Cloud9. Return to the terminal and press Control-C to exit the application.

Congratulations!
You have successfully built a chatbot with Bedrock and Streamlit!

Chatbot with RAG

Please complete the Running at an AWS-facilitated event steps before starting this lab.
Lab introduction

Final product:



In this lab, we will build a simple chatbot with Amazon Bedrock,Langchain and Streamlit. We will use the Amazon Bedrock Converse API

which simplifies development of chat experiences using models invoked through Bedrock.

Amazon Bedrock (and LLMs in general) don't have any concept of state or memory. Any chat history has to be tracked externally and then passed into the model with each new message. The Converse API supports conversational history in a structured way as part of the API request, reducing the complexity for multi-turn conversations.

We also want to supplement the model's underlying data with external knowledge through Retrieval-Augmented Generation (RAG). We will use a strategic plan PDF for a sample food bank to represent our corpus of data. We'll use the LangChain

python library, which provides convenient functions for chunking

data and interacting with services like vector databases.
In this lab, we will build a chatbot supported by Retrieval-Augmented Generation (RAG). We'll use Claude Sonnet, Amazon Titan Embeddings, LangChain, and Streamlit. We will use an in-memory FAISS

database to demonstrate the RAG pattern. In a real-world scenario, you will most likely want to use a persistent data store like Amazon Kendra or the vector engine for Amazon OpenSearch Serverless

.
You can build the application code by copying the code snippets below and pasting into the indicated Python file.

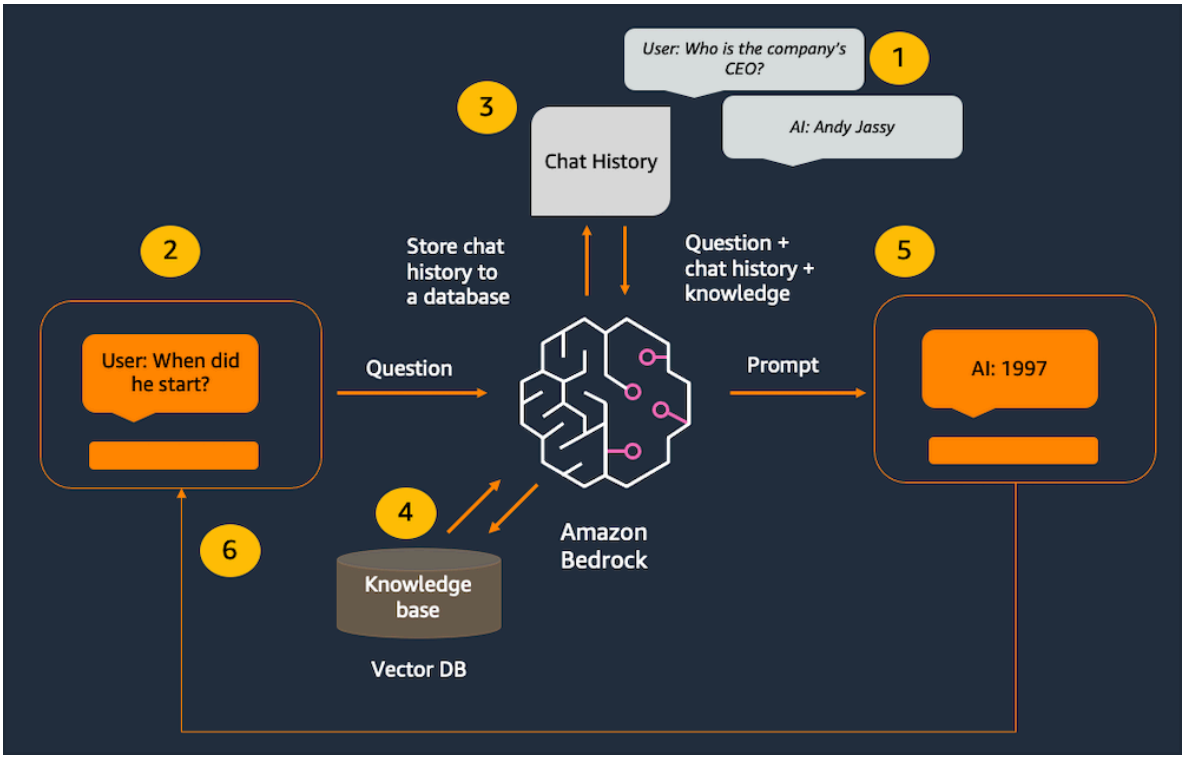
Just want to run the app?
You can jump ahead to run a pre-made application.

Use cases

The chatbot with RAG pattern is good for the following use cases:

- Simple interactive user conversation, supported by specialized knowledge or data

Architecture



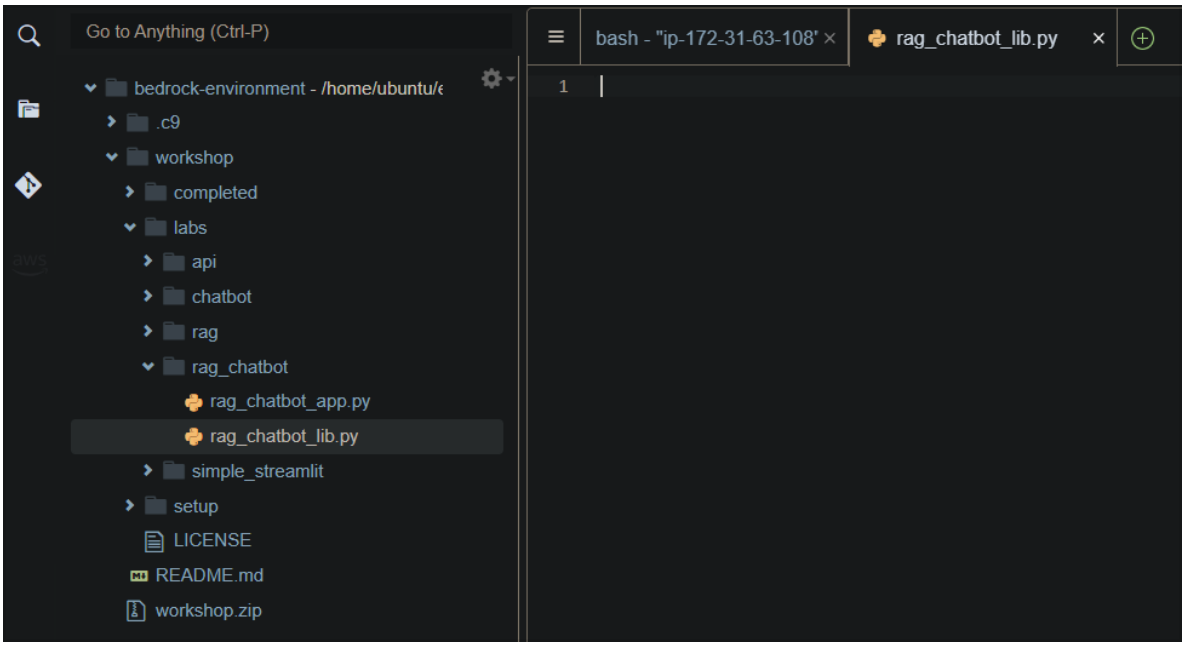
1. Past interactions are tracked in the chat memory object.
2. The user enters a new message.
3. The chat history is retrieved from the memory object and added before the new message.
4. The question is converted to a vector using Amazon Titan Embeddings, then matched to the closest vectors in the vector database.
5. The combined history, knowledge, and new message are sent to the model.
6. The model's response is displayed to the user.

This application consists of two files: one for the Streamlit front end, and one for the supporting library to make calls to Bedrock.

Create the library script

First we will create the supporting library to connect the Streamlit front end to the Bedrock back end.

1. Navigate to the **workshop/labs/rag_chatbot** folder, and open the file **rag_chatbot_lib.py**



2. Add the import statements.
    - Import LangChain libraries to help split our document into chunks of text. We will embed these chunks into an in-memory vector store to support retrieval later on.

```
import boto3
import os
import json

# use langchain to help with chunking text and storing embeddings in FAISS vector store

from langchain_aws.embeddings import BedrockEmbeddings
from langchain_community.document_loaders import PyPDFLoader
from langchain_community.vectorstores import FAISS
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

3. Create the Bedrock and OpenSearch clients for the AWS SDK.

```
bedrock = boto3.client('bedrock-runtime') # Creates a Bedrock client
```

4. Add a function that chunks our document into pieces and adds to an in-memory vector store. Use the Langchain helper libraries to support the text splitting.

```
def create_vector_db():
```

```
    # creates and returns an in-memory vector database to be used in the
application

    # create the client for Amazon Bedrock embeddings
    embeddings = BedrockEmbeddings(
        credentials_profile_name='default', #sets the profile name to use for AWS
credentials (default)
        model_id = 'amazon.titan-embed-text-v2:0',
    ) #create a Titan Embeddings client

    # path to a sample food bank strategic plan PDF
    pdf_path = "~/environment/workshop/completed/rag_chatbot/output.pdf"

    loader = PyPDFLoader(file_path=pdf_path) #load the pdf file

    documents = loader.load()

    text_splitter = RecursiveCharacterTextSplitter( #create a text splitter
        separators=["\n\n", "\n", ".", " "], #split chunks at (1) paragraph, (2) line, (3)
sentence, or (4) word, in that order
        chunk_size=1000, #divide into 1000-character chunks using the
separators above
        chunk_overlap=100 #number of characters that can overlap with previous
chunk
    )

    docs = text_splitter.split_documents(documents) # split our document into
chunks for targeted retreival later on

    # use the FAISS in-memory vector store for demo purposes
    faiss_db = FAISS.from_documents(
        docs, #use the chunked documents
        embeddings, #use Titan embeddings to "vectorize" our documents
    )

    return faiss_db # return the vector database to be cached by the client app
```

5. Add a function that executes the similarity search on the in-memory index given the user's query to return relevant chunks of context and source data.

```
def search_vector_db(index, query):

    # run a similarity search on the FAISS vector database to return context
relevant to the user's query
```

```
    results = index.similarity_search(query)

    # Extract the context chunk results
    contexts = list({doc.page_content for doc in results})

    # Extract the unique sources from the metadata
    unique_sources = list({doc.metadata["source"] for doc in results})

    # returns relevant chunks of information and the unique sources
    return contexts, unique_sources
```

6. Create a memory list to store conversation history. We will append messages into this list and submit through the Converse API request later on.

```
# Instantiate a messages list to store the conversation history. This will preserve context along with the latest message.

messages = []
```

7. Add this function to call bedrock.
    - We're creating a function we can call from the Streamlit front end application.
    - This function will call the search_vector_db function to retrieve relevant context from our in-memory vector store. It injects the results into our user prompt.
    - The function then defines model parameters like temperature and top_k, then submits the user prompt to the Claude 3 Sonnet model using the Converse API. The response is returned to the front end.

```
def invoke_model(index, query):

    # run a similarity search on the FAISS vector database to return context and sources relevant to answer the user's query
    results, sources = search_vector_db(index, query)

    prompt_data = f"""\n\nHuman:
    Answer the following question to the best of your ability based on the context provided.
    Do not include information that is not relevant to the question.
    Only provide information based on the context provided, and do not make assumptions.

    ###

    Question: {query}
```

```
    Context: {results}

    ###

    \n\nAssistant:
    """

    # Define the system prompts to guide the model's behavior and role.
    system_prompts = [{"text": "You are a helpful assistant. Keep your answers short and succinct."}]

    # Format the user's message as a dictionary with role and content
    message = {"role": "user", "content": [{"text": prompt_data}]}

    # Append the formatted user message to the list of messages.
    messages.append(message)

    # Set the temperature for the model inference, controlling the randomness of the responses.
    temperature = 0.5

    # Set the top_k parameter for the model inference, determining how many of the top predictions to consider.
    top_k = 200

    # Call the converse method of the Bedrock client object to get a response from the model.
    response = bedrock.converse(
        modelId="anthropic.claude-3-sonnet-20240229-v1:0",
        messages=messages,
        system=system_prompts,
        inferenceConfig={"temperature": temperature},
        additionalModelRequestFields= {"top_k": top_k}
    )

    # Extract the output message from the response.
    output_message = response['output']['message']

    # Append the output message to the list of messages.
    messages.append(output_message)

    # Print the message output to review how the conversation history builds after each turn
    print("Message History: ", messages)
    print("##############################")
```

```
    # Return the text of the model's response, along with the sources, to the
frontend
    return output_message['content'][0]['text'], sources
```

8. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Nice! You are done with the backing library. Now we will create the front-end application.

Create the Streamlit front-end app

1. In the same folder as your lib file, open the file **rag_chatbot_app.py**

2. Import the streamlit library and import the create_vector_db and invoke_model functions from our library script.
    - These statements allow us to use Streamlit elements and call functions in the backing library script.

```
import streamlit as st
from rag_chatbot_lib import create_vector_db, invoke_model
```

3. Add the page title and configuration.
    - Here we are setting the page title on the actual page and the title shown in the browser tab.

```
st.set_page_config(page_title="Chatbot") #HTML title
st.title("Amazon Bedrock RAG Chatbot with the Converse API") #page title
```

4. Add the message history to the session cache.
    - This allows us to maintain a unique chat memory per user session. Otherwise, the chatbot won't be able to remember past messages to display on the front end UI.
    - In Streamlit, session state is tracked server-side. If the browser tab is closed, or the application is stopped, the session and its chat history will be lost. In a real-world application, you would want to track the chat history in a database like Amazon DynamoDB

```
# configuring values for session state

if "messages" not in st.session_state:
```

```
    st.session_state.messages = []
```

5. Add the for loop to render previous chat messages in the UI.
    - Re-render previous messages based on the "messages" session state object.

```
# Streamlit is stateless, so we need to re-render our historical messages and sources from our preserved session state

for message in st.session_state.messages:

    with st.chat_message(message["role"]):

        st.markdown(message["content"])

        if message["role"] == "assistant":

            with st.expander("Sources"):
                st.write(message["sources"])
```

6. Create our FAISS in-memory vector store by calling the create_vector_db function in our backing library script.

```
if 'vector_db' not in st.session_state: # see if the vector store hasn't been created yet

    with st.spinner("Indexing document..."): #show a spinner while the code in this with block runs

        st.session_state.vector_db = create_vector_db() #retrieve the vector database through the supporting library and store in the app's session cache
```

7. Add the input elements.
    - We use the if block below to handle the user input. See the in-line comments below for more details.

```
# evaluating st.chat_input and determining if a question has been input

if question := st.chat_input("Ask me about anything about your document..."):

    # with the user icon, write the question to the front end
    with st.chat_message("user"):
        st.markdown(question)
```

```
    # append the question and the role (user) as a message to the session state
    st.session_state.messages.append({"role": "user", "content": question})

    # respond as the assistant with the answer
    with st.chat_message("assistant"):

        # making sure there are no messages present when generating the answer
        message_placeholder = st.empty()

        # passing the cached vector database and the user question into
invoke_model function from our library script, then return the response and
relevant sources
        answer, sources = invoke_model(index=st.session_state.vector_db,
query=question)



        # writing the answer to the front end
        message_placeholder.markdown(f"{answer}")

        with st.expander("Sources"):
            st.write(sources)

    # Append the assistant's response and sources to the session state
    # When streamlit re-renders, it will use this stored history to display past
messages to the UI
    st.session_state.messages.append({"role": "assistant", "content": answer,
"sources": sources})
```

8. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Magnificent! Now you are ready to run the application!

Run the Streamlit app

1. Select the **bash terminal** in AWS Cloud9 and change directory. If you don't see a bash terminal open, select **Window > New Terminal** from the Cloud9 toolbar.

