Below is a comprehensive, step‐by‐step guide documenting the entire process—from setting up your AWS Lambda function integrated with Amazon Kendra and Groq (as an alternative to Bedrock) to packaging and testing your solution. This guide is written in a clear and structured format so that even a novice can follow along.

**Comprehensive Guide: Building a FAQ Chatbot with AWS Lambda, Amazon Kendra, and Groq**

This documentation explains how to build a serverless FAQ chatbot using AWS services. The chatbot retrieves context from an Amazon Kendra index and uses Groq’s API to generate natural language responses.

**1. Prerequisites**

Before you begin, ensure you have:

* An active AWS account with permissions to create and manage Lambda functions, Kendra indexes, and IAM roles.
* Familiarity with the AWS Management Console and basic command-line operations.
* AWS CloudShell access (or a local Docker environment) to build your deployment package in an Amazon Linux environment.
* Environment variables for:
  + **KENDRA\_INDEX\_ID:** Your Amazon Kendra index ID.
  + **GROQ\_API\_KEY:** Your Groq API key.
  + **GROQ\_MODEL\_ID:** The identifier for the Groq model you wish to use (e.g., "llama3-8b-8192").

**2. Overview of the Architecture**

1. **AWS Lambda Function:**  
   Hosts your Python code. It:
   * Receives a user query.
   * Calls Amazon Kendra to retrieve relevant FAQ context.
   * Constructs a prompt using the Kendra context.
   * Invokes the Groq API (using an asynchronous client) to generate a response.
2. **Amazon Kendra:**  
   Indexes your FAQ data. The Lambda function queries this index for context related to the user’s query.
3. **Groq API:**  
   Generates a natural language answer based on the prompt (which includes both the user’s query and the context from Kendra).

**3. Step-by-Step Process**

**A. Setting Up Your AWS Lambda Function**

1. **Create the Lambda Function:**
   * Log in to the [AWS Lambda Console](https://console.aws.amazon.com/lambda/).
   * Click **Create function** → **Author from scratch**.
   * **Function Name:** e.g., CentriaFAQChatbot.
   * **Runtime:** Select **Python 3.9**.
   * **Architecture:** Choose **x86\_64** (to match the dependencies you’ll build).
   * For now, let AWS create a default execution role with basic CloudWatch logging permissions.
2. **Configure Environment Variables:**
   * In your Lambda function’s configuration tab, add the following environment variables:
     + KENDRA\_INDEX\_ID: Your Kendra index ID (e.g., "dfa7f843-4fb6-4a8a-a100-75b894a726de").
     + GROQ\_API\_KEY: Your Groq API key.
     + GROQ\_MODEL\_ID: Your Groq model identifier (e.g., "llama3-8b-8192").

**B. Developing the Lambda Function Code**

Your Lambda function will:

* Query Amazon Kendra for context.
* Construct a prompt incorporating the context and the user's query.
* Call the Groq API using an asynchronous client.
* Return the generated answer.

Below is the full Python code for lambda\_function.py:

import os

import json

import boto3

import asyncio

from groq import AsyncGroq # Ensure the Groq library is included in your deployment package or via a Lambda layer

# Retrieve environment variables

KENDRA\_INDEX\_ID = os.environ.get("KENDRA\_INDEX\_ID") # e.g., "dfa7f843-4fb6-4a8a-a100-75b894a726de"

GROQ\_API\_KEY = os.environ.get("GROQ\_API\_KEY") # Your Groq API key

GROQ\_MODEL\_ID = os.environ.get("GROQ\_MODEL\_ID") # e.g., "llama3-8b-8192"

# Initialize AWS Kendra client (adjust region if needed)

kendra\_client = boto3.client("kendra", region\_name="eu-west-1")

# Initialize Groq client

groq\_client = AsyncGroq(api\_key=GROQ\_API\_KEY)

async def generate\_groq\_response(prompt, user\_query):

"""

Generate a refined response using the Groq API.

"""

try:

response = await groq\_client.chat.completions.create(

messages=[

{

"role": "system",

"content": "You are an assistant. Provide user-friendly responses based on the provided context.",

},

{

"role": "user",

"content": f"The user asked: '{user\_query}'. Here is the context: {prompt}. Provide a detailed answer.",

}

],

model=GROQ\_MODEL\_ID,

max\_tokens=300, # Adjust as needed

temperature=0.7, # Adjust as needed

top\_p=0.9 # Adjust as needed

)

return response

except Exception as e:

print(f"Error invoking Groq: {e}")

return None

def lambda\_handler(event, context):

# Extract the user's query from the event payload.

user\_query = event.get("query", "Hello, what can I help you with?")