```
cd ~/environment/workshop/labs/rag_chatbot
```

Just want to run the app?

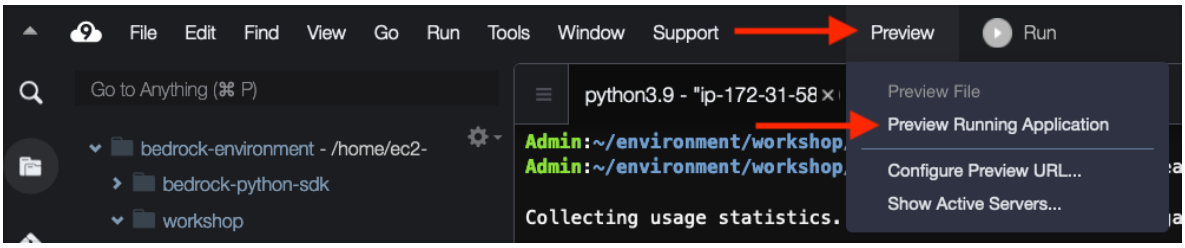
Expand here & run this command instead

2. Run the streamlit command from the terminal.

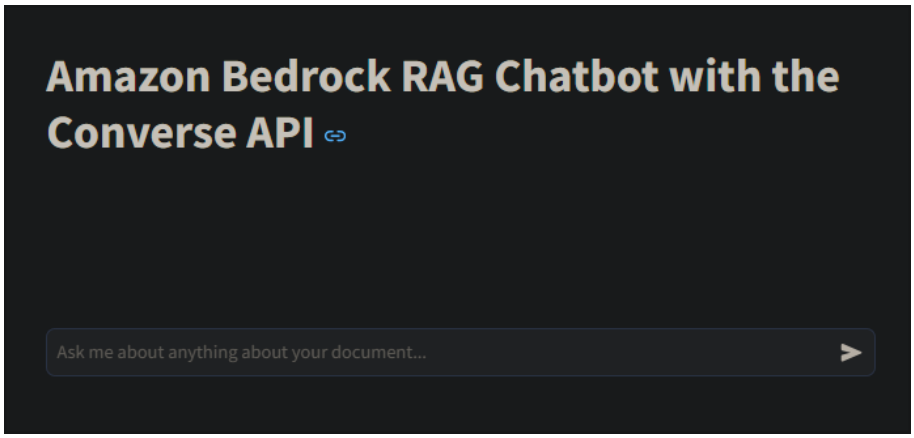
```
streamlit run rag_chatbot_app.py --server.port 8080
```

Ignore the Network URL and External URL links displayed by the Streamlit command. Instead, we will use AWS Cloud9's preview feature.

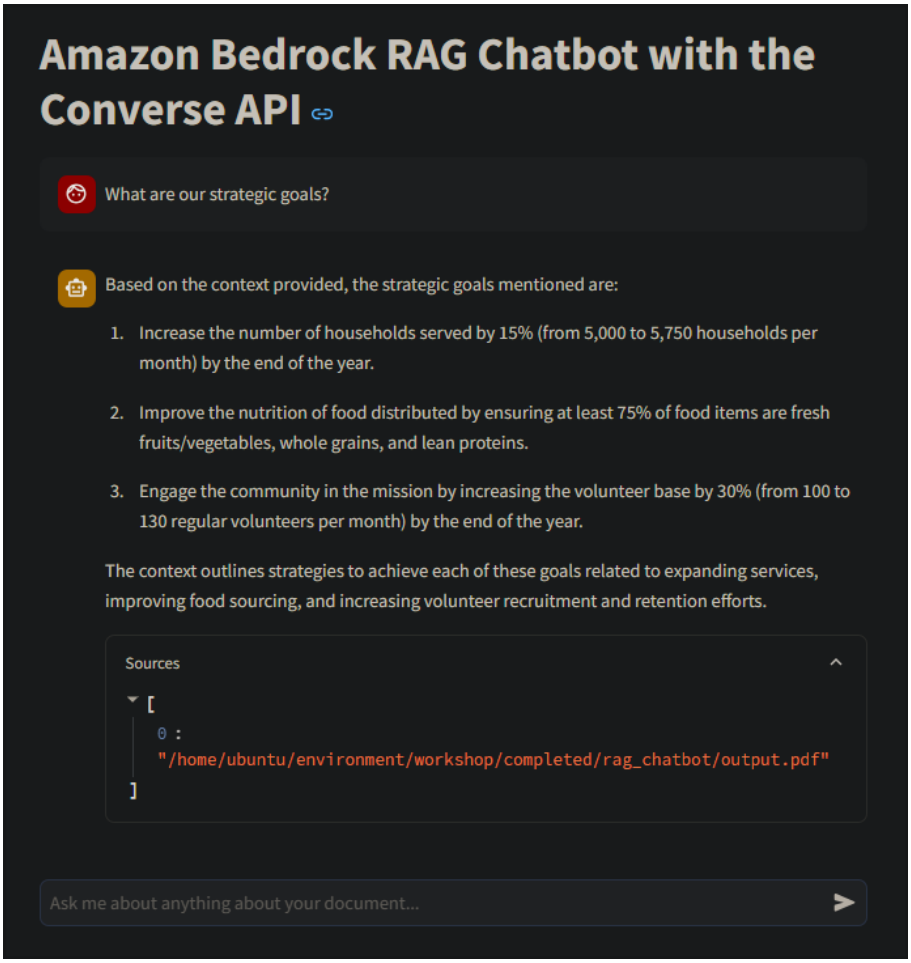
3. In AWS Cloud9, select **Preview** -> **Preview Running Application**.



You should see a web page like below:



4. Try out some prompts and see the results. The chatbot will answer questions based on the sample food bank strategic plan that we added to our in-memory vector store.
    - What are our strategic goals?
    - How will we retain volunteers?
    - How will we measure success of our food distribution goals?



Multilingual Queries

The Amazon Titan Embeddings model supports more than 25 languages, meaning that as your documents are embedded into a vector store they can be retrieved using multilingual prompts. Even though your source documents might be in one language, the model will translate source information to match the prompt language. Try this prompt in various languages.

- What is our strategy for volunteering?
- ¿Cuál es nuestra estrategia para el voluntariado?
- 我们的志愿服务策略是什么?

Natural Boundary

Using Retrieval-Augmented Generation (RAG) can help enforce a natural boundary of information by restricting the model to your supplied documents for generating answers. Try this prompt below to see how the model handles questions that are not covered in the supplied documents:

- What is the capital of Nebraska?

5. Close the preview tab in AWS Cloud9. Return to the terminal and press Control-C to exit the application.

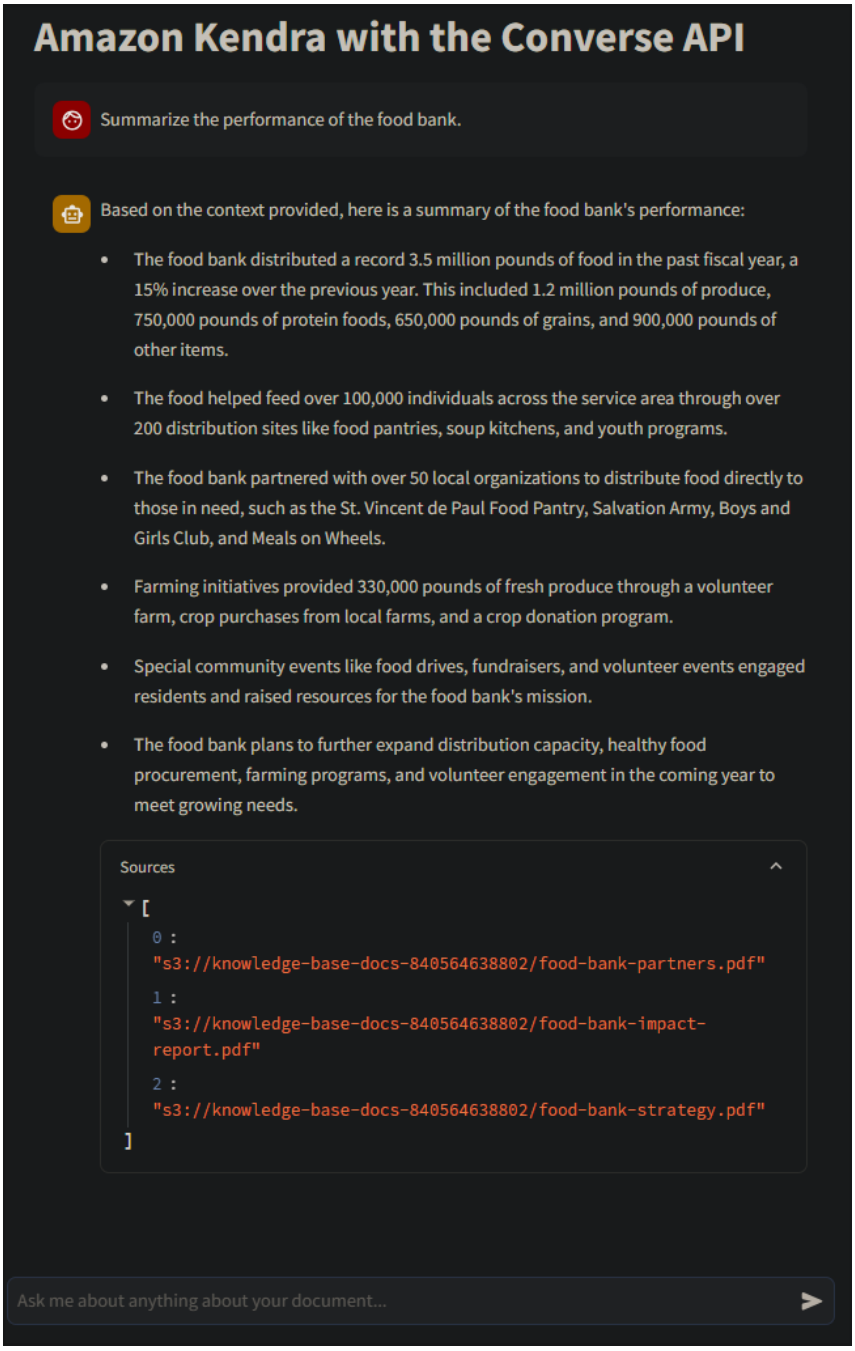
Congratulations!
You have successfully built a RAG chatbot with Bedrock, LangChain, and Streamlit!

Amazon Kendra

Please complete the Running at an AWS-facilitated event steps before starting this lab.

Lab introduction

Final product:



In this lab, we will build a simple question & answer application with Anthropic Claude, LangChain, and Streamlit. We will use Amazon Kendra

as our index to use with the **Retrieval-Augmented Generation** (RAG) portion of this lab.
Large language models are prone to **hallucination**, which is just a fancy word for making up a response. To correctly and consistently answer questions, we need to ensure that the model has real information available to support its responses. We use the RAG pattern to make this happen.
With Retrieval-Augmented Generation, we first pass a user's prompt to a data store. We could also create a numerical representation of the prompt using Amazon Titan Embeddings to pass to a vector database. We then retrieve the most relevant content from the data store to support the large language

model's response.
In this lab, we will use Amazon Kendra

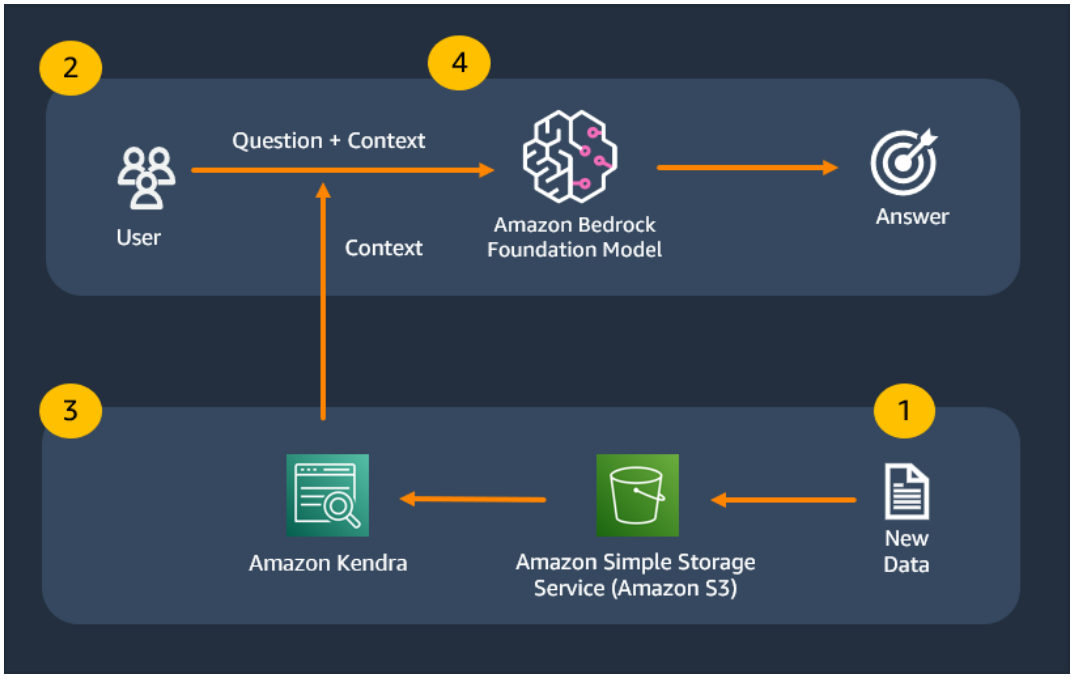
as our document store. Amazon Kendra is an intelligent search service that uses natural language processing and advanced machine learning algorithms to return specific answers to search questions from your data. While it's not a formal vector store, it is still a common index for RAG pipelines and provides simple document ingestion and web crawling capabilities.

Use cases

The Retrieval-Augmented Generation pattern is good for the following use cases:

- Question & answer, supported by specialized knowledge or data
- Intelligent search

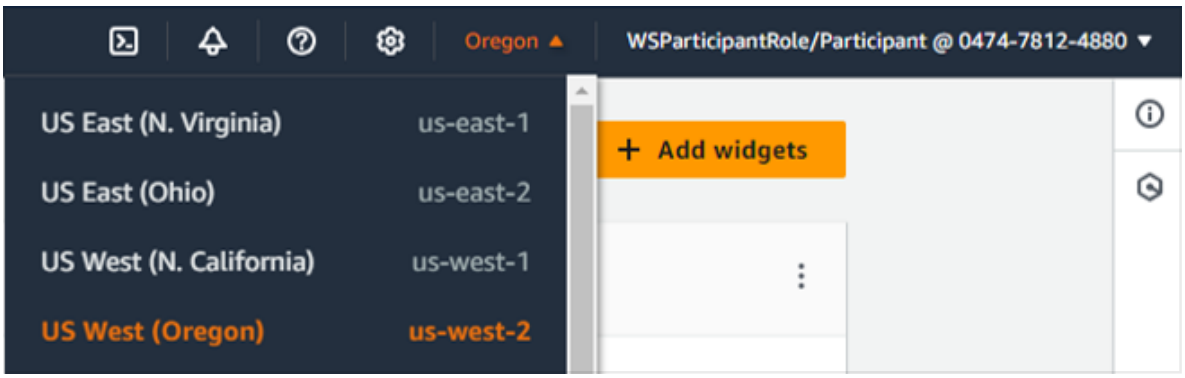
Architecture



1. Documents are added to an Amazon S3 bucket. The S3 bucket is used as a data source for the Amazon Kendra index.
2. The user submits a question.
3. The user's question is submitted to our Amazon Kendra index where relevant documents are retrieved from the search.
4. The combined content from the search results + the original question are then passed to the large language model to generate the best answer

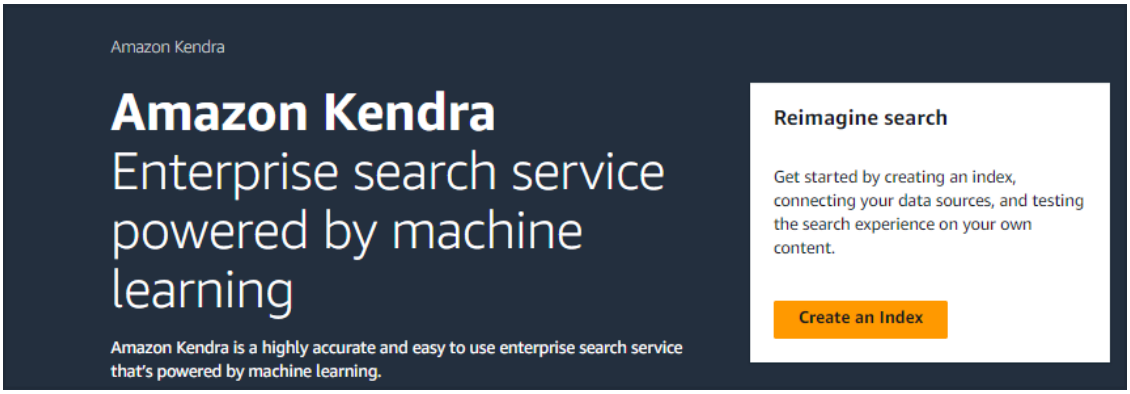
Create an Amazon Kendra Index

Important - Region Selection
Ensure that you're in the us-west-2 region (Oregon) when creating your Kendra index.



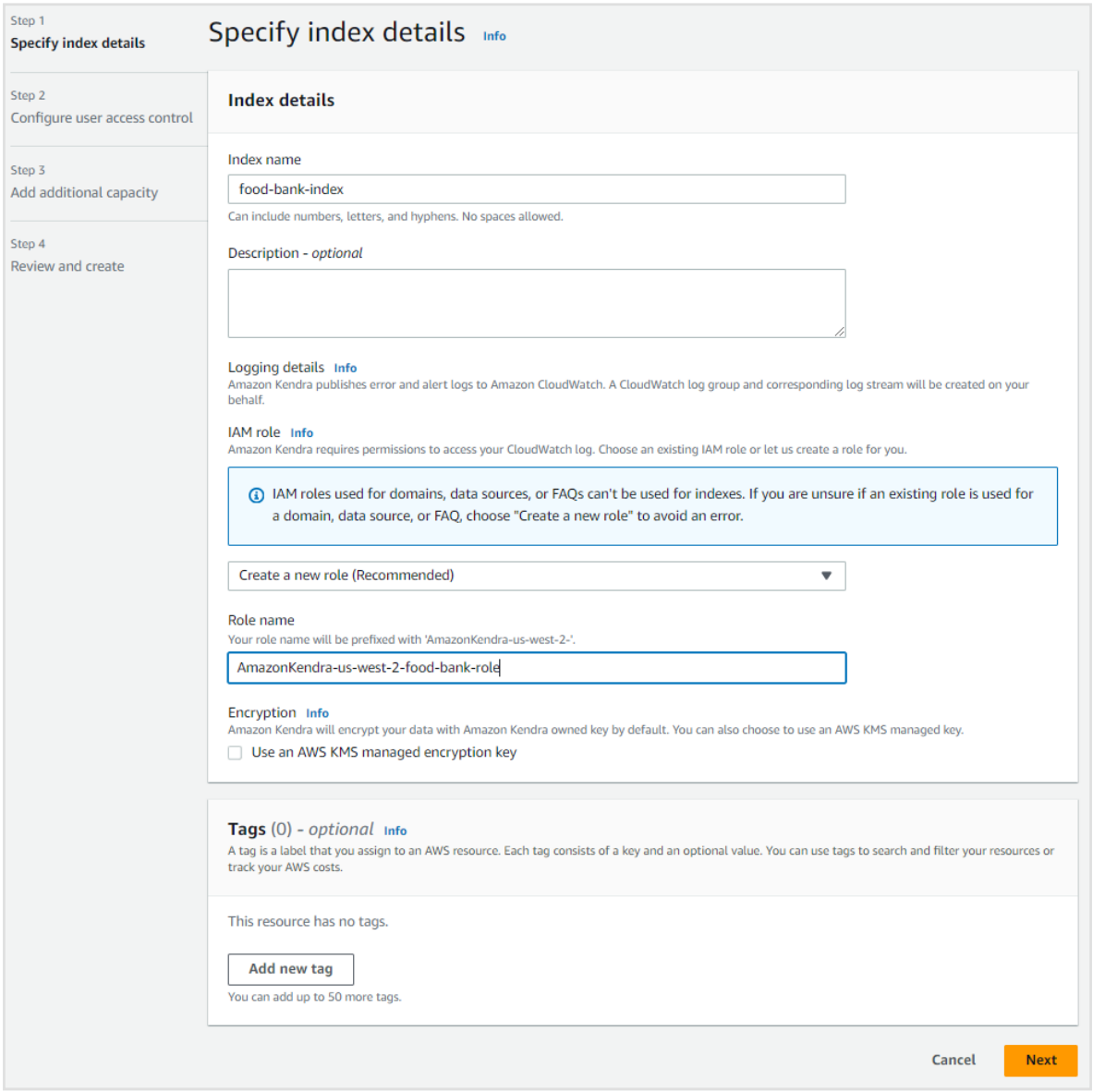
An Amazon S3 bucket with several PDF documents related to our food bank has been created on your behalf. You are ready to create a Kendra Index from your document store!

1. Find **Amazon Kendra** by searching in the AWS console.
2. From the home page, select **Create an Index**.



3.

4. Add a name (exp. food-bank-index)
5. Under **IAM role** select **Create a new role (Recommended)**
6. For **Role name** add a suffix (exp. food-bank-role).

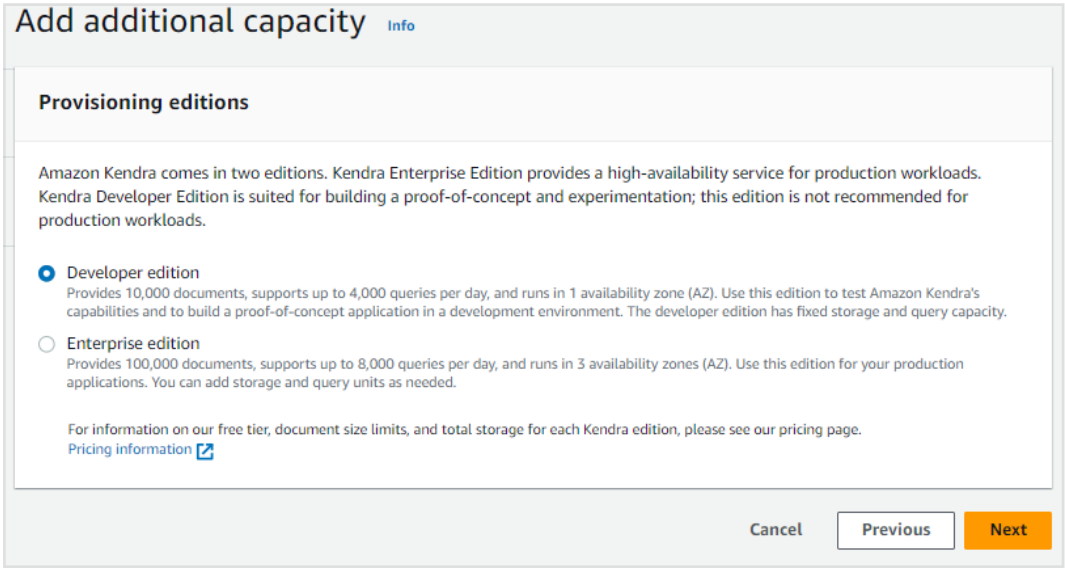


7.

8. Choose **Next**

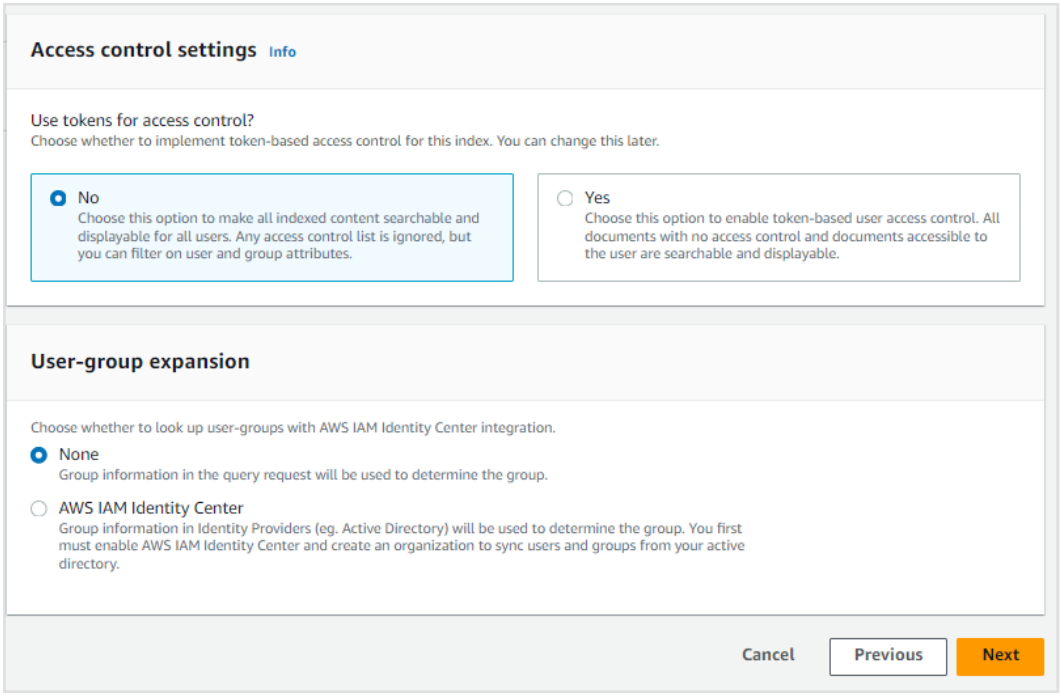
NOTE: You may see red error messages commenting on missing permissions for KMS. You can safely ignore these.

7. Select the **Developer edition** as we are provisioning a sample index for testing. Choose **Next**.



8.

9. Keep all other settings as default and choose **Next**.



10.

11. Confirm your index details and choose **Create**.

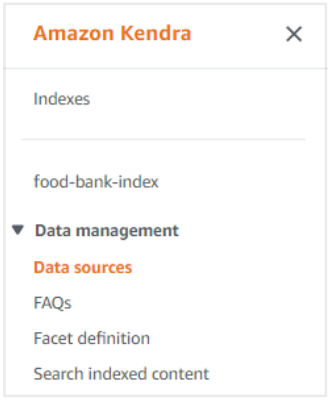
It may take 10-15 minutes to create the index. You are free to explore the other labs while you wait for the index to provision.

Create a data source

After the Kendra index is created, you need to add a data source and sync that

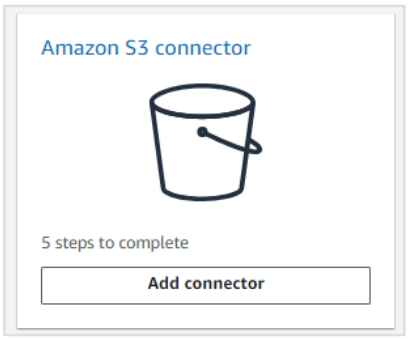
source to the index.

1. From the Amazon Kendra side menu, select **Data sources** under **Data Management**.



2.

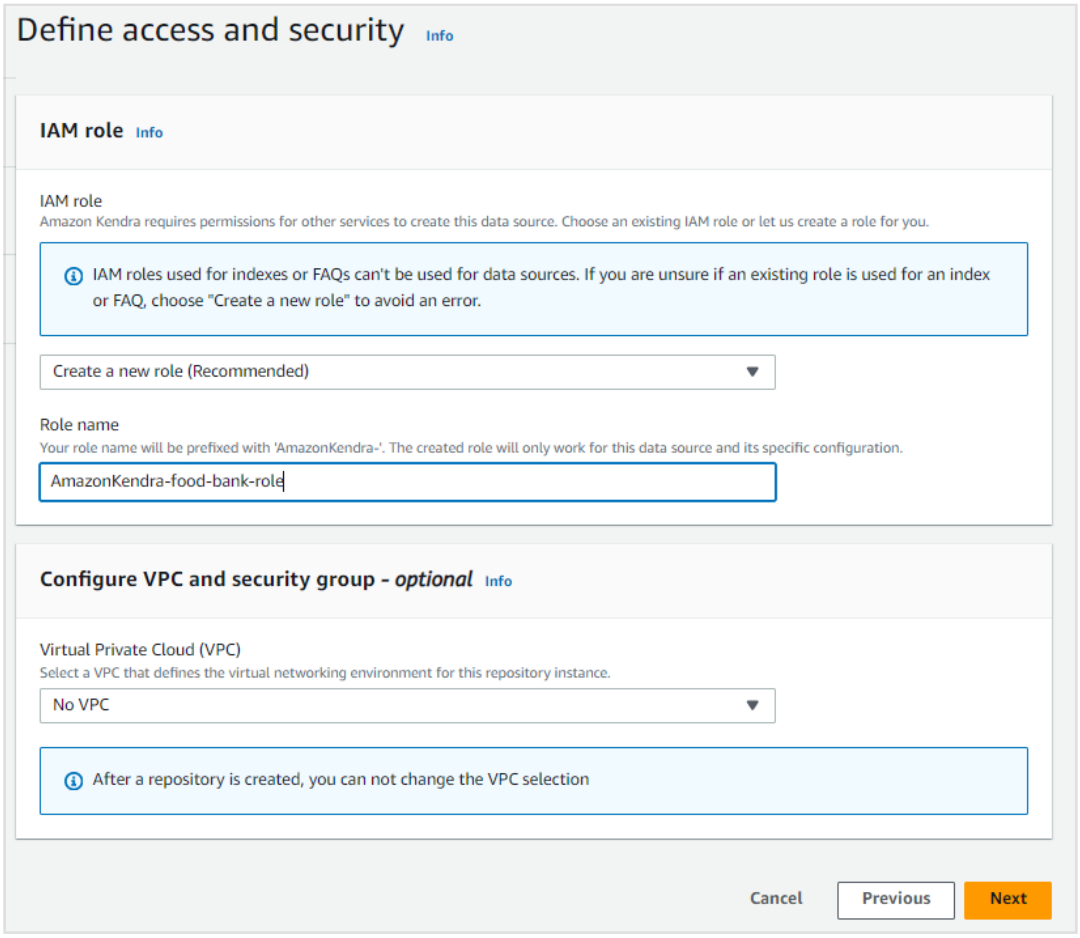
3. Review the list of available data sources. These connectors provide a simple entrypoint to ingest data into a Kendra index.
4. Scroll down or use the search feature to find the **Amazon S3 connector** data source and choose **Add connector**.



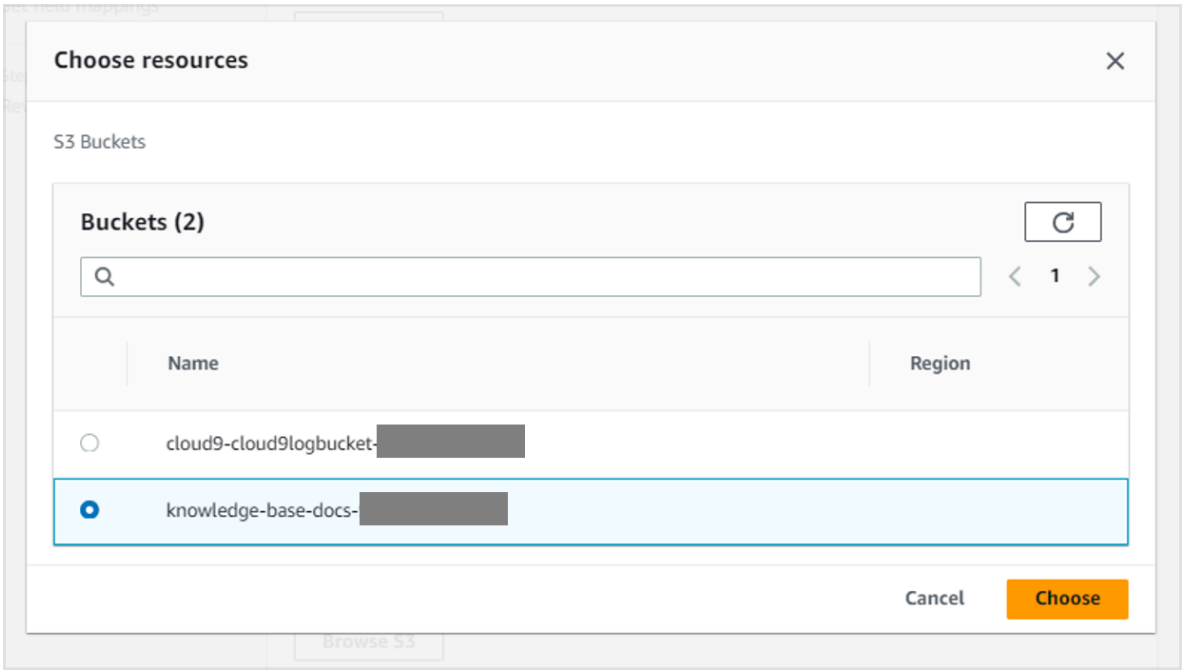
5.

6. Add a Data source name (exp. food-bank-docs) and choose **Next**.
7. Under **IAM role** select **Create a new role (Recommended)**.
8. For **Role name** add a suffix (exp. food-bank-role).
9. Choose **Next**.

NOTE: You may see red error messages commenting on missing permissions for ec2. You can safely ignore these.

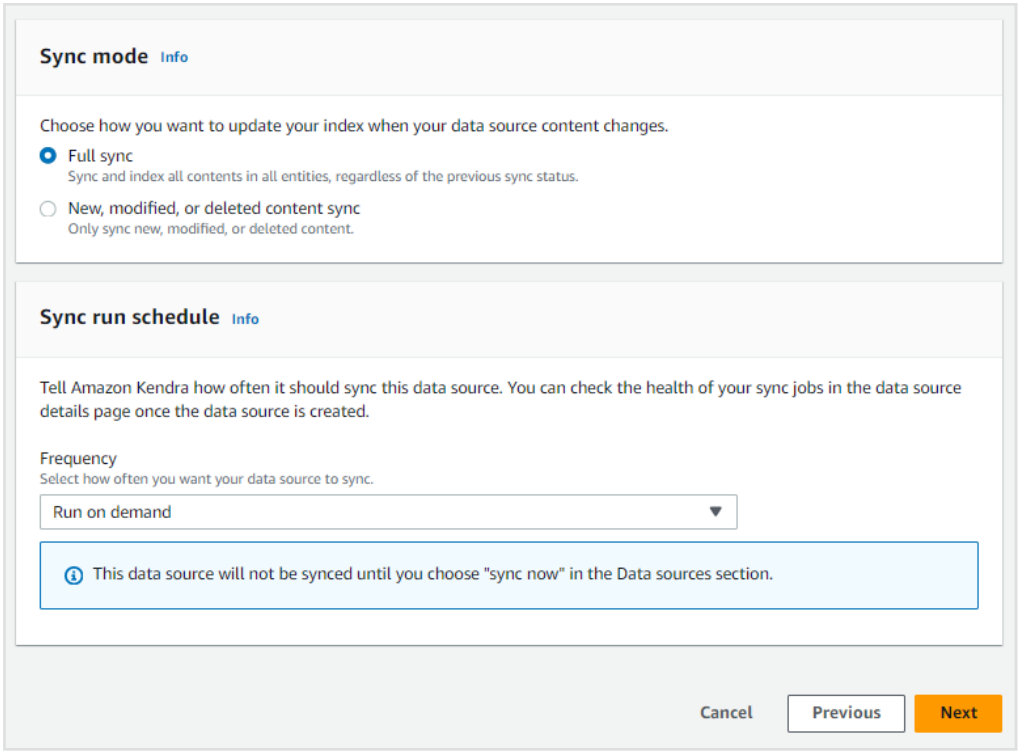


8. under **Enter the data source location**, select **Browse S3** and select the radio button next to the bucket that starts with **knowledge-base-docs**. Select **choose**.



9.

10. Under **Sync mode** select **Full sync**. These settings are useful if you have a growing document store that you want incrementally sync to your index. In this case, we are only syncing once so we select the Full sync.
11. Under **sync run schedule** select **Run on demand** for the frequency. These settings are useful if you have a growing document store that you want to sync on a schedule.
12. Choose **Next**.



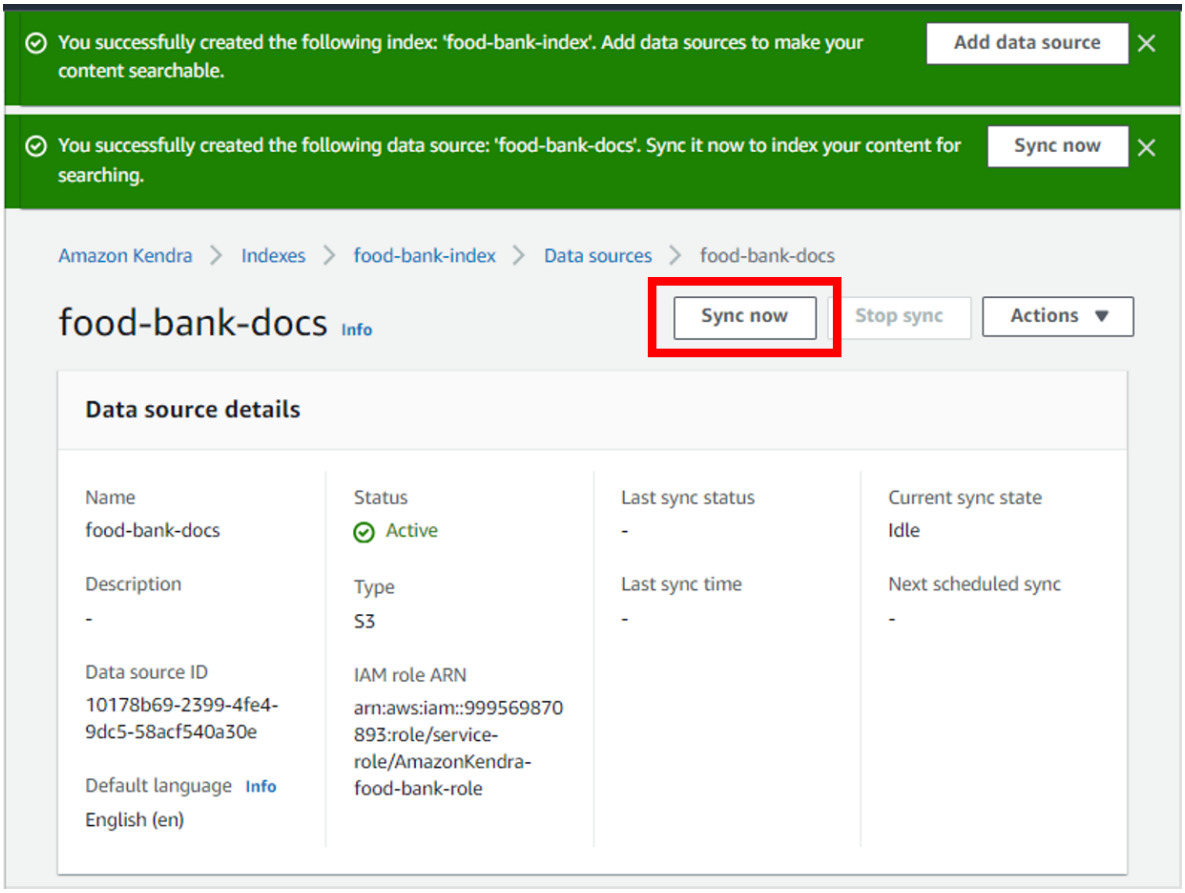
13.

14. Leave the default S3 field mappings and choose **Next**.
15. Review your selections and choose **Add data source**. It will take about 1 minute to create the data source from your Amazon S3 bucket.

Syncing your data source

Once your data source has successfully created, you need to sync the data source to your index.

1. Choose **Sync now** from the data source details page. This will take 3-5 minutes to crawl your documents in Amazon S3 and sync the data to the Kendra index.

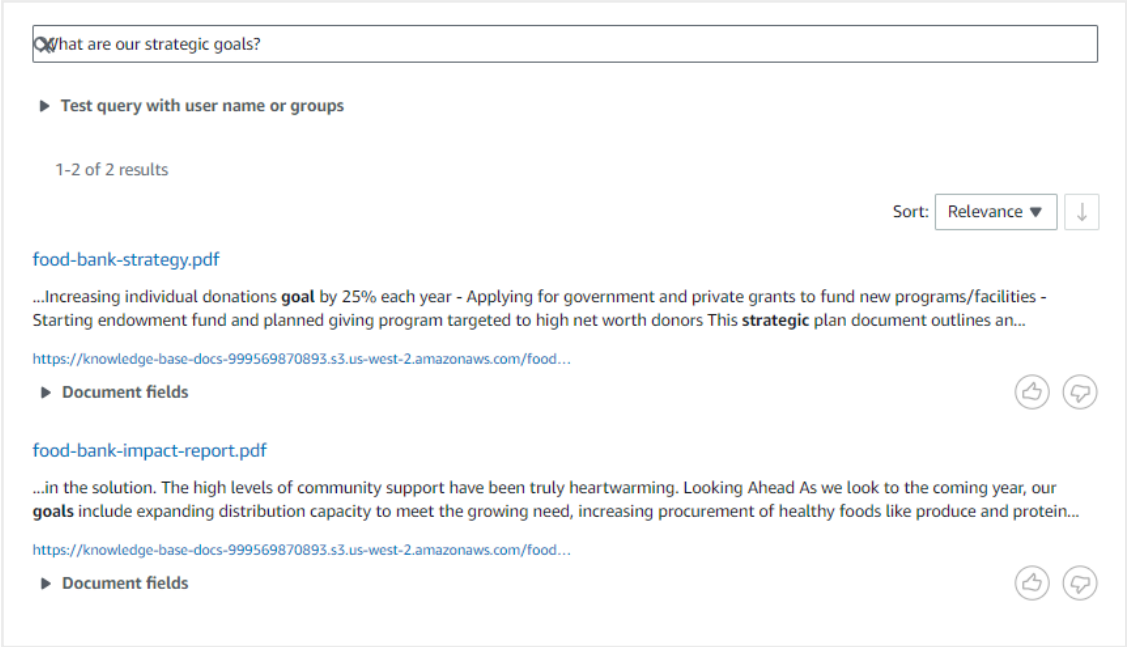


2.

Once the sync has completed, your Kendra index can be tested by selecting **Search indexed content** from the side menu under **Data management**.

2. Submit a test search query and review the documents that the Kendra index returns.

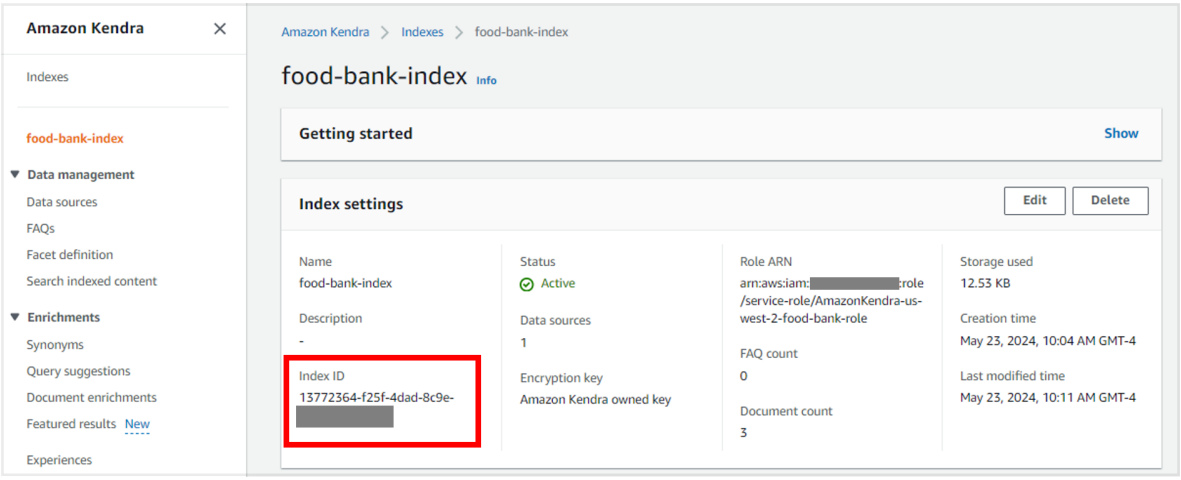
- What are our strategic goals?



-

Retrieve the Index ID

1. Navigate back to the index details page by clicking on the name of your index at the top of the side menu.
2. Take note of the **Index ID** value. We will use this in our python script to point to Kendra as our document retriever in our RAG application.



3.

RAG Chatbot with Amazon Kendra

Please complete the Running at an AWS-facilitated event steps before starting this lab.

Please complete the Launching Cloud9 section before starting this lab.

Lab introduction

In this lab, we will build a simple chatbot with Amazon Bedrock, Amazon Kendra and Streamlit. We will use the Amazon Bedrock Converse API

which simplifies development of chat experiences using models invoked through Bedrock.
Amazon Bedrock (and LLMs in general) don't have any concept of state or memory. Any chat history has to be tracked externally and then passed into the model with each new message. The Converse API supports conversational history in a structured way as part of the API request, reducing the complexity for multi-turn conversations.
We also want to supplement the model's underlying data with external knowledge through **Retrieval-Augmented Generation** (RAG). We'll Amazon Kendra as our index to support retrieval of relevant context, relying on data that has been pre-loaded into an Amazon S3 bucket.
You can build the application code by copying the code snippets below and pasting into the indicated Python file.

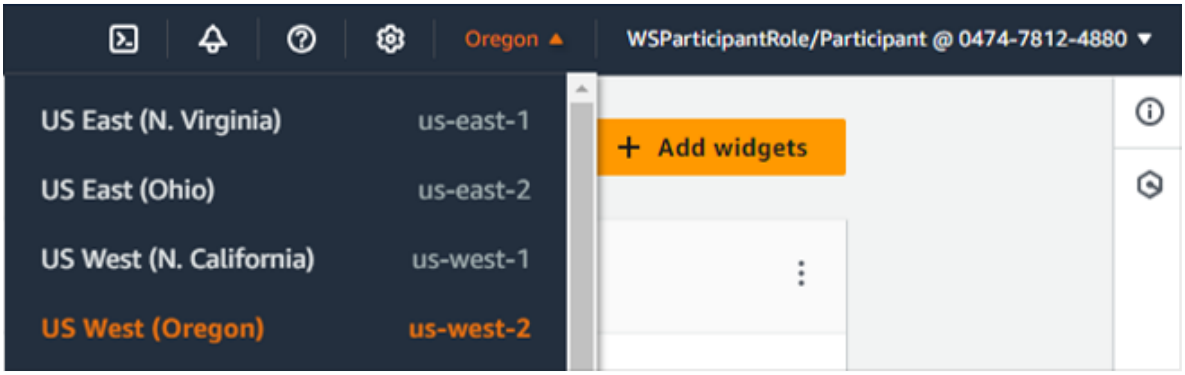
Just want to run the app?
You can jump ahead to run a pre-made application.

Architecture

- **Create an Amazon Bedrock Knowledge Base**

Create an Amazon Bedrock Knowledge Base

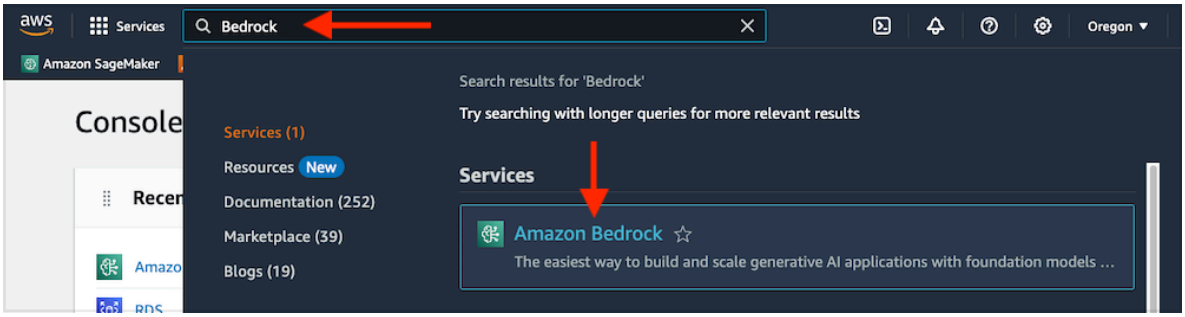
Important - Region Selection
Ensure that you're in the us-west-2 region (Oregon) when creating your Bedrock Knowledge Base.



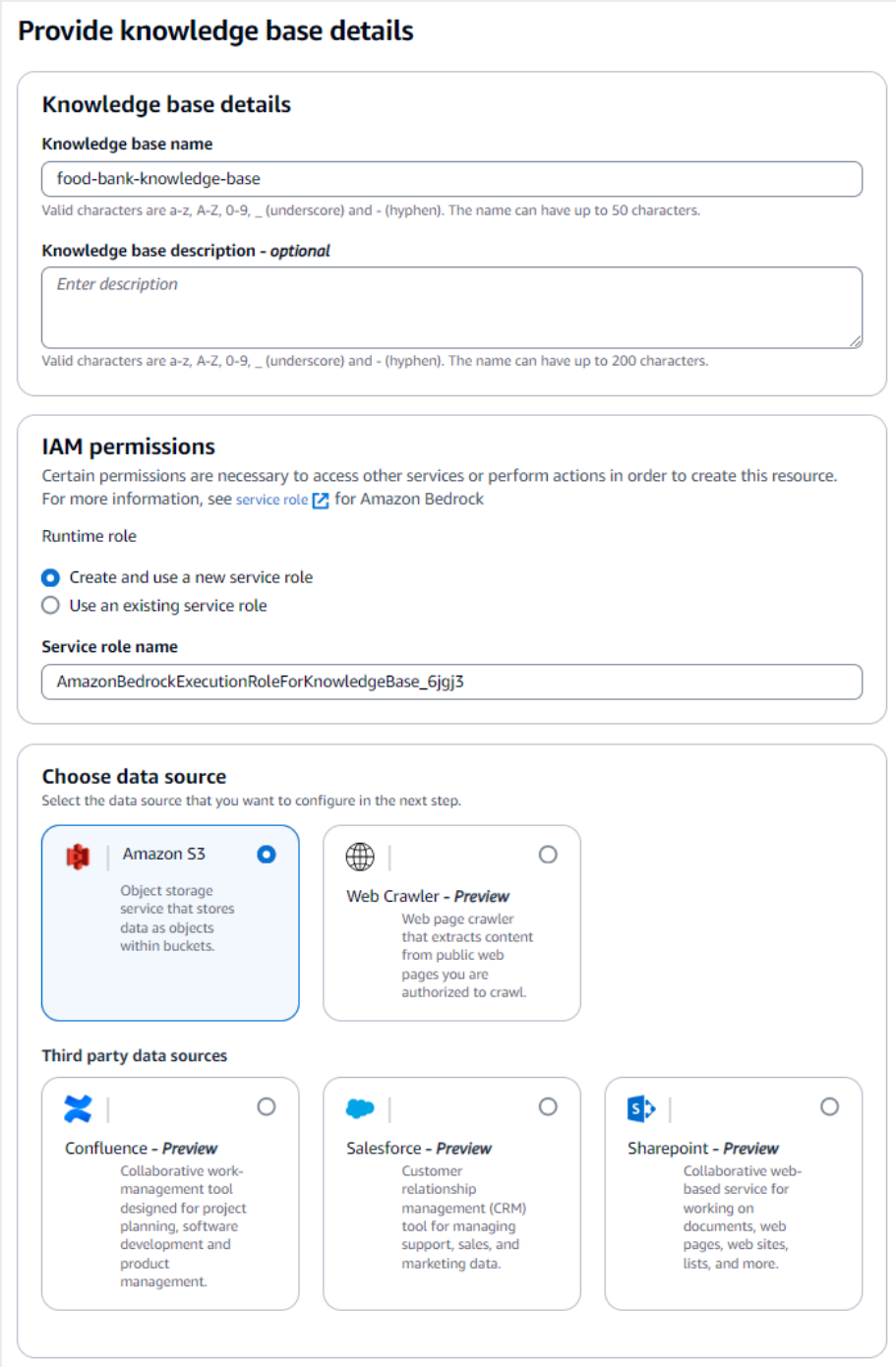
An Amazon S3 bucket with several PDF documents related to our food bank has been created on your behalf. You are ready to create a knowledge base from

your document store!

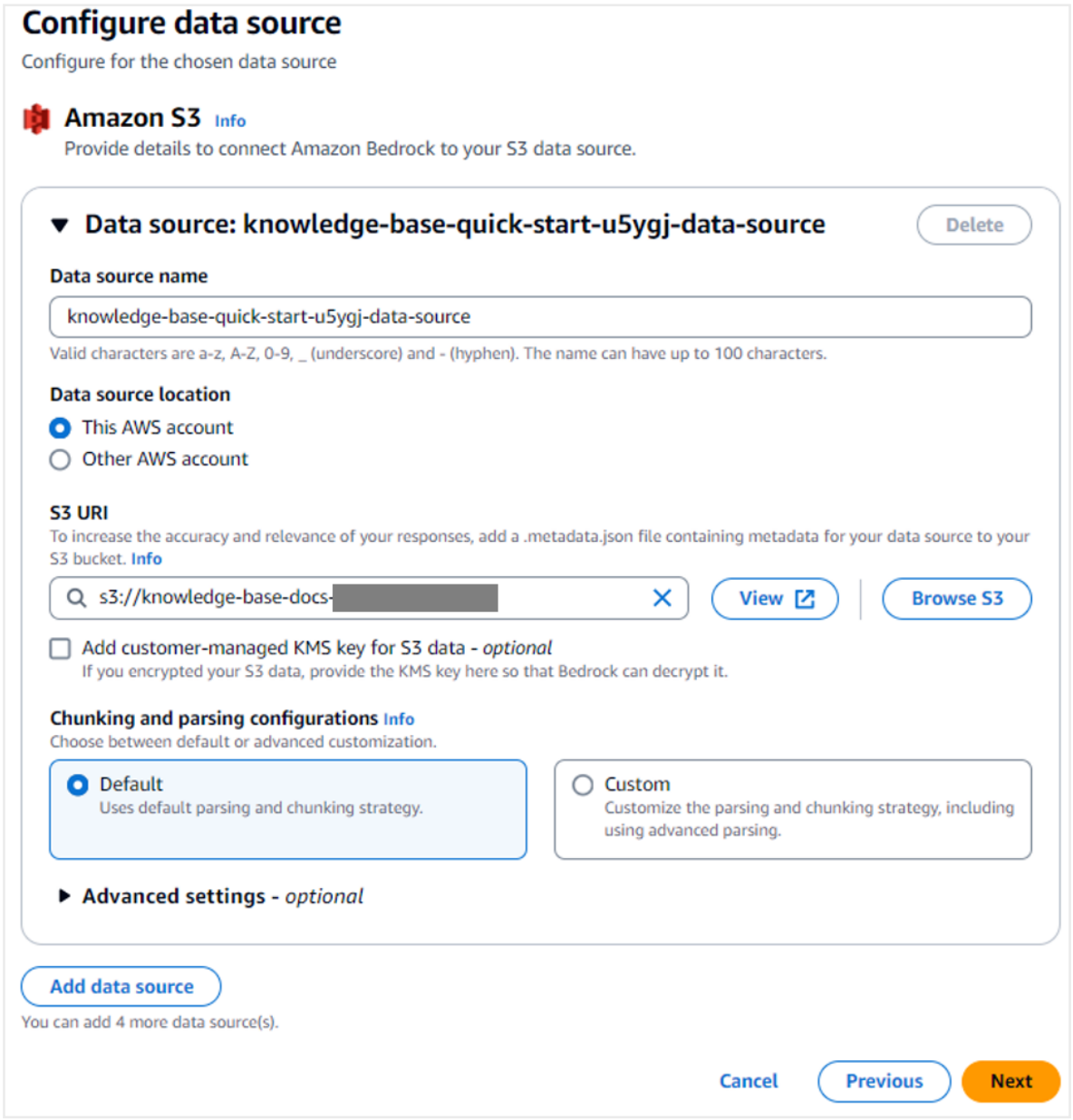
1. Find **Amazon Bedrock** by searching in the AWS console.



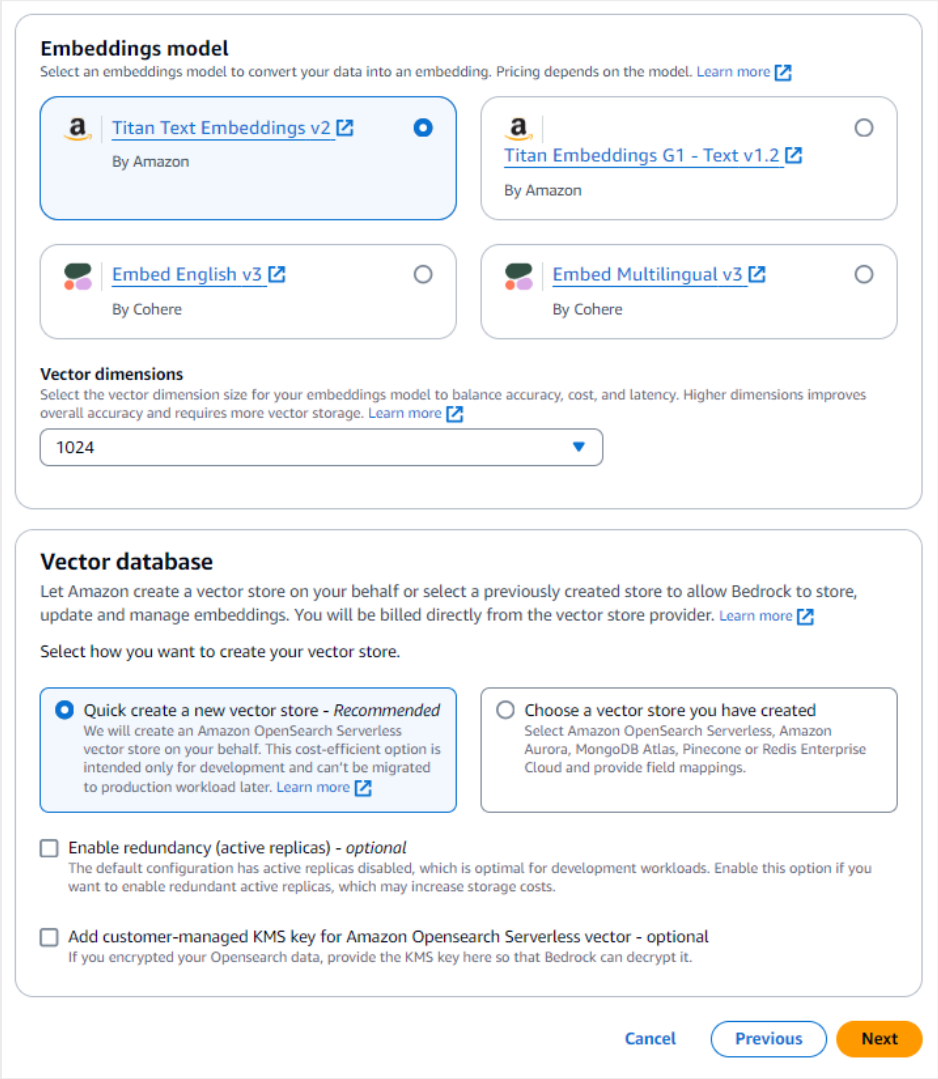
2. Expand the side menu.
3. From the side menu, select **Knowledge Bases** under the **Builder tools** section.
4. Select **Create knowledge base**.
5. Add a name (exp. food-bank-knowledge-base)
6. Keep the IAM permissions runtime role as **Create and use a new service role**. The name will be populated by default.
7. When choosing a data source, select **Amazon S3**. We will use the S3 bucket that has already been created on your behalf.
8. Keep all other settings as default, and choose **Next**.



9. Select **Browse S3** and select the radio button next to the bucket that starts with **knowledge-base-docs**. Select **Choose**.
10. Keep chunking configuration as the default, then choose **Next**.



11. We need to select an embeddings model that converts our documents into vector format before storing in a vector database. Choose the **Titan Text Embeddings V2** option. Ensure the model access has been granted following the steps from the Amazon Bedrock setup section.
12. Under vector database, select **Quick create a new vector store**. This will create an OpenSearch Serverless collection on your behalf for storing your vectorized data from S3.

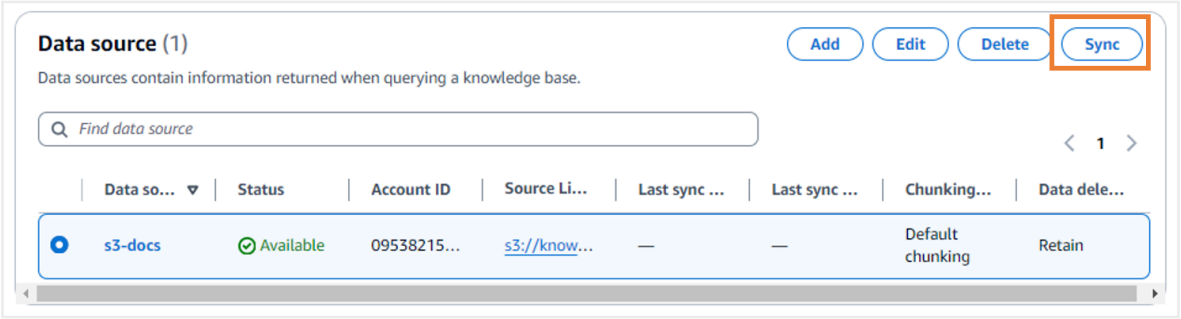


13. Choose **Next**.
14. Review the configurations, select **Create knowledge base** and stay on the page for 2-3 minutes.

Do not close or navigate away from the page while the knowledge base is being created. If you close or leave the page, the knowledge base may not be created and you will need to start the process over again.
You are free to open a new browser tab and explore the other labs while you wait for the knowledge base to provision.

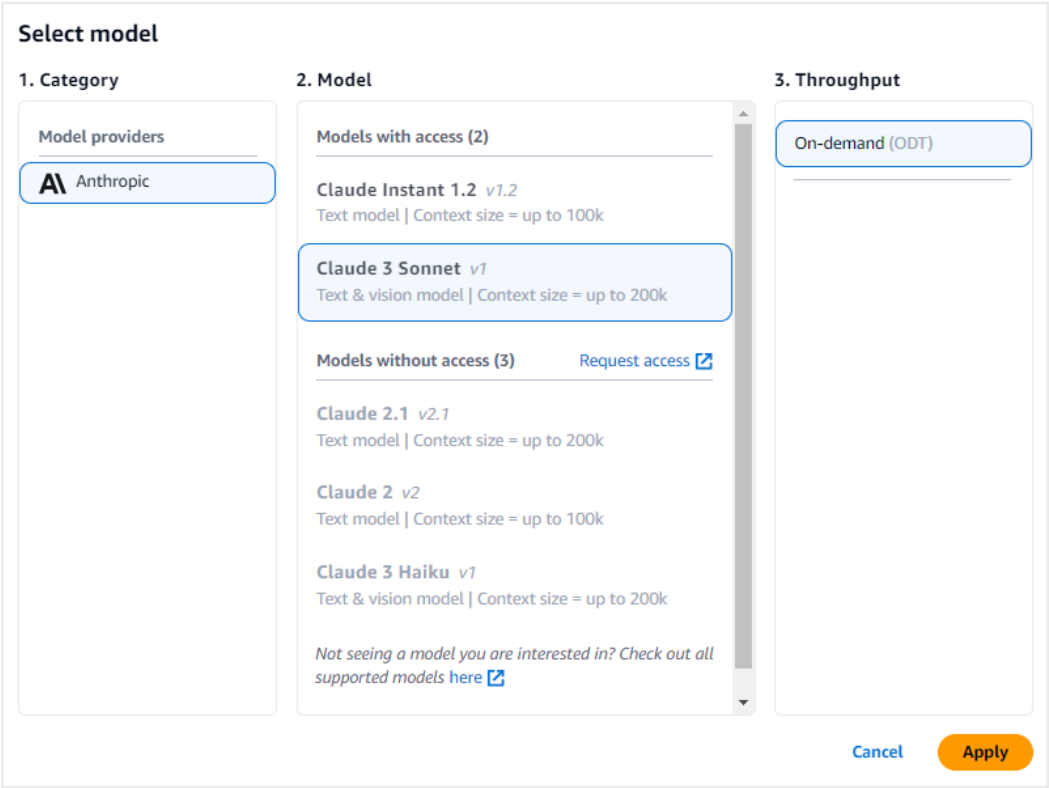
Syncing your data source

15. Once your knowledge base is completed, you need to sync the data from your Amazon S3 bucket to your vector database.
16. Select the radio button next to your S3 data source in the **Data sources** pane.
17. Select **Sync** to begin the process. This will take 1-2 minutes to sync your documents.



18.

19. After the sync is complete, you can sample the output in the test pane on the right.
20. Choose **select model** and choose a model to interact with your knowledge base. For this example, choose **Claude 3 Sonnet** and select **Apply**.



21.

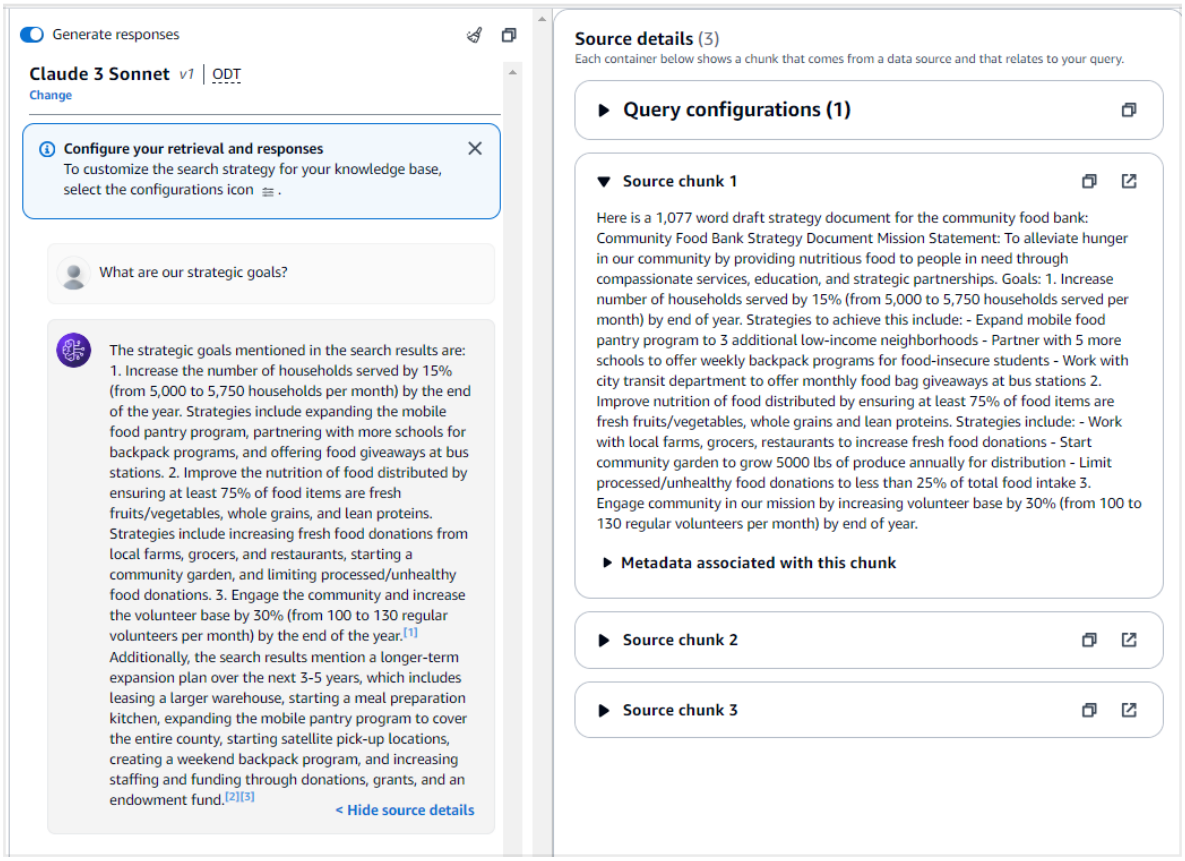
22. Paste the sample question below into the message input and select **Run**.

- What are our strategic goals?

21. Bedrock Knowledge bases will run a RAG process (embed your question to a vector representation, run semantic search against your vector database to collect relevant context, use the context to answer

your question) and return an answer to your question.

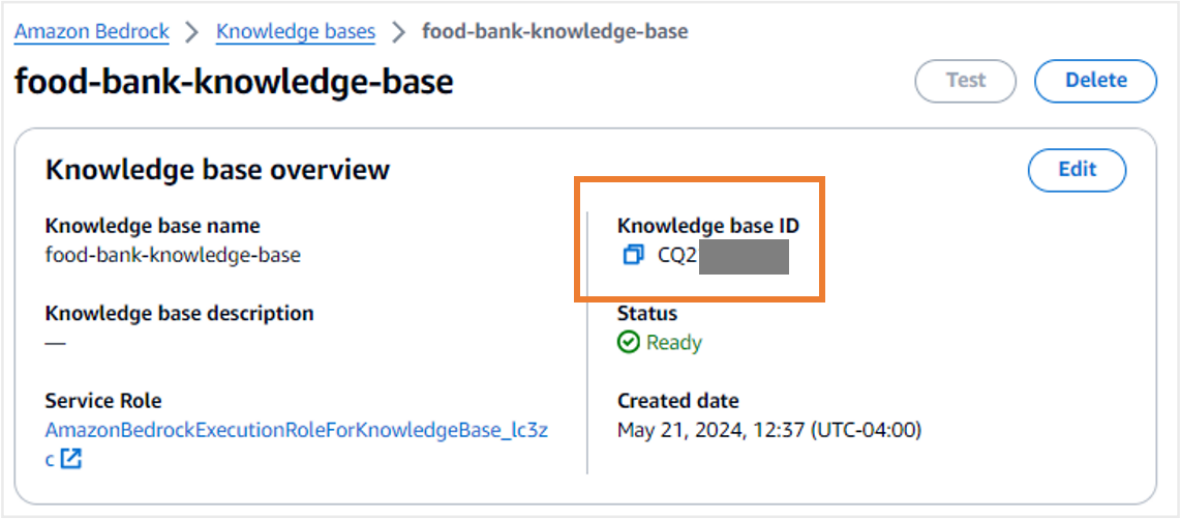
22. Select **Show details** to review the source chunks and metadata used to answer your question.



23.

Retrieve the knowledge base ID

23. Close the test pane to return to the Knowledge base overview.
24. Take note of the **knowledge base ID** value. We will use this in our python script to submit queries to our bedrock knowledge base and return the response in our streamlit application.



25.

Previous
Next

- **RAG Chatbot with Amazon Bedrock Knowledge Bases**

RAG Chatbot with Amazon Bedrock Knowledge Bases

Please complete the Running at an AWS-facilitated event steps before starting this lab.

Please complete the Launching Cloud9 section before starting this lab.

Lab introduction

In this lab, we will build a simple chatbot with Amazon Bedrock, Knowledge Bases for Amazon Bedrock and Streamlit. We will use the Amazon Bedrock Converse API

which simplifies development of chat experiences using models invoked through Bedrock.

Amazon Bedrock (and LLMs in general) don't have any concept of state or memory. Any chat history has to be tracked externally and then passed into the model with each new message. The

Converse API supports conversational history in a structured way as part of the API request, reducing the complexity for multi-turn conversations.

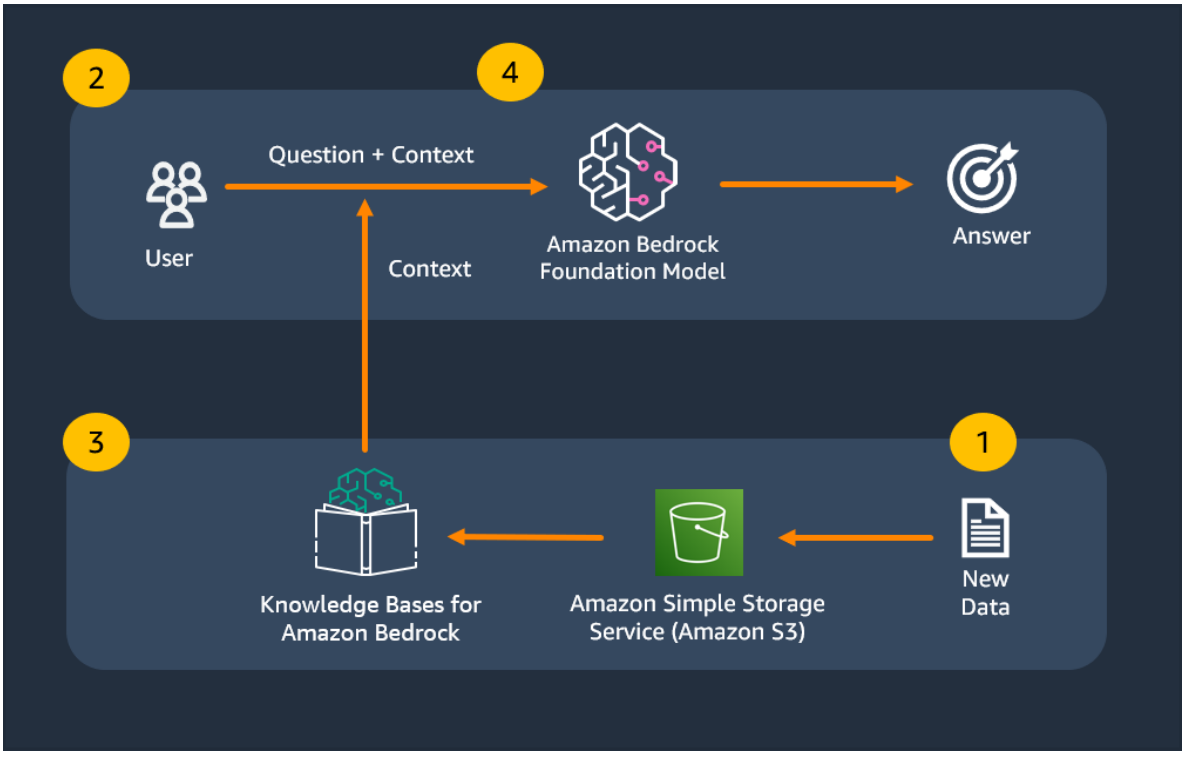
We also want to supplement the model's underlying data with external knowledge through Retrieval-Augmented Generation (RAG). We'll use Amazon Bedrock knowledge bases to manage the RAG pipeline for our chatbot, relying on data that has been pre-loaded into an Amazon S3 bucket.

You can build the application code by copying the code snippets below and pasting into the indicated Python file.

Just want to run the app?

You can jump ahead to run a pre-made application.

Architecture



1. Documents are added to an Amazon S3 bucket. Bedrock knowledge bases will break up the documents into chunks of text. The chunks are passed to your selected embeddings model to be converted to vectors. The vectors are then saved to the vector database. In knowledge bases for Amazon Bedrock, the default vector database is Amazon OpenSearch Serverless
   .
2. The user submits a question.
3. Knowledge bases for Amazon Bedrock will convert the question to a vector using your selected embeddings model, then match to the closest vectors in the vector database.

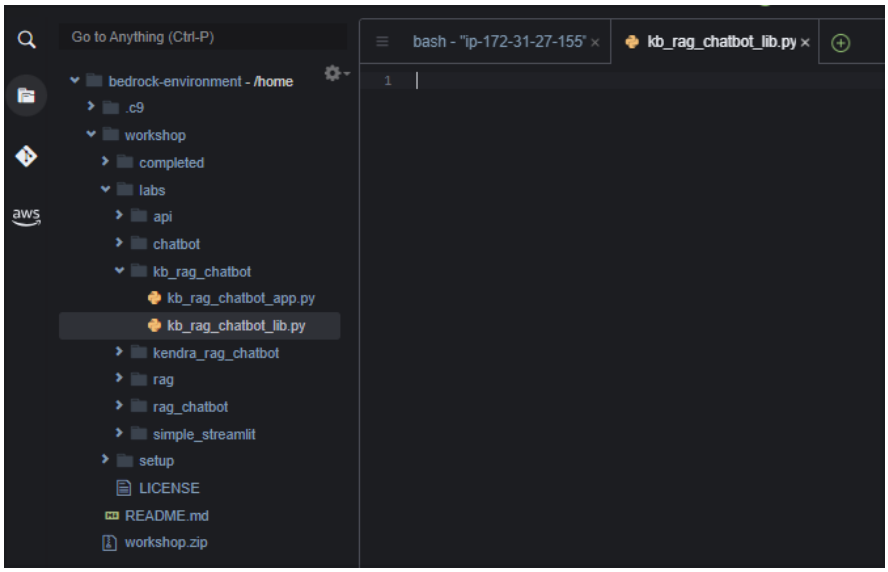
4. The combined content from the matching vectors + the original question are then passed to the large language model to get the best answer.

This application consists of two files: one for the Streamlit front end, and one for the supporting library to make calls to Bedrock.

Create the library script

First we will create the supporting library to connect the Streamlit front end to the Bedrock back end.

1. Launch the Cloud9 environment by following the steps in the Launching Cloud9 section.
2. Navigate to the **workshop/labs/kb_rag_chatbot** folder, and open the file **kb_rag_chatbot_lib.py**



3. Add the import statements.
    - You can use the copy button in the box below to automatically copy its code:

```
import boto3
import os
import json
```

4. Create the AWS SDK clients to use with Bedrock and Knowledge Bases.

```
bedrock = boto3.client('bedrock-runtime') # Creates a Bedrock client
```

```
bedrock_agent_runtime = boto3.client('bedrock-agent-runtime')
```

5. Set the knowledge_base_id from the knowledge base you created previously. If you haven't yet created a knowledge base for Bedrock, return to create an Amazon Bedrock Knowledge Base.

```
# set the knowledge base id from the previously created knowledge base

knowledge_base_id = '<insert your knowledge base ID here>'
```

6. Add a function that executes the similarity search on the Bedrock Knowledge Base given the user's query to return relevant chunks of context and source data.

```
def search_vector_db(query, numberOfResults=5): # getting the contexts for
the query from the knowledge base

    results = bedrock_agent_runtime.retrieve(
        retrievalQuery= {
            'text': query
        },
        knowledgeBaseId=knowledge_base_id,
        retrievalConfiguration= {
            'vectorSearchConfiguration': {
                'numberOfResults': numberOfResults
            }
        }
    )

    # creating lists to store the relevant context chunks and sources from the
Knowledge Base response
    contexts =  list(set(item['content']['text'] for item in
results['retrievalResults']))

    sources = list(set(item['metadata']['x-amz-bedrock-kb-source-uri'] for item
in results['retrievalResults']))

    # returning the contexts and sources
    return contexts, sources
```

7. Create a memory list to store conversation history. We will append

messages into this list and submit through the Converse API request later on.

```
# Instantiate a messages list to store the conversation history. This will preserve context along with the latest message.

messages = []
```

8. Add this function to call bedrock.
    - We're creating a function we can call from the Streamlit front end application.
    - This function will call the search_vector_db function to retrieve relevant context from our Bedrock Knowledge Base. It injects the results into our user prompt.
    - The function then defines model parameters like temperature and top_k, then submits the user prompt to the Claude 3 Sonnet model using the Converse API. The response is returned to the front end.

```
def invoke_model(query):

    # run a similarity search on the Amazon Knowledge Bases for Bedrock to return context relevant and sources to the user's query
    results, sources = search_vector_db(query)

    prompt_data = f"""\n\nHuman:
    Answer the following question to the best of your ability based on the context provided.
    Do not include information that is not relevant to the question.
    Only provide information based on the context provided, and do not make assumptions.

    ###

    Question: {query}

    Context: {results}

    ###

    \n\nAssistant:
    """

    # Define the system prompts to guide the model's behavior and role.
    system_prompts = [{"text": "You are a helpful assistant. Keep your answers short and succinct."}]

    # Format the user's message as a dictionary with role and content
```

```
    message = {"role": "user", "content": [{"text": prompt_data}]}

    # Append the formatted user message to the list of messages.
    messages.append(message)

    # Set the temperature for the model inference, controlling the randomness of
the responses.
    temperature = 0.5

    # Set the top_k parameter for the model inference, determining how many of
the top predictions to consider.
    top_k = 200

    # Call the converse method of the Bedrock client object to get a response
from the model.
    response = bedrock.converse(
        modelId="anthropic.claude-3-sonnet-20240229-v1:0",
        messages=messages,
        system=system_prompts,
        inferenceConfig={"temperature": temperature},
        additionalModelRequestFields= {"top_k": top_k}
    )

    # Extract the output message from the response.
    output_message = response['output']['message']

    # Append the output message to the list of messages.
    messages.append(output_message)

    # Print the message output to review how the conversation history builds
after each turn
    print("Message History: ", messages)
    print("##############################")

    # Return the text of the model's response, along with the sources, to the
frontend
    return output_message['content'][0]['text'], sources
```

9. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Excellent! You are done with the backing library. Now we will create the front-end application.

Create the Streamlit front-end app

1. In the same folder as your lib file, open the file **kb_rag_chatbot_app.py**

2. Import the streamlit library and import the invoke_model function from our library script.
    - These statements allow us to use Streamlit elements and call functions in the backing library script.

```
import streamlit as st #all streamlit commands will be available through the "st" alias
from kb_rag_chatbot_lib import invoke_model
```

3. Add the page title and configuration.
    - Here we are setting the page title on the actual page and the title shown in the browser tab.

```
st.set_page_config(page_title="RAG Chatbot") #HTML title
st.title("Amazon Bedrock Knowledge Bases with the Converse API")
```

4. Add the message history to the session cache.
    - This allows us to maintain a unique chat memory per user session. Otherwise, the chatbot won't be able to remember past messages to display on the front end UI.
    - In Streamlit, session state is tracked server-side. If the browser tab is closed, or the application is stopped, the session and its chat history will be lost. In a real-world application, you would want to track the chat history in a database like Amazon DynamoDB

```
# configuring values for session state

if "messages" not in st.session_state:

    st.session_state.messages = []
```

5. Add the for loop to render previous chat messages in the front end UI.
    - Re-render previous messages based on the "messages" session state object.

```
# Streamlit is stateless, so we need to re-render our historical messages and sources from our preserved session state
```

```
for message in st.session_state.messages:

    with st.chat_message(message["role"]):

        st.markdown(message["content"])

        if message["role"] == "assistant":

            with st.expander("Sources"):
                st.write(message["sources"])
```

6. Add the input elements.
    - We use the if block below to handle the user input. See the in-line comments below for more details.

```
# evaluating st.chat_input and determining if a question has been input

if question := st.chat_input("Ask me about anything about your document..."):

    # with the user icon, write the question to the front end
    with st.chat_message("user"):
        st.markdown(question)

    # append the question and the role (user) as a message to the session state
    st.session_state.messages.append({"role": "user", "content": question})

    # respond as the assistant with the answer
    with st.chat_message("assistant"):

        # making sure there are no messages present when generating the answer
        message_placeholder = st.empty()

        # passing the user question into invoke_model function from our library
script, then return the response and sources
        answer, sources = invoke_model(question)

        # writing the answer to the front end
        message_placeholder.markdown(f"{answer}")

        with st.expander("Sources"):
            st.write(sources)

    # Append the assistant's response and sources to the session state
    # When streamlit re-renders, it will use this stored history to display past
messages to the UI
```

```
    st.session_state.messages.append({"role": "assistant", "content": answer,
"sources": sources})
```

7. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Magnificent! Now you are ready to run the application!

Run the Streamlit app

1. Select the **bash terminal** in AWS Cloud9 and change directory. If you don't see a bash terminal open, select **Window > New Terminal** from the Cloud9 toolbar.