# Query Kendra for context (fetch up to 3 results).

try:

kendra\_response = kendra\_client.query(

IndexId=KENDRA\_INDEX\_ID,

QueryText=user\_query,

PageSize=3

)

except Exception as e:

print("Error querying Kendra:", str(e))

kendra\_response = {}

# Extract context snippets from Kendra results.

context\_snippets = []

for item in kendra\_response.get("ResultItems", []):

snippet = item.get("DocumentExcerpt", {}).get("Text", "")

if snippet:

context\_snippets.append(snippet)

context\_text = "\n".join(context\_snippets)

# Build the prompt for Groq, including context if available.

if context\_text:

prompt = f"Using the following context, answer the question:\n\nContext:\n{context\_text}\n\nQuestion: {user\_query}"

else:

prompt = f"Question: {user\_query}"

# Run the asynchronous Groq call.

try:

try:

loop = asyncio.get\_event\_loop()

except RuntimeError:

loop = asyncio.new\_event\_loop()

asyncio.set\_event\_loop(loop)

groq\_response = loop.run\_until\_complete(generate\_groq\_response(prompt, user\_query))

except Exception as e:

print("Error running Groq async call:", str(e))

groq\_response = None

# Extract the answer from the Groq response.

try:

# Based on documentation, extract answer using:

# groq\_response.choices[0].message.content

if groq\_response is not None and hasattr(groq\_response, "choices") and len(groq\_response.choices) > 0:

answer = groq\_response.choices[0].message.content

else:

answer = "No response from Groq."

except Exception as e:

print("Error extracting answer from Groq response:", str(e))

answer = "Error generating answer."

return {

"statusCode": 200,

"body": json.dumps({"answer": answer})

}

**Key Notes:**

* **Kendra Query:**  
  The function retrieves context snippets using Kendra’s Query API.
* **Groq Integration:**  
  An asynchronous call is made to Groq’s chat completions endpoint. The answer is extracted using the documented attribute chain:  
  groq\_response.choices[0].message.content
* **Async Execution:**  
  The code uses asyncio to run the Groq call synchronously within the Lambda handler.
* **Error Handling:**  
  Various try/except blocks catch errors, logging them and providing fallback messages.

**C. Creating Your Deployment Package**

To deploy your Lambda function code along with its dependencies (including Groq, pydantic\_core, and others), you need to build your package in an environment that matches AWS Lambda’s architecture (Python 3.9, x86\_64).

**Using Docker with the Official AWS Lambda Python 3.9 Image**

1. **Pull and Run the Container:**
2. docker run -it --rm -v "$PWD/lambda\_package":/var/task public.ecr.aws/lambda/python:3.9 bash

This command:

* + Uses the official AWS Lambda Python 3.9 image.
  + Mounts your local directory lambda\_package to /var/task in the container.
  + Opens a bash shell in the container.

1. **Inside the Container:**
   * **Verify Python Version:**
   * python3.9 --version
   * **Create a Directory for Dependencies:**
   * cd /var/task
   * mkdir python
   * **Install Dependencies:**

To ensure that compiled binaries are built for the Lambda architecture (x86\_64), force pip to use the manylinux2014\_x86\_64 platform. For example:

python3.9 -m pip install groq --target python --platform manylinux2014\_x86\_64 --only-binary=:all:

python3.9 -m pip install pydantic\_core==2.27.1 --target python --platform manylinux2014\_x86\_64 --only-binary=:all:

# Repeat for any other dependencies needed.

* + **Ensure Your Lambda Code Is in /var/task:**

If you haven’t already created your Lambda code file, you can create it directly:

nano lambda\_function.py

Paste your code and save it (Ctrl+O, Enter, Ctrl+X).

* + **Package Everything into a Zip File:**
  + zip -r9 lambda\_deployment\_package.zip lambda\_function.py python
  + **Exit the Container:**
  + exit

**D. Uploading Your Deployment Package**

1. **Using AWS CLI from CloudShell:**
2. aws lambda update-function-code --function-name CentriaFAQChatbot --zip-file fileb://lambda\_deployment\_package.zip
3. **Or via the AWS Lambda Console:**
   * Log in to the AWS Lambda Console.
   * Open your function (e.g., CentriaFAQChatbot).
   * In the Code tab, choose “Upload from .zip file” and select your lambda\_deployment\_package.zip.
   * Save changes.

**E. Testing and Monitoring**

1. **Test Your Lambda Function:**
   * In the AWS Lambda Console, create a test event (e.g., with JSON payload {"query": "What are the admission requirements?"}) and run the test.
   * Verify that the output returns a well-formed answer.
2. **Check CloudWatch Logs:**
   * Monitor the logs for any errors or warnings and adjust the code or dependencies as necessary.
3. **Refine Parameters:**
   * If the responses need tuning, adjust the inference parameters (max\_tokens, temperature, top\_p) in your Lambda code.
   * Adjust the Kendra query settings if you need more relevant context.
4. **Integrate with Your Front End:**
   * Once your Lambda function is stable, you can expose it via API Gateway or a Lambda Function URL so that your chatbot front end can interact with it.

**4. Summary**

* **Setup:**  
  Create and configure a Lambda function with Python 3.9 and appropriate environment variables.
* **Development:**  
  Write code that integrates Amazon Kendra for context retrieval and uses Groq’s asynchronous API to generate responses.
* **Packaging:**  
  Build your deployment package in an environment matching AWS Lambda’s architecture (using Docker with the official Python 3.9 image) and force pip to install dependencies for the x86\_64 architecture.
* **Deployment & Testing:**  
  Upload your package to AWS Lambda, test thoroughly, and adjust as needed.