```
cd ~/environment/workshop/labs/kb_rag_chatbot
```

Just want to run the app?

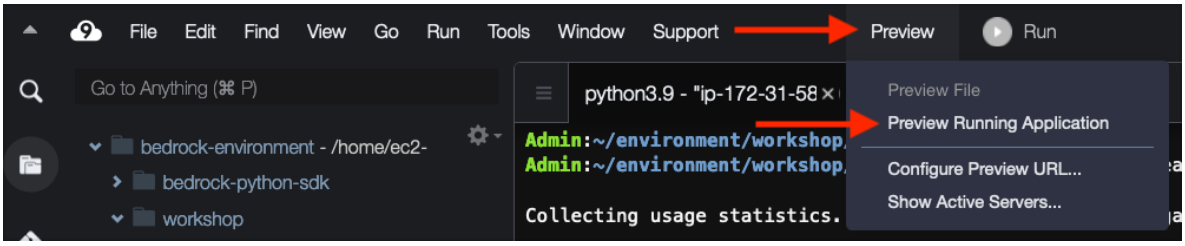
Expand here & run this command instead

2. Run the streamlit command from the terminal.

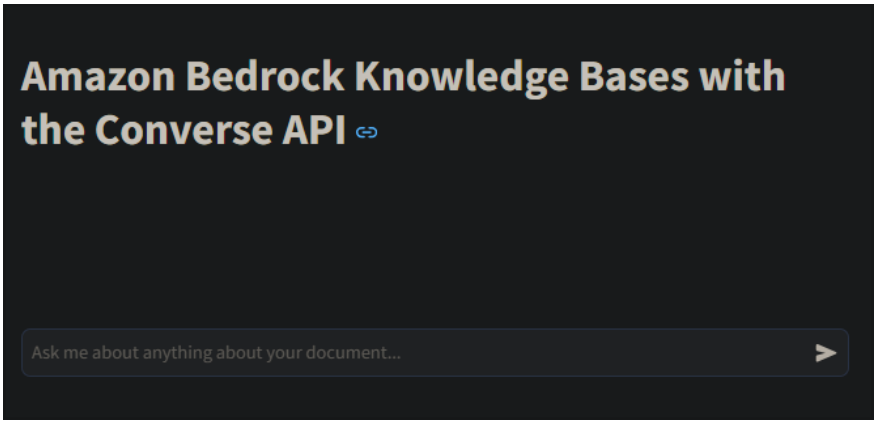
```
streamlit run kb_rag_chatbot_app.py --server.port 8080
```

Ignore the Network URL and External URL links displayed by the Streamlit command. Instead, we will use AWS Cloud9's preview feature.

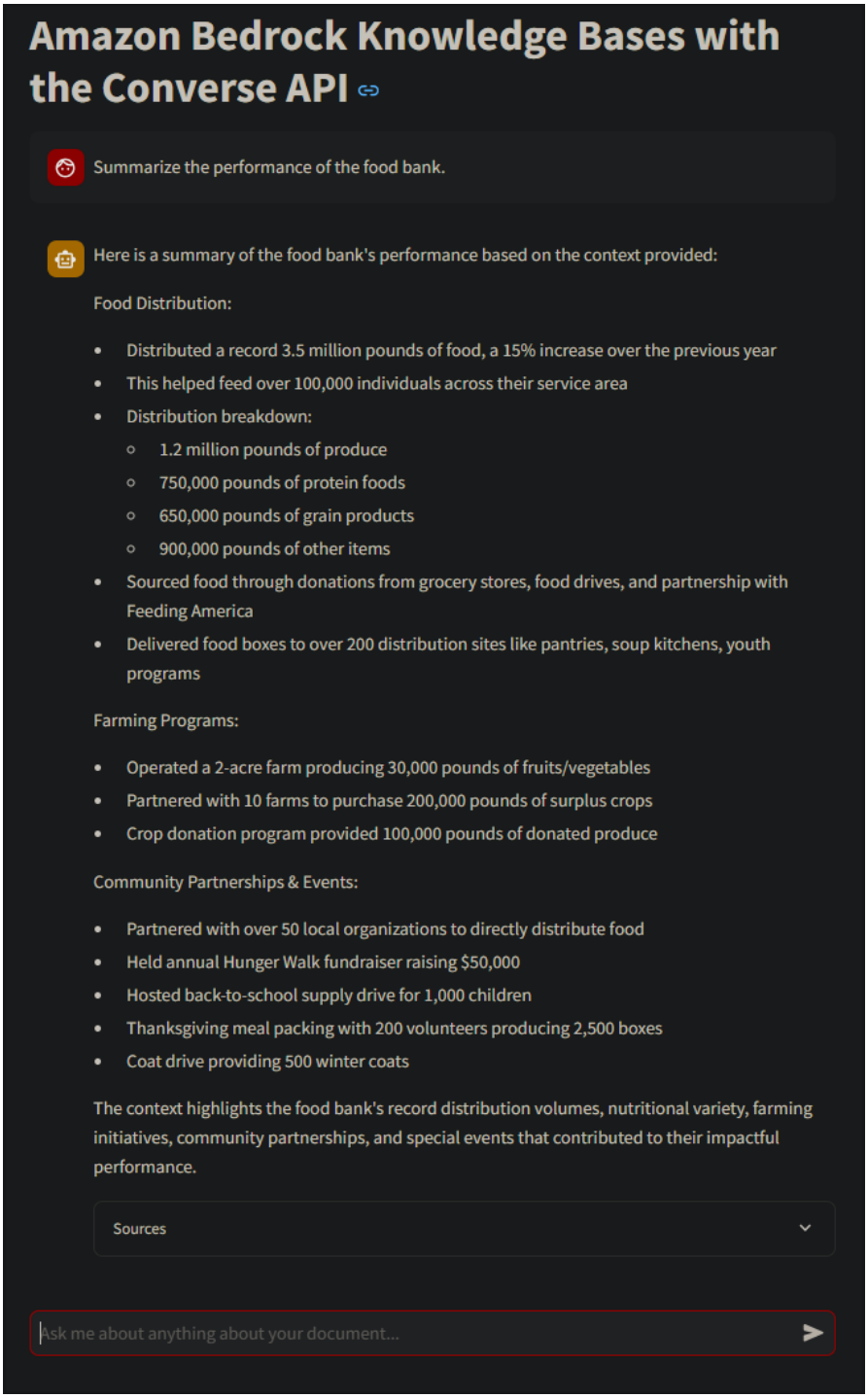
3. In AWS Cloud9, select **Preview** -> **Preview Running Application**.



You should see a web page like below:



4. Try out some prompts and see the results:
    - Summarize the performance of the food bank.
    - What organizations did we partner with to create impact?



Multilingual Queries

The Amazon Titan Embeddings model supports more than 25 languages, meaning that as your documents are embedded into a vector store they can be retrieved using multilingual prompts. Even though your source documents might be in one language, the model will translate source information to match the prompt language. Try this prompt in various languages.

- What is our strategy for volunteering?
- ¿Cuál es nuestra estrategia para el voluntariado?
- 我们的志愿服务策略是什么?

Natural Boundary

Using Retrieval-Augmented Generation (RAG) can help enforce a natural boundary of information by restricting the model to your supplied documents for generating answers. Try this prompt below to see how the model handles questions that are not covered in the supplied documents:

- What is the capital of Nebraska?

5. Close the preview tab in AWS Cloud9. Return to the terminal and press Control-C to exit the application.

Congratulations!

You have successfully built a question & answer app with Bedrock, Knowledge Bases and Streamlit!

Previous

Next

Full-code

Overview

1. Working with vector stores
    - Full-code: Amazon Opensearch Serverless

Previous

Next

- **Amazon OpenSearch Serverless**

Amazon OpenSearch Serverless

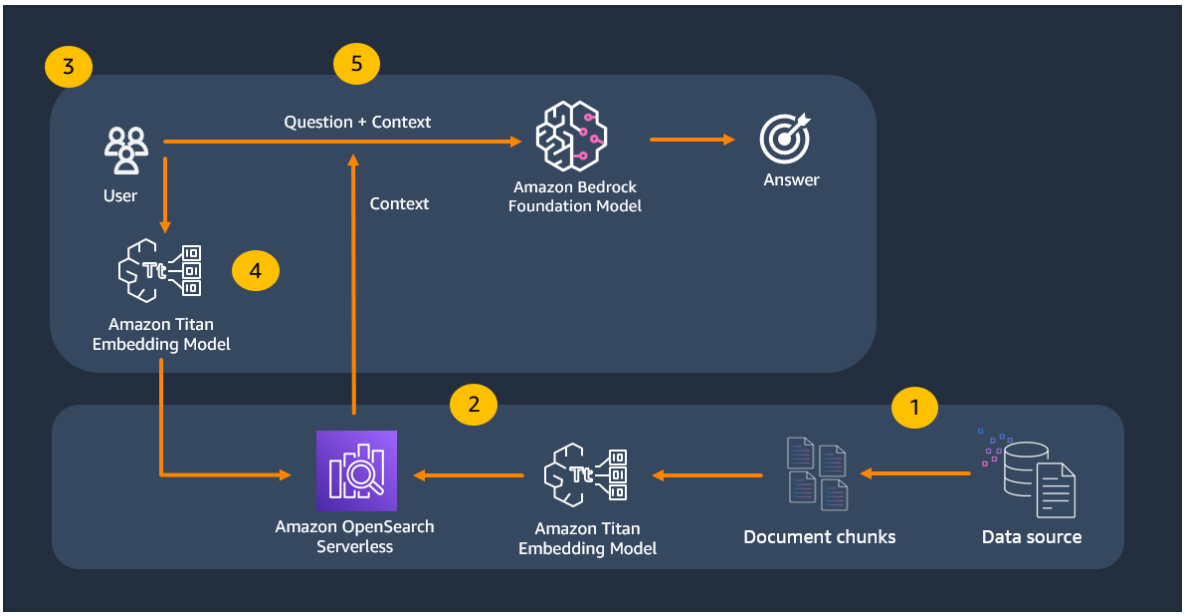
In this lab, we will build a simple chatbot with Amazon Bedrock, Amazon OpenSearch Serverless and Streamlit. We will use the Amazon Bedrock Converse API

which simplifies development of chat experiences using models invoked through Bedrock.

Amazon Bedrock (and LLMs in general) don't have any concept of state or memory. Any chat history has to be tracked externally and then passed into the model with each new message. The Converse API supports conversational

history in a structured way as part of the API request, reducing the complexity for multi-turn conversations.
We also want to supplement the model's underlying data with external knowledge through **Retrieval-Augmented Generation** (RAG). We'll use Amazon OpenSearch serverless as our vector database for semantic search retrieval of relevant context.

Architecture



1. Source data is split into chunks to support ingestion into our vector store.
2. The Titan Embeddings model creates numerical representations of the chunks and stores them into a vector index in an Amazon OpenSearch Serverless collection.
3. The user submits a question.
4. The user's question is converted to a vector representation for semantic search against the OpenSearch serverless collection. Relevant chunks of data are returned from the search.
5. The combined content from the matching context + the original question are then passed to the large language model to get the best answer.

Previous
Next

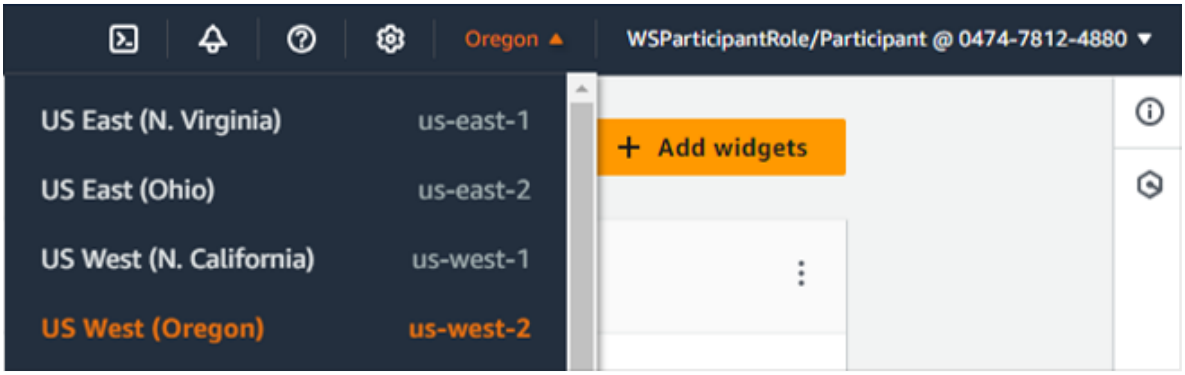
- Amazon OpenSearch Serverless

- **Ingest data into an Amazon OpenSearch Serverless collection**

Ingest data into an Amazon OpenSearch Serverless collection

Important - Region Selection

Ensure that you're in the us-west-2 region (Oregon) when setting up your OpenSearch Serverless collection.



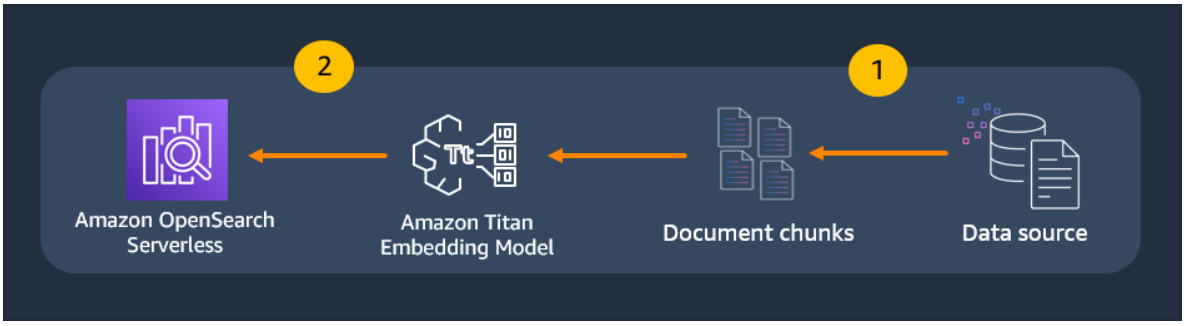
Please complete the Launching Cloud9 section before starting this lab.

Introduction

An OpenSearch Serverless collection has been created on your behalf. The collection uses the vector engine for Amazon OpenSearch serverless

, which offers high-performing vector storage and search. The collection requires an index to be created before documents can be ingested into the vector store. In this section, you will run a python script that uses the AWS SDK to create a vector index in the OpenSearch Serverless collection, then ingests data from a food bank's strategic plan PDF.

Architecture



1. Source data is split into chunks to support ingestion into our vector

store.
2. The Titan Embeddings model creates numerical representations of the chunks and stores them into a vector index in an Amazon OpenSearch Serverless collection.

Create the data ingestion script

We will be using AWS Cloud9 as our integrated development environment for this lab section. If you have not already setup Cloud9 in a previous section, please refer to the Launch AWS Cloud9 lab.

1. Launch the Cloud9 environment by following the steps in the Launching Cloud9 section.
2. Navigate to the **workshop/labs/oss_rag_chatbot** folder, and open the file **oss_load_docs.py**
3. Add the import statements.
    - These statements import langchain libraries to help with document chunking before ingesting into our index.

```
import boto3
import json
import os
import time
from opensearchpy import OpenSearch, RequestsHttpConnection,
AWSV4SignerAuth

from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

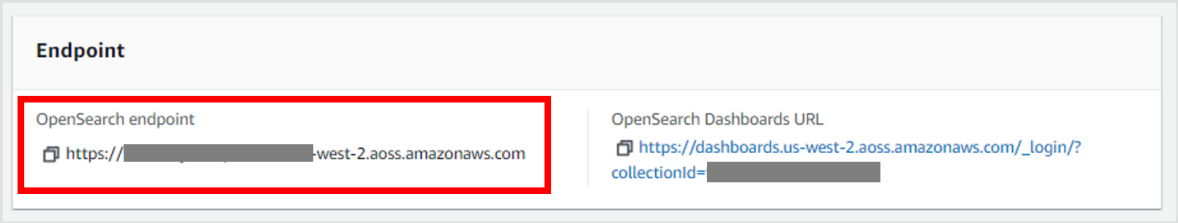
4. Create the Bedrock and OpenSearch clients for the AWS SDK.

```
# instantiating the bedrock client, with specific CLI profile

bedrock = boto3.client('bedrock-runtime') # Creates a Bedrock client

opensearch = boto3.client("opensearchserverless")
```

5. Create the client connection to our existing OpenSearch Serverless collection. We will need the collection endpoint.

To find your collection endpoint, navigate to the your Amazon OpenSearch Serverless collection

and copy the OpenSearch endpoint.



- Paste your OpenSearch collection endpoint in the **host** variable. Do NOT include the https:// at the beginning of the endpoint.

```
# Instantiating the OpenSearch client, with specific CLI profile

# make sure you DO NOT include the https:// at the front of the endpoint

host = '<insert your collection endpoint here>' # Add your collection endpoint, for example: upw75ytw6462.us-west-2.aoss.amazonaws.com
region = 'us-west-2'
service = 'aoss'
credentials = boto3.Session().get_credentials()
auth = AWSV4SignerAuth(credentials, region, service)

client = OpenSearch(
hosts=[{'host': host, 'port': 443}],
http_auth=auth,
use_ssl=True,
verify_certs=True,
timeout=30,
connection_class=RequestsHttpConnection,
pool_maxsize=20
)
```

6. Add a function that splits our document into chunks prior to ingestion. Use the langchain helper libraries to support the text splitting.

```
def chunk_documents(pdf_path):

    loader = PyPDFLoader(file_path=pdf_path) #load the pdf file

    documents = loader.load()
```

```
    print("Loading document(s) for chunking...")

    text_splitter = RecursiveCharacterTextSplitter( #create a text splitter
        separators=["\n\n", "\n", ".", " "], #split chunks at (1) paragraph, (2) line, (3)
sentence, or (4) word, in that order
        chunk_size=1000, #divide into 1000-character chunks using the
separators above
        chunk_overlap=100 #number of characters that can overlap with previous
chunk
    )

    docs = text_splitter.split_documents(documents) # split our document into
chunks for targeted retreival later on.

    print(f'After chunking we have {len(docs)} documents.')

    return docs
```

7. Add a function that creates a new vector index where we will ingest our chunked documents.
    - When we define the body of the index we must specify the data types of the fields. We need to include a vector field to store the embeddings, which are numeric representations of our document chunks.

```
# Function to create an index

def create_index(index_name):

    # Define index mapping
    # Include a vector field to store vector embeddings
    index_body = {
        "settings": {
            "index.knn": True,
        },
        "mappings": {
            "properties": {
                "vectors": {
                    "type": "knn_vector",
                    "dimension": 1024,
                    "method": {
                        "engine": "faiss",
                        "space_type": "l2",
                        "name": "hnsw",
                    }
                },
```

```
            "text": {"type": "text"}
          }
        }
    }

    response = client.indices.create(index=index_name, body=index_body,
ignore=400)
    print(f"Create index response: {json.dumps(response, indent=2)}")
    print(f"Index '{index_name}' created.")
```

8. Add a function that performs the embeddings of text content, using the Amazon Titan embeddings model to create numeric representations of text.

```
def get_embedding(body):

    modelId = 'amazon.titan-embed-text-v2:0'
    accept = 'application/json'
    contentType = 'application/json'
    response = bedrock.invoke_model(body=body, modelId=modelId,
accept=accept, contentType=contentType)
    response_body = json.loads(response.get('body').read())
    embedding = response_body.get('embedding')
    return embedding
```

9. Add a function that adds each chunked document to the recently created vector index in our OpenSearch collection. This will populate our index with data from our local PDF. Note that we create a metadata field to capture the source of the document we are indexing.

```
def indexDoc(doc_num, client, index_name, vectors, text, metadata):

    indexDocument = {
        'vectors': vectors,
        'text': text,
        'metadata': {
            'source' : metadata["source"]
        }
    }

    print(f'Indexing document: {doc_num} ...')

    # Configuring the specific index
    response = client.index(
```

```
        index=index_name,
        body=indexDocument,
        refresh=False
    )

    return response
```

10. Define a main function that calls each helper function in order.
    - Chunks the documents using the langchain helper libraries
    - Creates a new vector index in our OpenSearch Serverless collection
    - For each document chunk, create an embedding of the text and ingest into the vector index.

```
def main():

    # Chunk your PDF document to optimize vector storage and improve detail retrieval
    pdf_path = "~/environment/workshop/completed/rag_chatbot/output.pdf"
    documents = chunk_documents(pdf_path)

    # Create a new vector index in your OpenSearch collection
    index_name = 'vector-index'
    create_index(index_name)

    # Delay for a few seconds to allow new index to fully provision
    time.sleep(5)

    # Iterate through each of your chunked documents, generate embeddings for each, and add them to your vector index.
    for doc_num, doc in enumerate(documents, start=1):
        # The text and source data of each chunk
        exampleContent = doc.page_content
        exampleMetadata = doc.metadata
        # Generating the embeddings for each chunk of text data
        exampleInput = json.dumps({"inputText": exampleContent})
        exampleVectors = get_embedding(exampleInput)
        # setting the text data as the text variable, and generated vector to a vector variable
        text = exampleContent
        vectors = exampleVectors
        # calling the indexDoc function, passing in the OpenSearch Client, the created vector, and corresponding text data
        indexDoc(doc_num, client, index_name, vectors, text, exampleMetadata)

# Run your main function
```

```
main()
```

11. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).
Magnificent! Now you are ready to run ingestion script!

Run the ingestion script

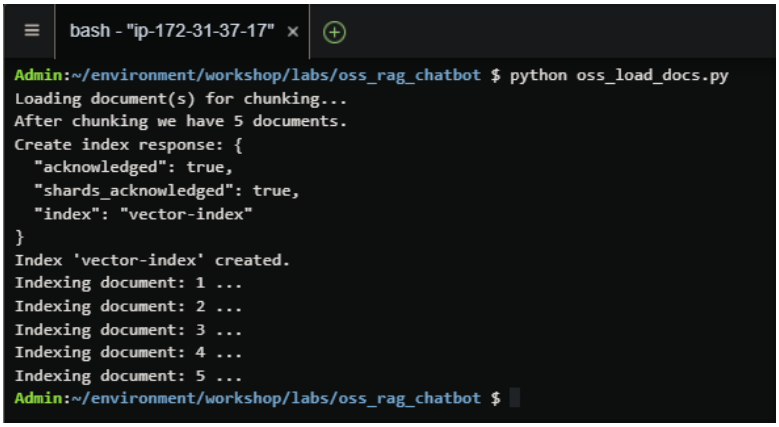
1. Select the **bash terminal** in AWS Cloud9 and change directory. If you don't see a bash terminal open, select **Window > New Terminal** from the Cloud9 toolbar.

```
cd ~/environment/workshop/labs/oss_rag_chatbot
```

2. Run the streamlit command from the terminal.

```
python oss_load_docs.py
```

From the Bash terminal you will see printed output as the documents are chunked and ingested. Your output should look similar to the below.



Previous
Next

- Amazon OpenSearch Serverless

- RAG Chatbot with Amazon OpenSearch Serverless

RAG Chatbot with Amazon OpenSearch Serverless

Please complete the Running at an AWS-facilitated event steps before starting this lab.

Please complete the Launching Cloud9 section before starting this lab.

Lab introduction

In this lab, we will build a simple chatbot with Amazon Bedrock, Amazon OpenSearch Serverless and Streamlit. We will use the Amazon Bedrock Converse API

which simplifies development of chat experiences using models invoked through Bedrock.

Amazon Bedrock (and LLMs in general) don't have any concept of state or memory. Any chat history has to be tracked externally and then passed into the model with each new message. The Converse API supports conversational history in a structured way as part of the API request, reducing the complexity for multi-turn conversations.

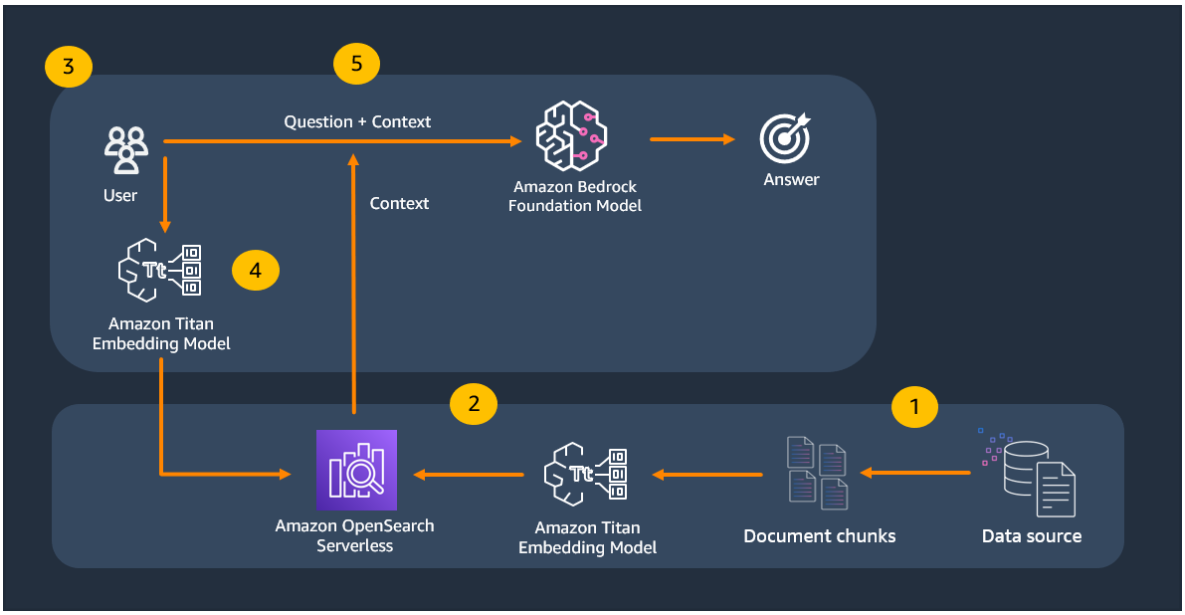
We also want to supplement the model's underlying data with external knowledge through Retrieval-Augmented Generation (RAG). We'll use Amazon OpenSearch serverless as our vector database for semantic search retrieval of relevant context.

You can build the application code by copying the code snippets below and pasting into the indicated Python file.

Just want to run the app?

You can jump ahead to run a pre-made application.

Architecture



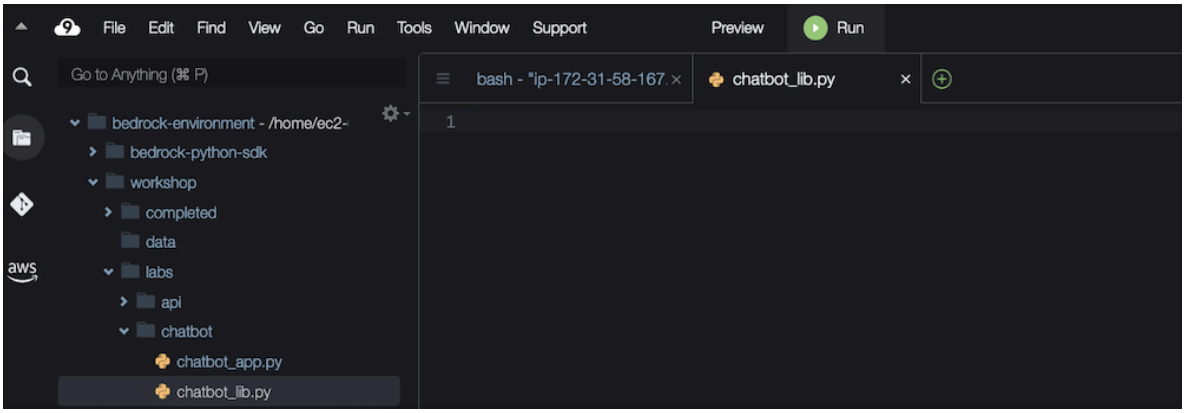
1. Source data is split into chunks to support ingestion into our vector store.
2. The Titan Embeddings model creates numerical representations of the chunks and stores them into a vector index in an Amazon OpenSearch Serverless collection.
3. The user submits a question.
4. The user's question is converted to a vector representation for semantic search against the OpenSearch serverless collection. Relevant chunks of data are returned from the search.
5. The combined content from the matching context + the original question are then passed to the large language model to get the best answer.

This application consists of two files: one for the Streamlit front end, and one for the supporting library to make calls to Bedrock.

Create the library script

First we will create the supporting library to connect the Streamlit front end to the Bedrock back end.

1. Navigate to the **workshop/labs/oss_rag_chatbot** folder, and open the file **oss_rag_chatbot_lib.py**



2. Add the import statements.

```
import boto3
import json
import os
from opensearchpy import OpenSearch, RequestsHttpConnection,
AWSV4SignerAuth
```

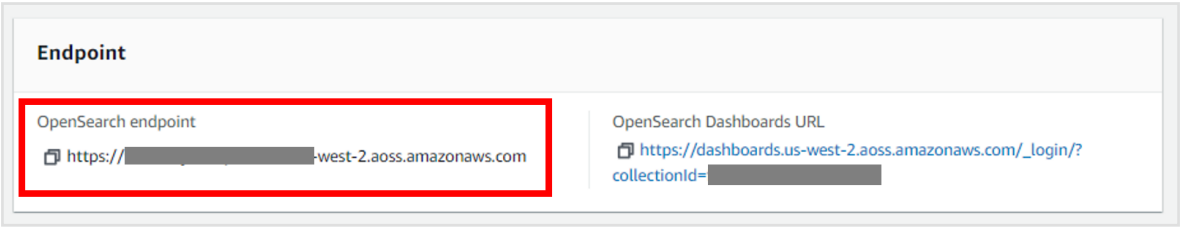
3. Create the Bedrock and OpenSearch clients for the AWS SDK.

```
# create the bedrock client, with specific CLI profile

bedrock = boto3.client('bedrock-runtime') # Creates a Bedrock client

opensearch = boto3.client("opensearchserverless")
```

4. Create the client connection to our existing OpenSearch Serverless collection. We will need the collection endpoint.

To find your collection endpoint, navigate to the your Amazon OpenSearch Serverless collection

and copy the OpenSearch endpoint.



- Paste your OpenSearch collection endpoint in the **host** variable. Do NOT include the https:// at the beginning of the endpoint.

```
# Instantiating the OpenSearch client, with specific CLI profile

# make sure you DO NOT include the https:// at the front of the endpoint

host = '<insert your collection endpoint here>' # Add your collection endpoint, for example: upw75ytw6462.us-west-2.aoss.amazonaws.com
region = 'us-west-2'
service = 'aoss'
credentials = boto3.Session().get_credentials()
auth = AWSV4SignerAuth(credentials, region, service)

client = OpenSearch(
hosts=[{'host': host, 'port': 443}],
http_auth=auth,
use_ssl=True,
verify_certs=True,
timeout=30,
connection_class=RequestsHttpConnection,
pool_maxsize=20
)
```

5. Add a function that performs the embeddings of text content, using the Titan embeddings model to create numeric representations of text. This will be used to embed the user's query from the front end so we can run a semantic search across our embeddings in the vector index.

```
def get_embedding(body):

    # defining the embeddings model
    modelId = 'amazon.titan-embed-text-v2:0'
    accept = 'application/json'
    contentType = 'application/json'
    # invoking the embedding model
    response = bedrock.invoke_model(body=body, modelId=modelId, accept=accept, contentType=contentType)
    # reading in the specific embedding
```

```
    response_body = json.loads(response.get('body').read())
    embedding = response_body.get('embedding')
    return embedding
```

6. Add this function to perform a semantic search across the vector index.
    - Run the user's query through the embedding function
    - Perform a semantic search on the vector index to return relevant context
    - Iterate through the context and return the response

```
def search_vector_db(query):

    # formatting the user input
    userQueryBody = json.dumps({"inputText": query})
    # creating an embedding of the user input to perform a KNN (K-nearest-neighbors) search
    userVectors = get_embedding(userQueryBody)
    # the query parameters for the KNN search performed by Amazon OpenSearch with the generated User Vector passed in.
    query = {
        "size": 3,
        "query": {
            "knn": {
                "vectors": {
                    "vector": userVectors, "k": 3
                }
            }
        },
        "_source": True,
        "fields": ["text"],
    }
    # performing the search on OpenSearch passing in the query parameters constructed above
    response = client.search(
        body=query,
        index="vector-index" # vector index that was created in the previous section
    )

    # Format Json responses into text
    similaritysearchResponse = ""
    # iterating through all the findings of Amazon OpenSearch query and adding them to a single string to pass in as context
    for i in response["hits"]["hits"]:
        outputtext = i["fields"]["text"]
```

```
        similaritysearchResponse = similaritysearchResponse + "Info = " +
str(outputtext)
        similaritysearchResponse = similaritysearchResponse

    # Extract the sources associated with the context chunks from the json
response
    sources = list(set(item["_source"]["metadata"]["source"] for item in
response["hits"]["hits"]))

    return similaritysearchResponse, sources
```

7. Create a memory list to store conversation history.

```
# Instantiate a messages list to store the conversation history. This will
preserve context along with the latest message.

messages = []
```

8. Add this function to call bedrock, passing in context from the vector search.
    - We're creating a function we can call from the Streamlit front end application.
    - This function will call the search_vector_db function to retrieve relevant context from our OpenSearch serverless collection. It injects the results into our user prompt.
    - The function then defines model parameters like temperature and top_k, then submits the user prompt to the Claude 3 Sonnet model using the Converse API. The response is returned to the front end.

```
def invoke_model(query):

    # run a similarity search on the Amazon OpenSearch Serverless Collection to
return context and sources relevant to the user's query
    results, sources = search_vector_db(query)

    prompt_data = f"""\n\nHuman:
    Answer the following question to the best of your ability based on the context
provided.
    Do not include information that is not relevant to the question.
    Only provide information based on the context provided, and do not make
assumptions.

    ###

    Question: {query}
```

```
    Context: {results}

    ###

    \n\nAssistant:

    """

     # Define the system prompts to guide the model's behavior and role.
    system_prompts = [{"text": "You are a helpful assistant. Keep your answers short and succinct."}]

    # Format the user's message as a dictionary with role and content
    message = {"role": "user", "content": [{"text": prompt_data}]}

    # Append the formatted user message to the list of messages.
    messages.append(message)

    # Set the temperature for the model inference, controlling the randomness of the responses.
    temperature = 0.5

    # Set the top_k parameter for the model inference, determining how many of the top predictions to consider.
    top_k = 200

    # Call the converse method of the Bedrock client object to get a response from the model.
    response = bedrock.converse(
        modelId="anthropic.claude-3-sonnet-20240229-v1:0",
        messages=messages,
        system=system_prompts,
        inferenceConfig={"temperature": temperature},
        additionalModelRequestFields= {"top_k": top_k}
    )

    # # Extract the output message from the response.
    output_message = response['output']['message']

    # # Append the output message to the list of messages.
    messages.append(output_message)

    # Print the message output to review how the conversation history builds after each turn
    print("Message History: ", messages)
    print("##############################")
```

```
    # Return the text of the model's response, along with the sources, to the
frontend
    return output_message['content'][0]['text'], sources
```

9. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Nice! You are done with the backing library. Now we will create the front-end application.

Create the Streamlit front-end app

1. In the same folder as your lib file, open the file **chatbot_app.py**

2. Import the streamlit library and import the invoke_model function from our library script.
    - These statements allow us to use Streamlit elements and call functions in the backing library script.

```
import streamlit as st #all streamlit commands will be available through the "st"
alias
from oss_rag_chatbot_lib import invoke_model
```

3. Add the page title and configuration.
    - Here we are setting the page title on the actual page and the title shown in the browser tab.

```
st.set_page_config(page_title="RAG Chatbot") #HTML title
st.title("Amazon OpenSearch Serverless with the Converse API")
```

4. Add the message history to the session cache.
    - This allows us to maintain a unique chat memory per user session. Otherwise, the chatbot won't be able to remember past messages to display on the front end UI.
    - In Streamlit, session state is tracked server-side. If the browser tab is closed, or the application is stopped, the session and its chat history will be lost. In a real-world application, you would want to track the chat history in a database like Amazon DynamoDB

```
# configuring values for session state

if "messages" not in st.session_state:
```

```
    st.session_state.messages = []
```

5. Add the for loop to render previous chat messages in the front end UI.
    - Re-render previous messages based on the "messages" session state object.

```
# Streamlit is stateless, so we need to re-render our historical messages and sources from our preserved session state

for message in st.session_state.messages:

    with st.chat_message(message["role"]):

        st.markdown(message["content"])

        if message["role"] == "assistant":

            with st.expander("Sources"):
                st.write(message["sources"])
```

6. Add the input elements.
    - We use the if block below to handle the user input. See the in-line comments below for more details.

```
# evaluating st.chat_input and determining if a question has been input

if question := st.chat_input("Ask me about anything about your document..."):

    # with the user icon, write the question to the front end
    with st.chat_message("user"):
        st.markdown(question)

    # append the question and the role (user) as a message to the session state
    st.session_state.messages.append({"role": "user", "content": question})

    # respond as the assistant with the answer
    with st.chat_message("assistant"):

        # making sure there are no messages present when generating the answer
        message_placeholder = st.empty()

        # passing the user question into invoke_model function from our library script, then return the response and sources
```

```
        answer, sources = invoke_model(question)

        # writing the answer to the front end
        message_placeholder.markdown(f"{answer}")

        with st.expander("Sources"):
           st.write(sources)

    # Append the assistant's response and sources to the session state
    # When streamlit re-renders, it will use this stored history to display past
messages to the UI
    st.session_state.messages.append({"role": "assistant", "content": answer,
"sources": sources})
```

7. Save the file (Ctrl-S, or File > Save from Cloud9 toolbar).

Magnificent! Now you are ready to run the application!

Run the Streamlit app

1. Select the **bash terminal** in AWS Cloud9 and change directory. If you don't see a bash terminal open, select **Window > New Terminal** from the Cloud9 toolbar.

```
cd ~/environment/workshop/labs/oss_rag_chatbot
```

Just want to run the app?

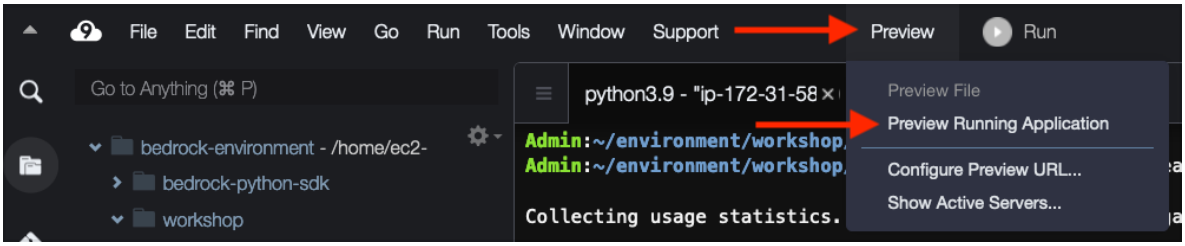
Expand here & run this command instead

2. Run the streamlit command from the terminal.

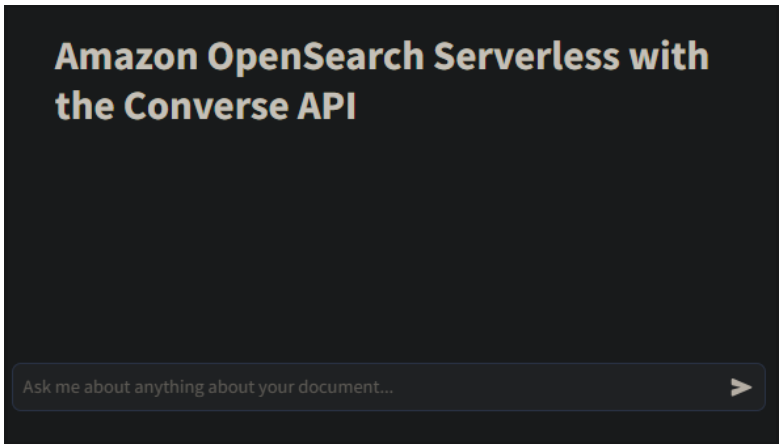
```
streamlit run oss_rag_chatbot_app.py --server.port 8080
```

Ignore the Network URL and External URL links displayed by the Streamlit command. Instead, we will use AWS Cloud9's preview feature.

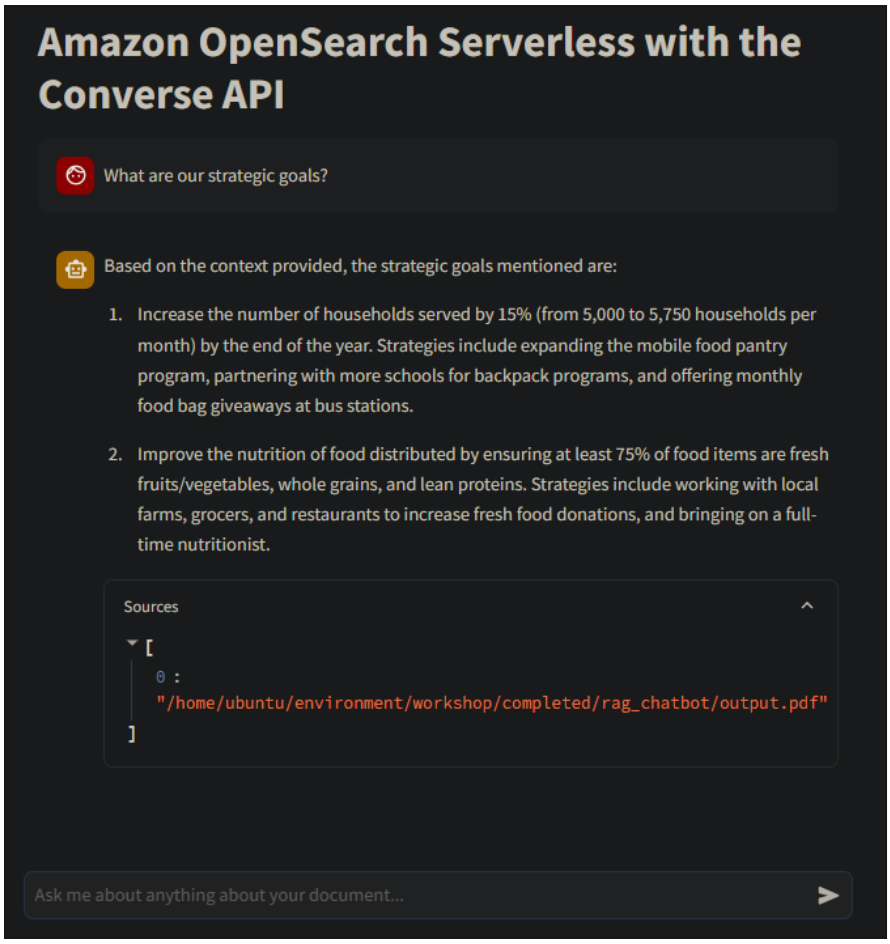
3. In AWS Cloud9, select **Preview** -> **Preview Running Application**.



You should see a web page like below:



4. Try out some prompts and see the results.
    - What are our strategic goals?
    - What is our expansion plan for the future?



5. Close the preview tab in AWS Cloud9. Return to the terminal and press Control-C to exit the application.

Multilingual Queries

The Amazon Titan Embeddings model supports more than 25 languages, meaning that as your documents are embedded into a vector store they can be retrieved using multilingual prompts. Even though your source documents might be in one language, the model will translate source information to match the prompt language. Try this prompt in various languages.

- What is our strategy for volunteering?
- ¿Cuál es nuestra estrategia para el voluntariado?
- 我们的志愿服务策略是什么?

Natural Boundary

Using Retrieval-Augmented Generation (RAG) can help enforce a natural boundary of information by restricting the model to your supplied documents for generating answers. Try this prompt below to see how the model handles questions that are not

covered in the supplied documents:

- What is the capital of Nebraska?

6. Close the preview tab in AWS Cloud9. Return to the terminal and press Control-C to exit the application.

Congratulations!
You have successfully built a question & answer app with Bedrock, OpenSearch Serverless and Streamlit!
Previous
Next

Hosting a Front End for GenAI

This section includes options for deploying your GenAI chatbot to external applications using Amazon Lex.

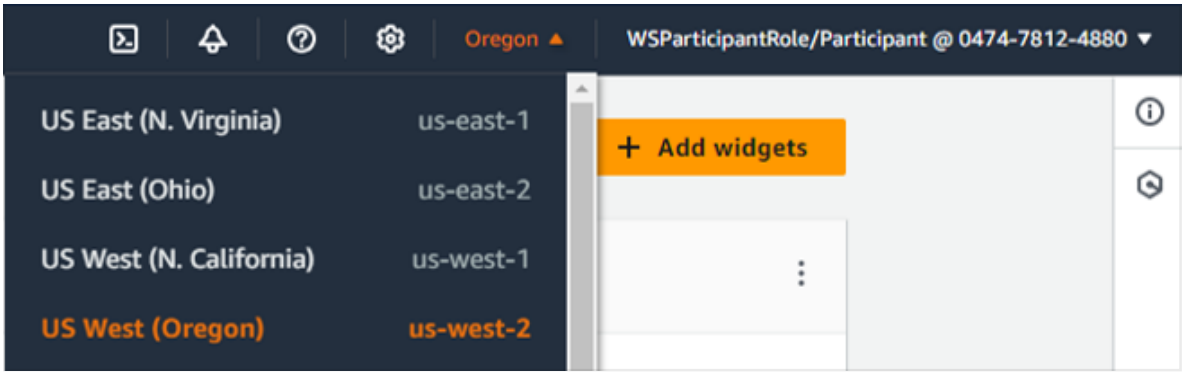
Overview

1. Hosting a front end for GenAI
    - No-code: Amazon Lex with built-in QnAIntent
    - Full-code: Amazon Lex with AWS Lambda and Amazon bedrock

Previous
Next

Create an Amazon Lex Bot

Important - Region Selection
Ensure that you're in the us-west-2 region (Oregon) when setting up your Lex chatbot.

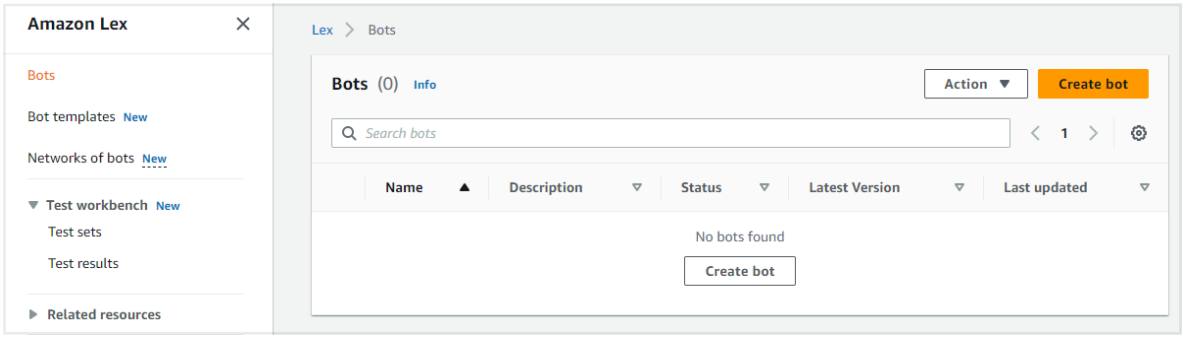


Creating an Amazon Lex chatbot

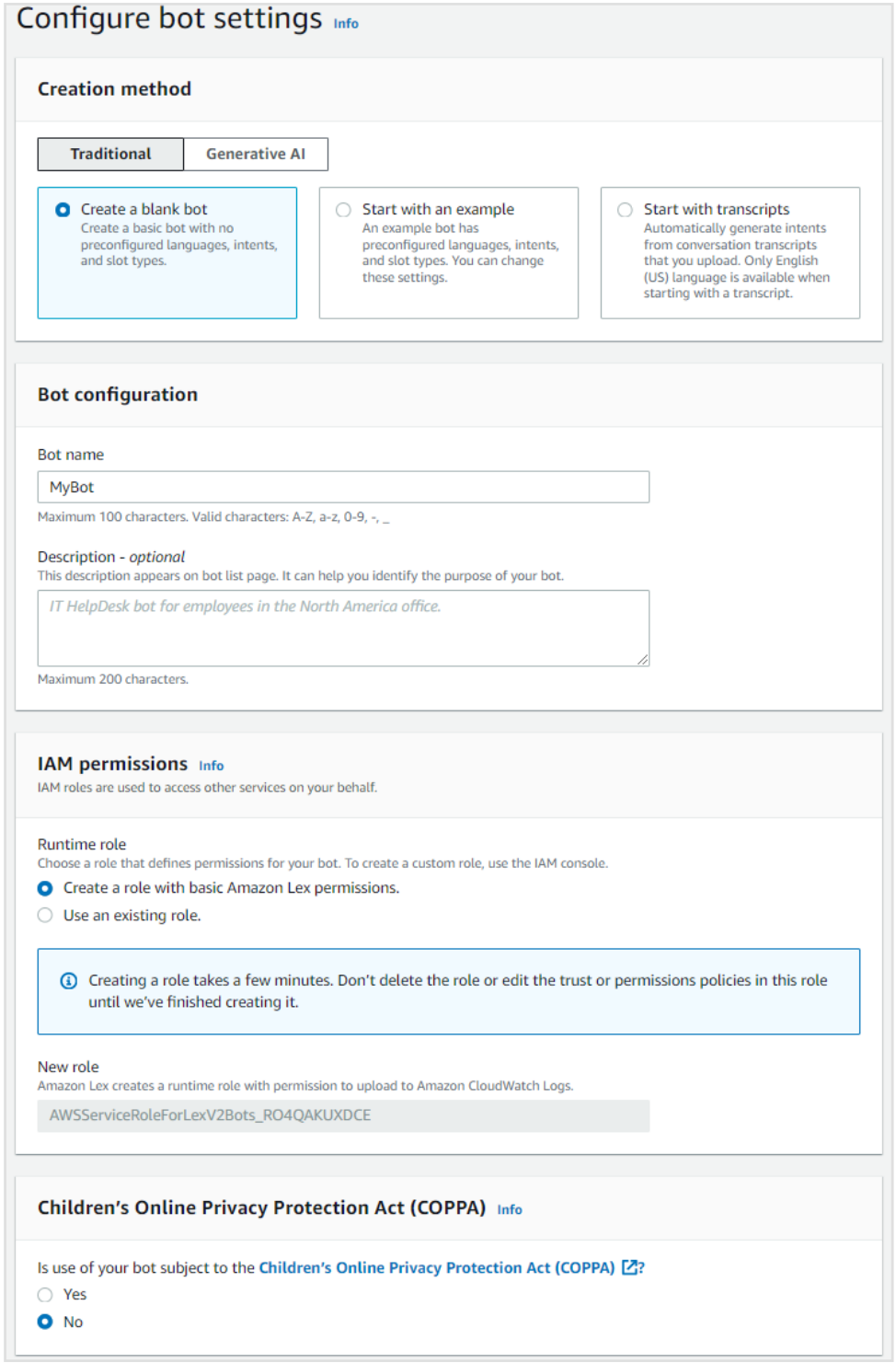
You can use Amazon Lex to create a chatbot interface that's deployable to your existing applications. In this section we will create the basic foundation for a bot, and in the no-code and full code labs you will integrate your RAG components for a complete GenAI application.

1. Open the AWS Management Console for Amazon Lex

2. From the **Bots** menu, select **Create Bot**



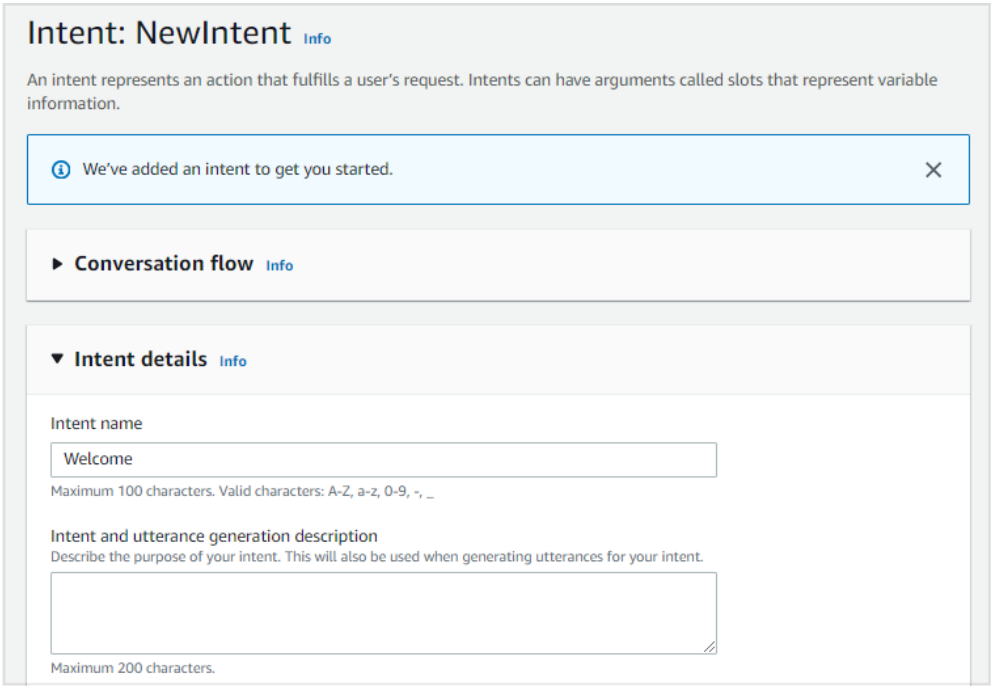
3. Select the **Traditional** creation method.
4. Select **Create a blank bot** and choose a simple bot name (exp. MyBot).
5. Under **IAM Permissions** choose **Create a role with basic Amazon Lex permissions**.
6. Under the COPPA section, choose **No**.
7. Select **Next**



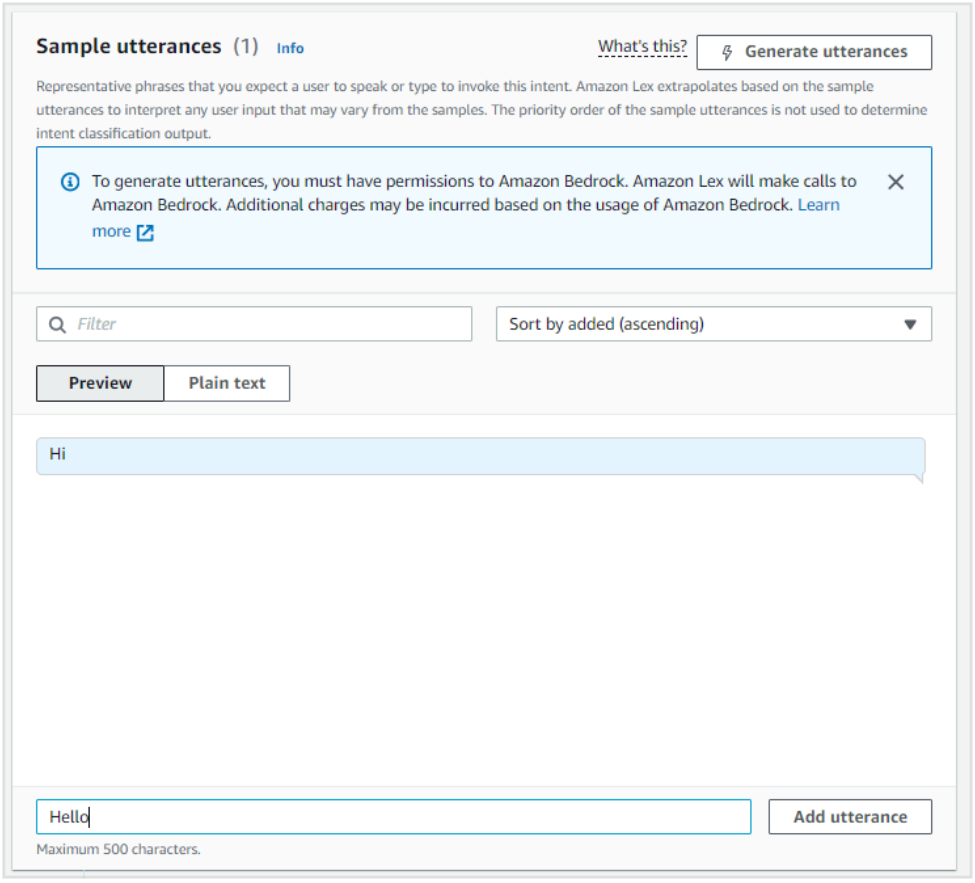
8. On the **Add language to bot** page, leave the options as the default and choose **Done**.

NOTE: You may see error messages commenting on missing permissions for Voice Interaction. You can safely ignore these. Text-to-speech is out of scope for this workshop.

9. The bot will create a placeholder intent called **NewIntent** on your behalf. From the **NewIntent** page, update the **Intent name** to **Welcome**.



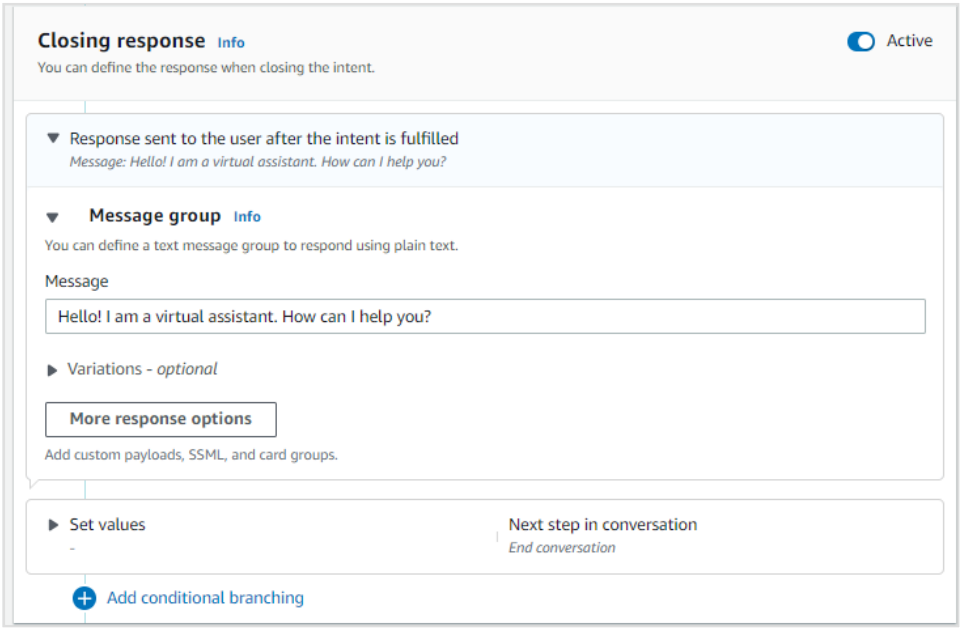
10. In the **Sample utterances** section, add two sample utterances that will trigger this intent. In this example, we will add "Hi" and "Hello" as utterances. Add the utterance into the input at the bottom of the section and choose **Add utterance**.



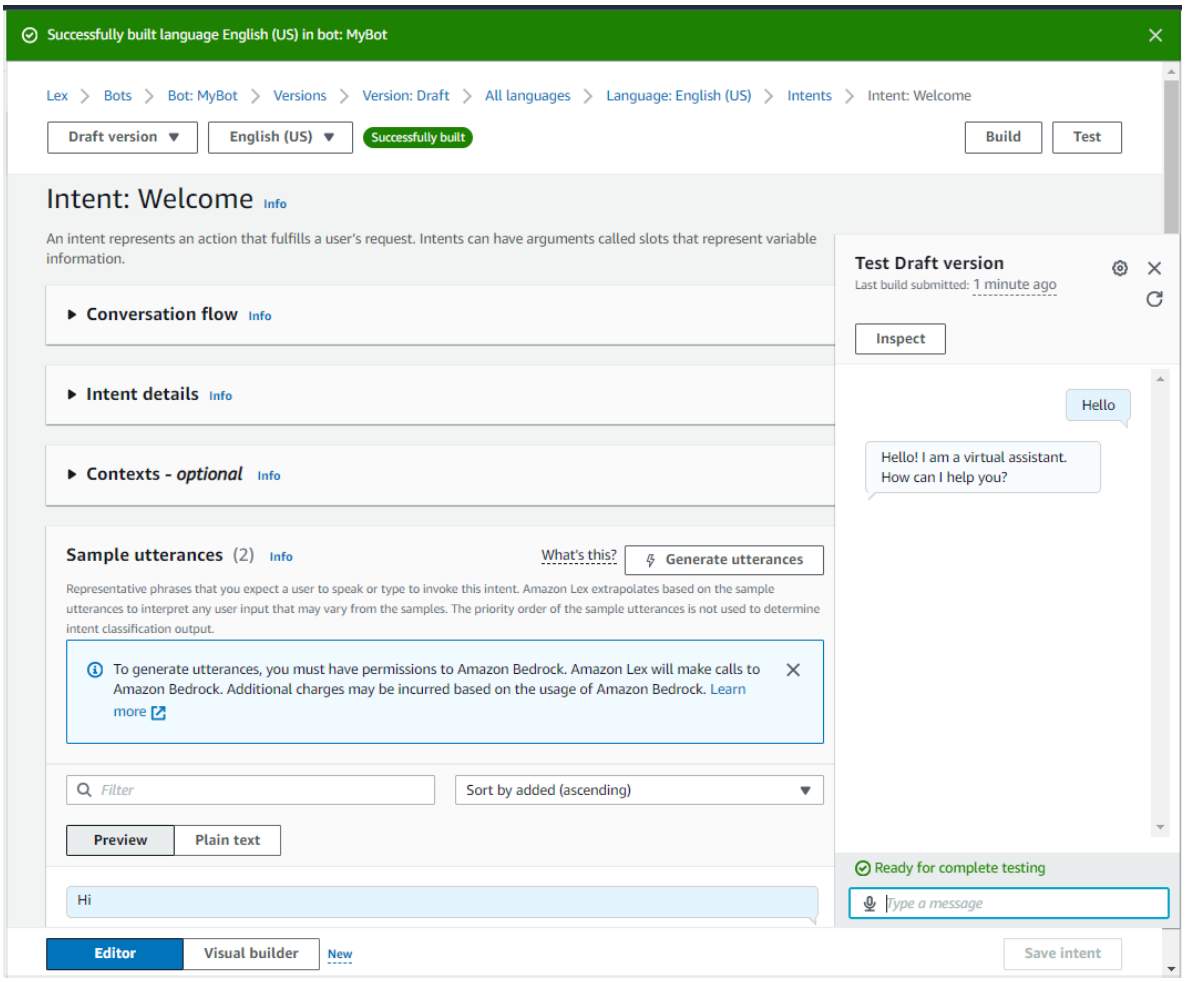
11. Scroll down to the **Closing response** section and expand the **Response sent to the user after the intent is fulfilled** block.

Under **Message group**, add the message below to the **Message** input. This message will be displayed when the intent is triggered by the utterances we added previously.

- Hello! I am a virtual assistant. How can I help you?



12. Choose **Save intent**.
13. Select **Build** from the top of the page. Once the build completes, select **Test** to experiment with your bot. Enter "Hi" or "Hello" into the test chatbot to trigger your closing response.

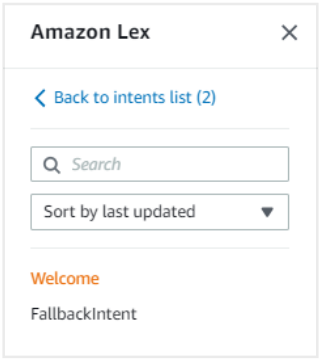


Congratulations!
You have successfully created a basic bot. Next, we will add Generative AI capabilities.

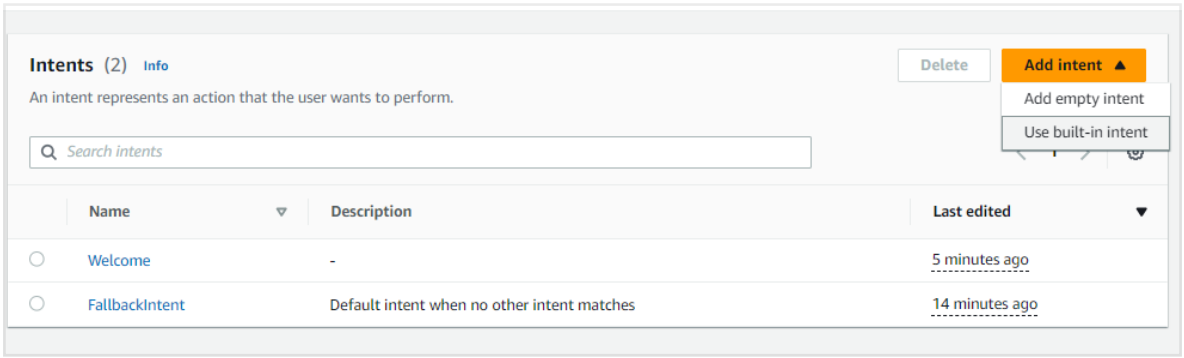
Using the Built-in QnaIntent

Adding the built-in QnAIntent for RAG

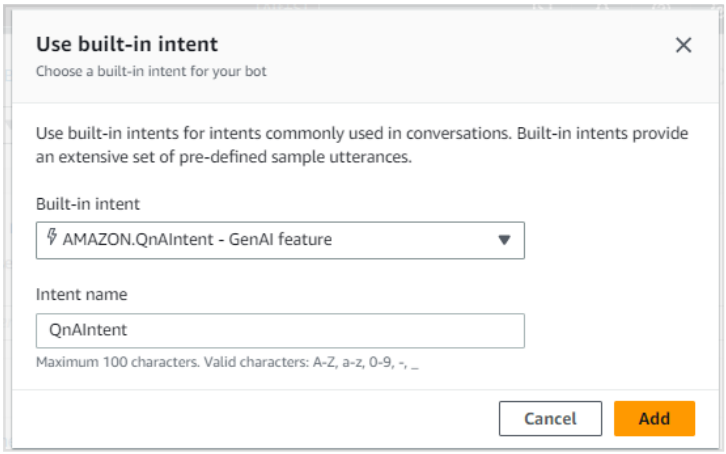
1. From the current intent page, navigate back to your list of intents by selecting **Back to intents list (2)** from the side menu.



2. Select **Add intent** and choose **Use built-in intent** from the Intents list.



3. Select the **AMAZON.QnAIntent - GenAI feature** from the list of built-in intents. Add a simple name for the intent (exp. QnAIntent).



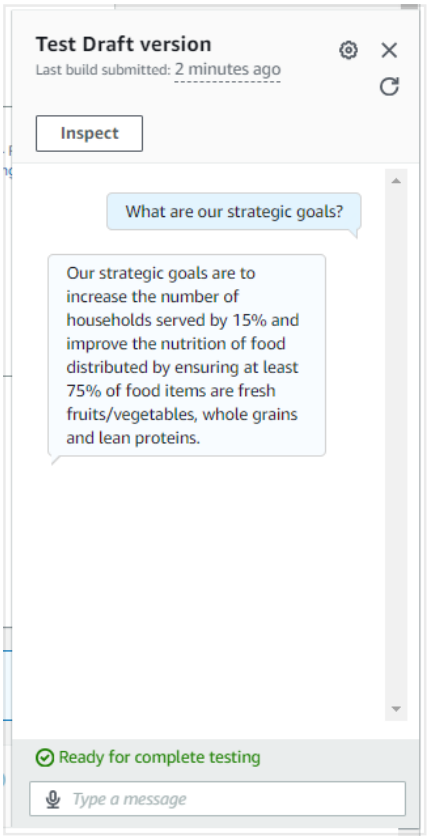
You'll notice that there are three knowledge store options for Lex to choose from. As of **November 2024**, the QnAIntent GenAI feature does not integrate with OpenSearch serverless, which is where we stored our index in the full-code **Working with Vector Stores** section.
However, both Knowledge Bases for Bedrock and Amazon Kendra are available options for this section.

Expand here to use Knowledge Bases for Amazon Bedrock

Expand here to use Amazon Kendra

6. Under **Exact Response** select **No**.
7. Skip the **Sample Utterances**, **Fulfillment** and **Closing response** sections.
8. Choose **Save intent**.
9. Select **Build** from the top of the page. Once the build completes, select **Test** to experiment with your bot that is now backed by either knowledge bases for Amazon Bedrock or Amazon Kendra.
10. Enter a sample question into the chatbot. The Lex bot will use your knowledge store to return a response from the documents you have indexed.

- What are our strategic goals?
- What is plan for volunteer retention?



Congratulations!
You have successfully created a Lex RAG chatbot!
Previous
Next

Full-code

Overview

1. Hosting a front end for GenAI
    - Full-code: Amazon Lex
      with AWS Lambda and Amazon bedrock

Previous
Next

Using a Lambda function with Amazon Bedrock

As an alternative to using the built-in QnAIntent, consider using Lambda for more control over the output and vector store selection.

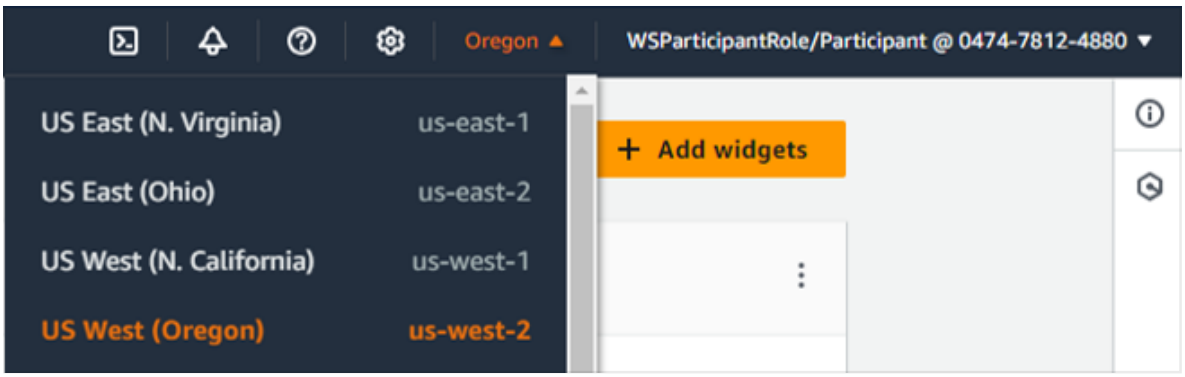
In order for the Lex bot to fall back to the Lambda function we create in this section, we need to delete the built-in QnAIntent you created in the previous section. If we do not delete the intent, then all inputs will continue to be routed to the QnAIntent instead of our Lambda function.

From the intents list, select radio button next to the QnAIntent and choose Delete.

In this section, we will back our Lex chatbot with Amazon Bedrock through an AWS Lambda function. We will first create a Lambda function, then we will attach the Lambda function to our Lex chatbot so that all inputs will be routed to Amazon Bedrock.

Important - Region Selection

Ensure that you're in the us-west-2 region (Oregon) when creating your Lambda function.

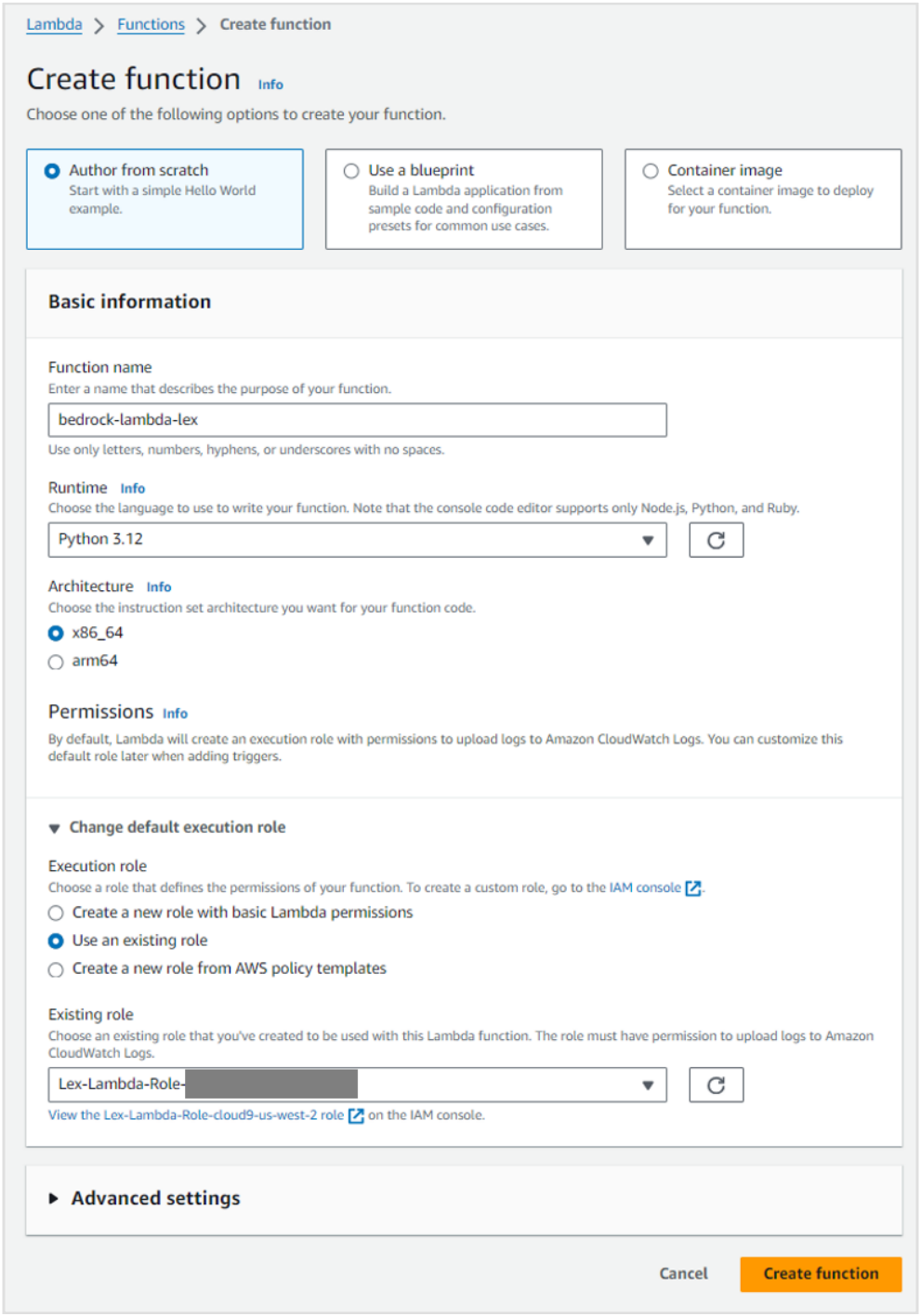


Create a Lambda function

1. Open the console for AWS Lambda

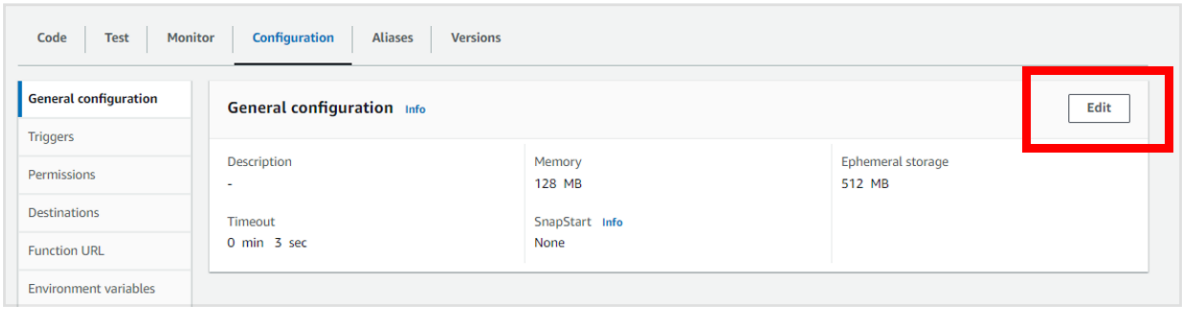
.

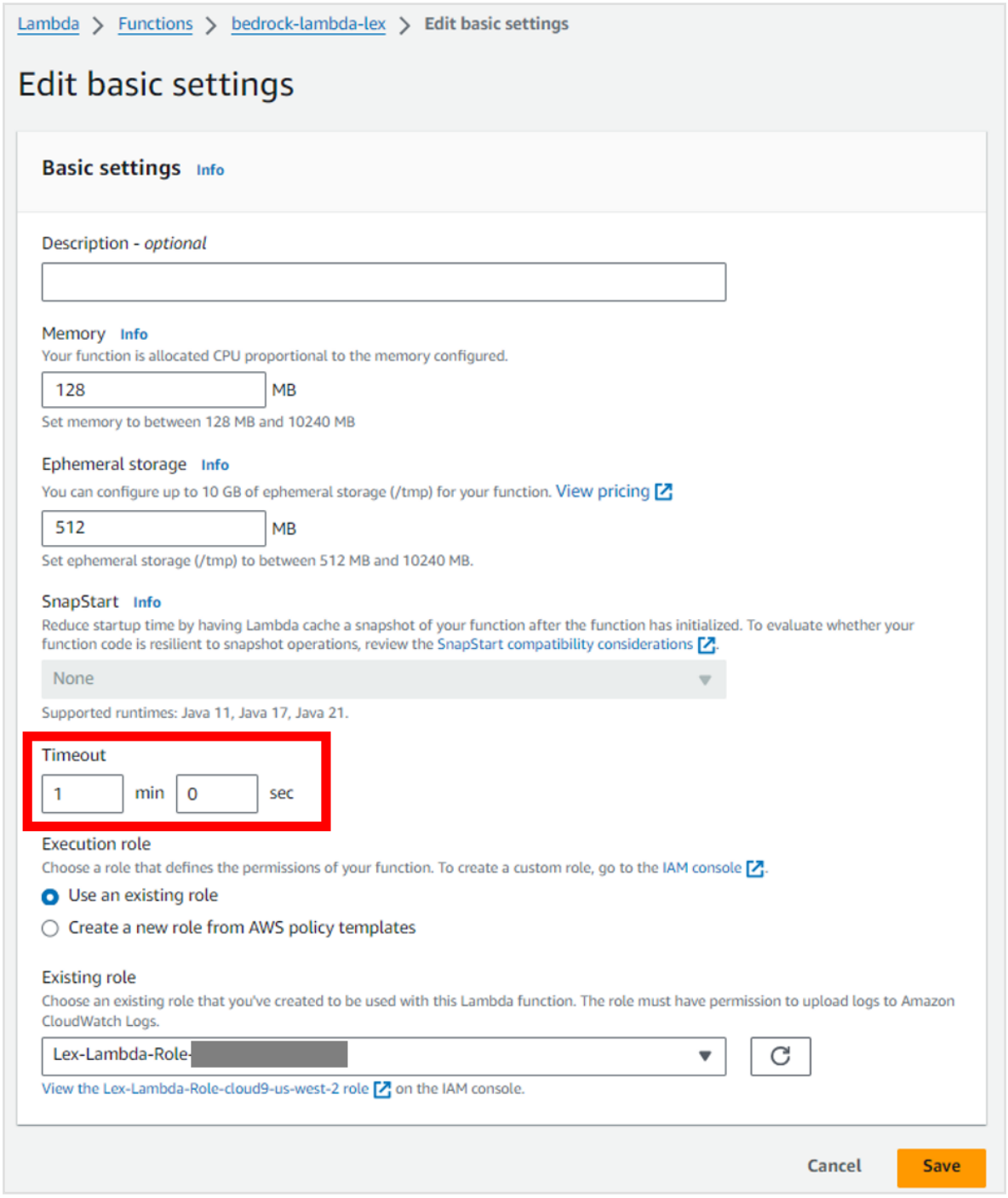
2. From the functions page, choose **Create function**.
3. Choose a function name (Exp. bedrock-lambda-lex).
4. Under **Runtime**, select Python 3.13.
5. Expand the **Change default execution role** and choose **Use an existing role**. Select the existing IAM role that starts with **Lex-Lambda-Role-**. This previously created role has the neccesary permissions to invoke models through Bedrock, and access the knowledge stores.
6. Select **Create function**.



7. Before updating the code, navigate to the **Configuration** tab and

update the timeout in the General configuration section. Up the timeout to 1 minute so that the function has time to invoke models through Bedrock.

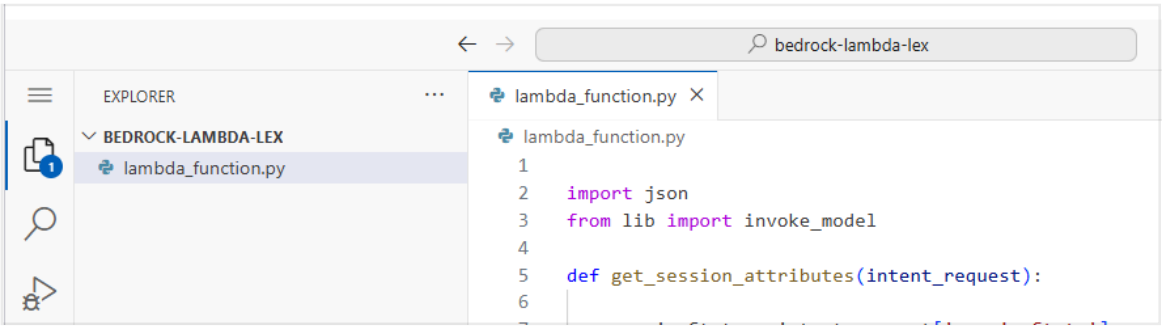




8. Select **Save** and navigate to the **Code** tab.

Update Lambda scripts

Our function will require two scripts. The first script will handle recieving inputs and submitting outputs to our Lex chatbot. The second script will operate as a backing library to handle Bedrock model invocations. This script will closely mirror the library scripts from previous lab sections.



1. From the **Code** tab, select the existing **lambda_function.py** script and replace the starter code with the code block below.

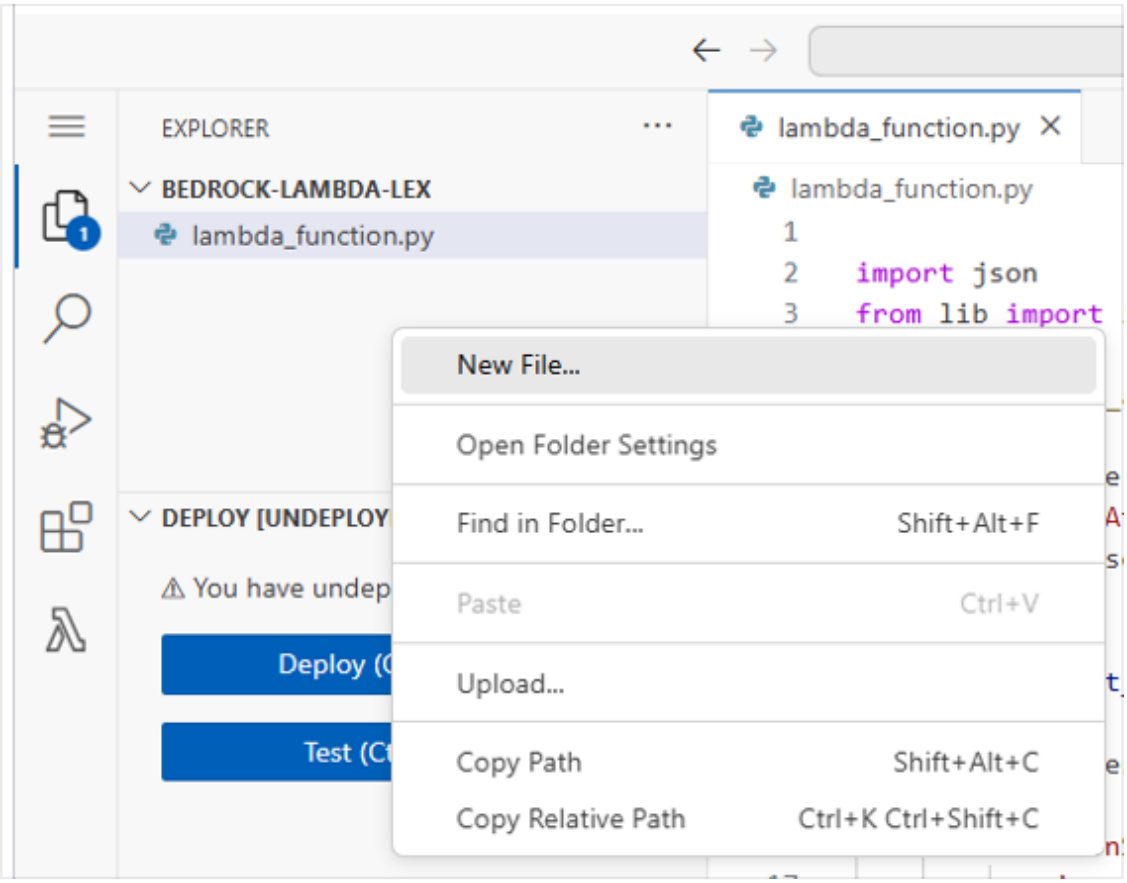
This script supports integration with Lex. The functions parse through context passed in from our Lex bot, and returns values from our Bedrock model output.

- **get_session_attributes()**: extracts relevant session context from the event body passed in from our Lex bot.
- **close()**: builds the response body that Lex needs to interpret the message from the Bedrock model output and showcase on the front end.
- **buildResponse()**: extracts the user input from the event body passed in from our Lex bot and submits to the backing library invoke_model function which executes a RAG process to generate a model response from our knowledge store.
- **lambda_handler()**: calls the requisite functions to recieve Lex context and return a response from our model.

```
import json
from lib import invoke_model

def get_session_attributes(intent_request):

    sessionState = intent_request['sessionState']
    if 'sessionAttributes' in sessionState:
        return sessionState['sessionAttributes']
    return {}

def close(intent_request, session_attributes, fulfillment_state, message):

    intent_request['sessionState']['intent']['state'] = fulfillment_state
    return {
        'sessionState': {
            'sessionAttributes': session_attributes,
            'dialogAction': {
                'type': 'Close'
            },
            'intent': intent_request['sessionState']['intent']
        },
```

```
    'messages': [message],
    'sessionId': intent_request['sessionId'],
    'requestAttributes': intent_request['requestAttributes'] if 'requestAttributes'
in intent_request else None
    }

def buildResponse(event):

    # pull the user input from the event
    query = event['inputTranscript']

    answer = invoke_model(query)

    fulfillment_state = "Fulfilled"

    session_attributes = get_session_attributes(event)

    message =  {
        'contentType': 'CustomPayload',
        'content': answer
    }

    return close(event, session_attributes, fulfillment_state, message)

def lambda_handler(event, context):

    print("Received event: %s" % json.dumps(event))

    output = buildResponse(event)
    print(output)

    return output
```

2. Right click in the environment directory and create a new file called **lib.py**.



3. From the **Code** tab, select the new **lib.py** script and paste one of the code blocks below.

This script will implement a RAG approach using Amazon Bedrock and one of the knowledge stores you configured in a previous lab. This section will require either a Knowledge Base for Amazon Bedrock, a Kendra index, or an OpenSearch Serverless collection. Expand the preferred section below, then copy and paste the provided code into the lib.py file. Notice that these scripts closely mirror the backing scripts from the previous RAG chatbot sections.

Expand here to use Knowledge Bases for Amazon Bedrock

Expand here to use Amazon Kendra

Expand here to use OpenSearch Serverless

6. From the **Code** tab, select **Deploy** to save the latest function. Now the lambda function is ready to execute through our Lex chatbot!

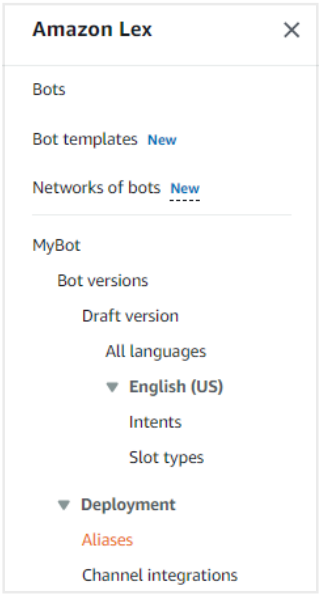
Attach our Lambda function to the Lex chatbot

To learn more about attaching lambda functions to Lex bots, view

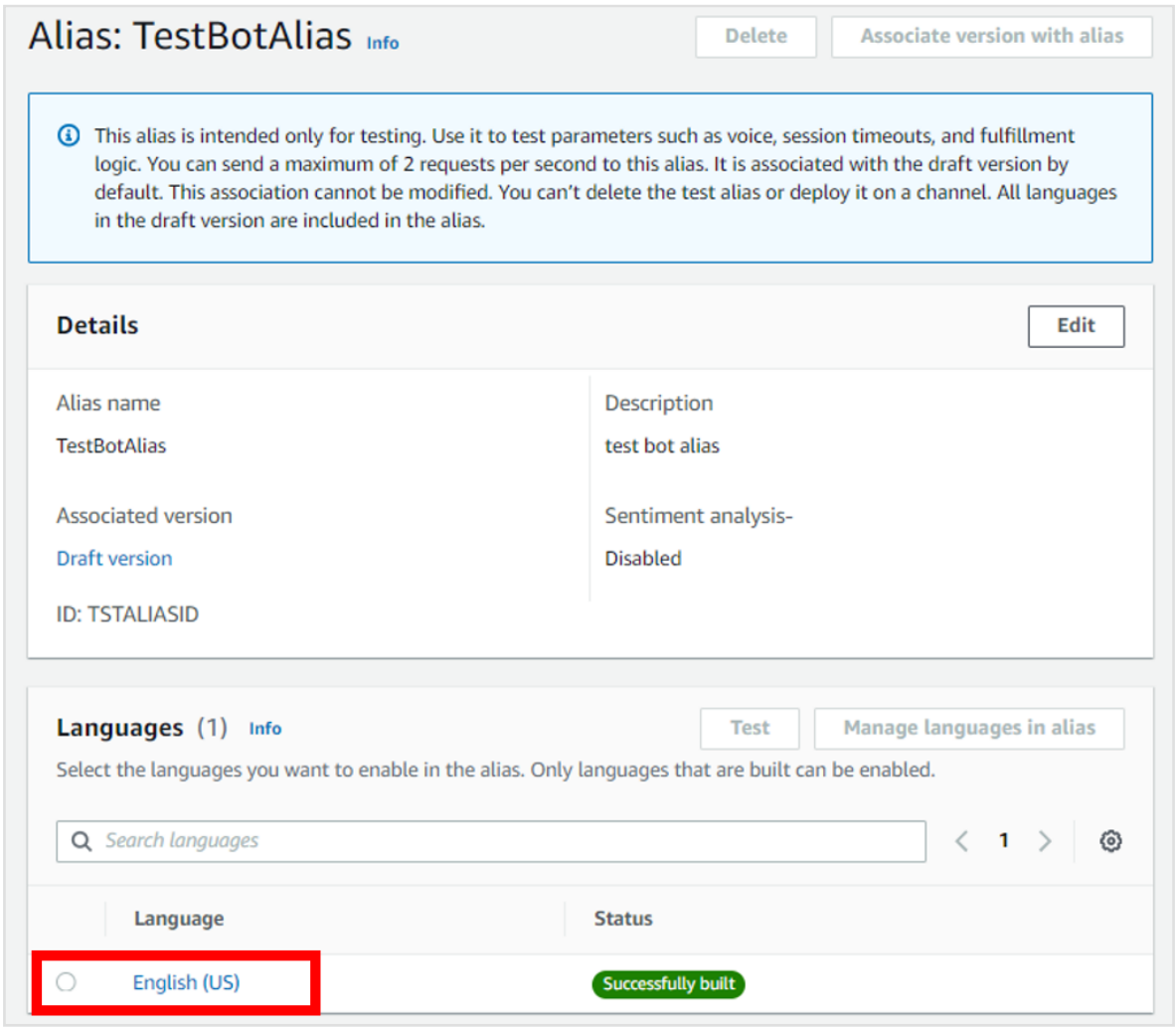
the Lex documentation

1. Open the AWS Management Console for Amazon Lex

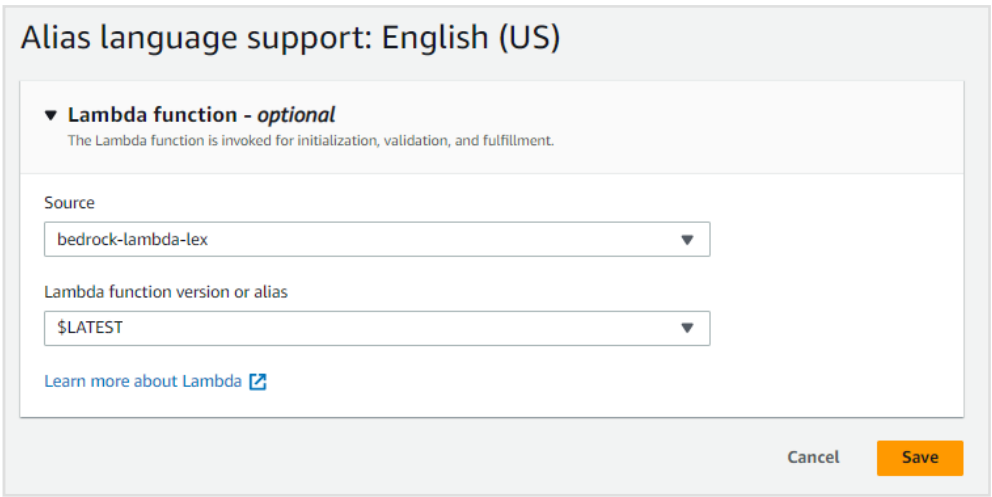
2. Click into the Lex bot that you created earlier.
3. In the navigation on the left, select **Aliases** under the **Deployment** menu.



4. Click into **TestBotAlias** then click into **English** under the **Languages** section.



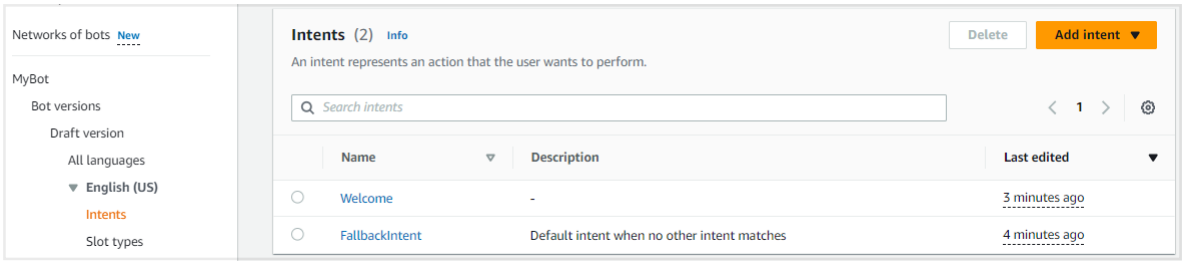
5. Lex bots can be linked to a single Lambda function per language. This single Lambda function supports all intents of your Lex bot through router functions. Learn more by visiting the Lex documentation
   .
6. Select your lambda function from the **Source** dropdown list. Use **$LATEST** for the lambda function version. Choose **Save**.



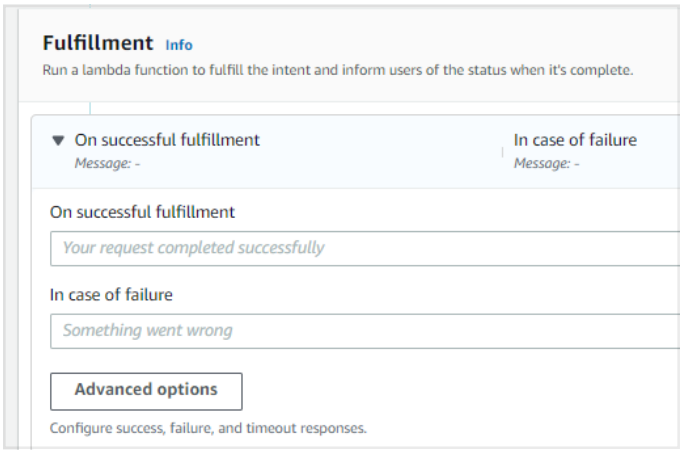
7. The final step is the set an intent to invoke the Lambda function. We

will align the **FallBackIntent** to invoke our lambda function. This means that all inputs that don't satisfy an existing intent or utterance will be sent to our Lambda function.

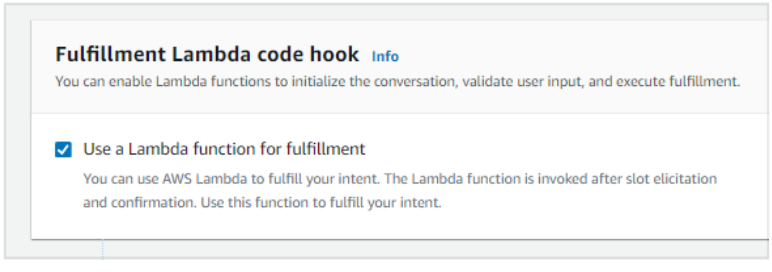
8. From the navigation pane on the left choose **Intents** from the sub menu under the bot's name.
9. Click into the **FallbackIntent**, which is created by default when you first create a bot.



10. Scroll down to the Fulfillment section, expand the **On successful fulfillment** section and choose **Advanced options**.



11. Check the box for **Use a lambda function for fulfillment**. This will link the lambda function you associated with the alias to any fulfillments that come through the FallbackIntent. Select **Update options**.



12. Choose **Save intent**.

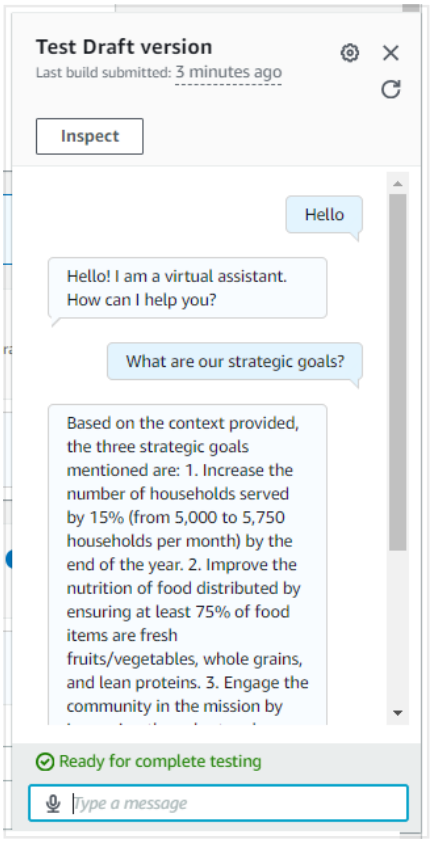
In order for the Lex bot to fall back to the Lambda function we create in this section, we need to delete the built-in QnAIntent

you created in the previous section. If we do not delete the intent, then all inputs will continue to be routed to the QnAIntent instead of our Lambda function.

From the intents list, select radio button next to the QnAIntent and choose Delete.

13. Select **Build** from the top of the page. Once the build completes, select **Test** to experiment with your bot.
14. Try out a few sample questions and see the results. These questions will go through our FallbackIntent since we don't have any utterances configured to invoke another intent. These questions will run through our Lambda function, which uses the Amazon Bedrock converse API to produce a response from your previously configured knowledge store. Feel free to review the Cloudwatch logs associated with the Lambda function.

- What are our strategic goals?
- What is our target metric for volunteer retention?
- What is our expansion plan for the future?



Congratulations!

You have successfully built a RAG chatbot using Amazon Lex and AWS Lambda invoking Amazon Bedrock!

Previous

Next

Deploying with Lex-Web-UI

In this section you developed a RAG chatbot on Lex backed by GenAI. As a next step, you may want to deploy this chatbot to a website. While this workshop does not currently include deployment in the scope, you can review the steps below for deplying Lex to custom websites using the Lex-Web-UI.

Introducing the Lex-Web-UI

The AWS Lex-Web-UI is a prebuilt fully featured web client for Amazon Lex chatbots. It eliminates the heavy lifting of recreating a chat UI from scratch. You can quickly deploy its features and minimize time to value for your chatbot-powered applications.

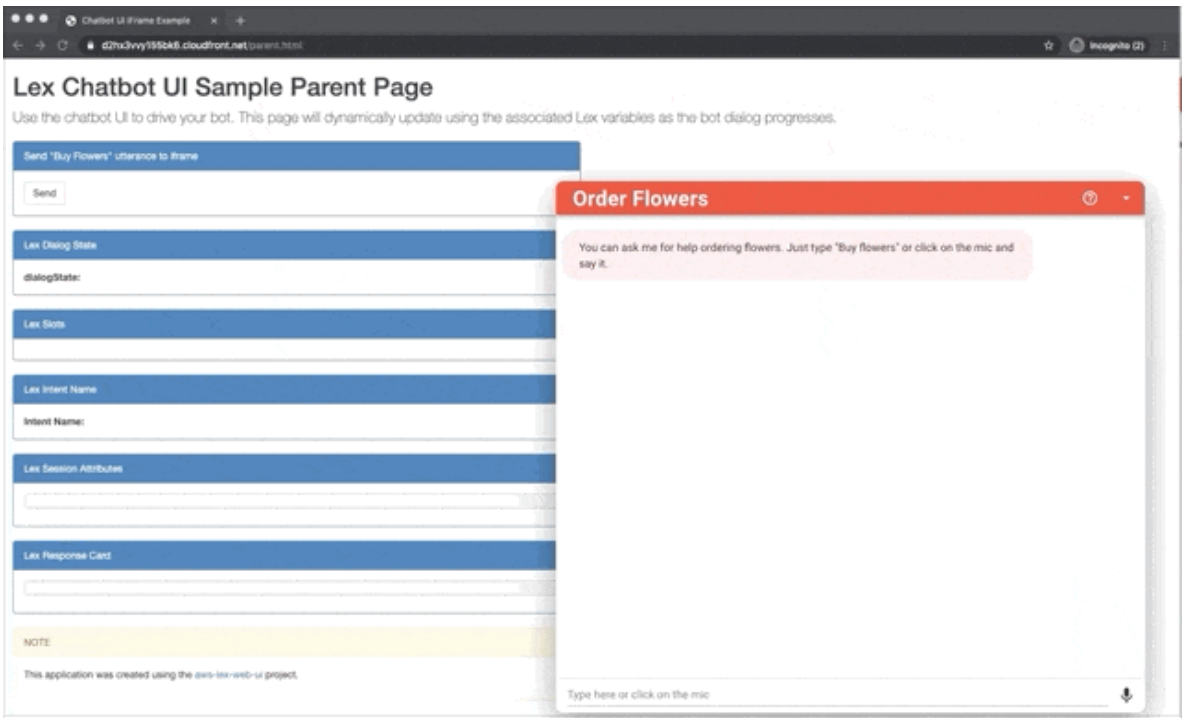
The Lex bots that you created previously can be deployed to HTML websites using the Lex Web UI cloudformation templates.

For full instructions on deploying the Lex web UI, review these steps in the Amazon Lex workshop

.

For more detail, visit the Lex web UI Github repo

.



Previous
Next

Summary

Congratulations on completing the workshop!

We hope you now are comfortable with the basics of building generative AI prototypes. We encourage you to start customizing the code and prompts to solve your business problems!

You can learn more below:

- Amazon Bedrock service page
- Amazon Bedrock User Guide
- Amazon Bedrock API Reference
- Streamlit API reference
- Amazon Bedrock Workshop
- LangChain documentation
- Solutions for Machine Learning (AI/ML)
- Workshop: Building generative AI applications with Amazon Bedrock using agents

Model provider-specific documentation:

- AI21 documentation
- Amazon Titan documentation
- Anthropic documentation
- Cohere documentation
- Stability AI documentation

Credits

- **Lead authors:** Ben Turnbull, David Marsh
- **Original Content from:** Building with Amazon Bedrock and LangChain
- **Amazon Q Business Content from:** AWS Gen AI Workshop

Architecture images based on content from the Amazon Bedrock Workshop

Previous