**Plot lambda function,**  
  
import json

import boto3

bedrock = boto3.client("bedrock-runtime", region\_name="ap-south-1")

def lambda\_handler(event, context):

    try:

        # Extract from Bedrock Agent format

        parameters = {p["name"]: p["value"] for p in event.get("parameters", [])}

        question = parameters.get("question", "")

        query = parameters.get("query", "")

        table\_data = json.loads(parameters.get("table\_data", "[]"))

        if not question or not table\_data:

            return respond\_with\_text("❌ Missing required inputs: question and table\_data.")

        # Step 1: Classify chart type

        chart\_type\_prompt = f"""

Given the following user query and table data, decide the most appropriate chart type from:

- line

- bar

- pie

- scatter

- histogram

- box

User query: "{question}"

Only return one word (the chart type) with no explanation.

"""

        chart\_type = classify\_chart\_type(chart\_type\_prompt)

        # Step 2: Generate visualization code

        code\_prompt = f"""

Generate Python code using matplotlib or plotly to create a '{chart\_type}' chart.

User query: "{question}"

The data is available as a Python variable named `data` which is a list of dictionaries like:

{json.dumps(table\_data[:3], indent=2)}

Use appropriate labels and formatting. Return only the code with no explanation.

"""

        code = generate\_code\_from\_llm(code\_prompt)

        result\_message = f"""✅ Visualization Type: \*\*{chart\_type}\*\*

```python

{code}

```"""

        return respond\_with\_text(result\_message)

    except Exception as e:

        return respond\_with\_text(f"❌ Lambda error: {str(e)}")

def classify\_chart\_type(prompt):

    response = bedrock.invoke\_model(

        modelId="anthropic.claude-3-sonnet-20240229-v1:0",

        body=json.dumps({

            "messages": [{"role": "user", "content": prompt}],

            "max\_tokens": 10,

            "anthropic\_version": "bedrock-2023-05-31"

        }),

        contentType="application/json",

        accept="application/json"

    )

    output = json.loads(response["body"].read())

    return output["content"][0]["text"].strip().lower()

def generate\_code\_from\_llm(prompt):

    response = bedrock.invoke\_model(

        modelId="anthropic.claude-3-sonnet-20240229-v1:0",

        body=json.dumps({

            "messages": [{"role": "user", "content": prompt}],

            "max\_tokens": 500,

            "anthropic\_version": "bedrock-2023-05-31"

        }),

        contentType="application/json",

        accept="application/json"

    )

    output = json.loads(response["body"].read())

    return output["content"][0]["text"].strip()

def respond\_with\_text(message):

    return {

        "messageVersion": "1.0",

        "response": {

            "actionGroup": "visualization\_action\_group",

            "function": "generate\_visualization\_code",

            "functionResponse": {

                "responseBody": {

                    "TEXT": {

                        "body": message

                    }

                }

            }

        }

    }

**--------------------------------------------------------------------------------------------------------------**

**Intent lambda function,**  
  
import json

import boto3

import logging

logger = logging.getLogger()

logger.setLevel(logging.INFO)

bedrock = boto3.client("bedrock-runtime", region\_name="ap-south-1")

def lambda\_handler(event, context):

    logger.info(f"Received event: {json.dumps(event, default=str)}")

    try:

        action\_group = event.get('actionGroup', 'intent-finder')

        function\_name = event.get('function', 'classify\_intent')

        parameters = event.get('parameters', [])

        user\_query = ""

        for param in parameters:

            if param.get("name") == "query":

                user\_query = param.get("value", "")

                break

        if not user\_query:

            return create\_error\_response(action\_group, function\_name, "Missing 'query' parameter.")

        prompt = f"""

Classify the intent of this healthcare query into one of the following categories:

1. table – When the user is expecting raw data, tabular output, or detailed lists.

   Examples:

   - "List all ICU patients admitted this month"

   - "Show me the patient details for heart failure"

2. visualization – When the user expects a chart, graph, or visual summary of data.

   Examples:

   - "Show a trend of admissions over the last 6 months"

   - "Display a pie chart of patient diagnoses"

3. text – When the user expects a summary, explanation, or insight in natural language.

   Examples:

   - "Summarize ICU usage in the last week"

   - "Explain the change in patient admissions this quarter"

Query: "{user\_query}"

Respond with only one word: table, visualization, or text.

"""

        bedrock\_response = bedrock.invoke\_model(

            modelId="anthropic.claude-3-sonnet-20240229-v1:0",

            body=json.dumps({

                "messages": [

                    {"role": "user", "content": prompt}

                ],

                "max\_tokens": 20,

                "anthropic\_version": "bedrock-2023-05-31"

            }),

            contentType="application/json",

            accept="application/json"

        )

        output = json.loads(bedrock\_response["body"].read())

        intent = output.get("content", [{}])[0].get("text", "").strip().lower()

        return create\_success\_response(action\_group, function\_name, {

            "query": user\_query,

            "intent": intent

        })

    except Exception as e:

        logger.error(f"Error during intent detection: {str(e)}")

        return create\_error\_response(action\_group, function\_name, f"Function execution error: {str(e)}")

def create\_success\_response(action\_group, function\_name, result):

    return {

        "messageVersion": "1.0",

        "response": {

            "actionGroup": action\_group,

            "function": function\_name,

            "functionResponse": {

                "responseBody": {

                    "TEXT": {

                        "body": json.dumps(result, indent=2)

                    }

                }

            }

        }

    }

def create\_error\_response(action\_group, function\_name, error\_message):

    result = {"error": True, "message": error\_message}

    return {

        "messageVersion": "1.0",

        "response": {

            "actionGroup": action\_group,

            "function": function\_name,

            "functionResponse": {

                "responseBody": {

                    "TEXT": {

                        "body": json.dumps(result, indent=2)

                    }

                }

            }

        }

    }

**--------------------------------------------------------------------------------------------------------------**

**Intent Test passing json,**

{

"parameters": [

{

"name": "query",

"value": "I want to see how many surgeries are being cancelled each month"

}

]

}

**Intent Query executor lambda function,**  
  
import boto3

import time

import json

import logging

logger = logging.getLogger(\_\_name\_\_)

logger.setLevel(logging.INFO)

ATHENA\_DATABASE = 'insightsrx\_db'

ATHENA\_OUTPUT\_BUCKET = 's3://insightsrxdata/AWS\_Logs/'  # external S3 path

athena = boto3.client("athena")

def lambda\_handler(event, context):

    try:

        # 🔧 Extract sql\_query from parameters correctly

        parameters = event.get("parameters", [])

        sql\_query = ""

        for param in parameters:

            if param.get("name") == "sql\_query":

                sql\_query = param.get("value", "")

                break

        if not sql\_query:

            return respond\_with\_text("❌ No SQL query provided.")

        # Start Athena query

        response = athena.start\_query\_execution(

            QueryString=sql\_query,

            QueryExecutionContext={'Database': ATHENA\_DATABASE},

            ResultConfiguration={'OutputLocation': ATHENA\_OUTPUT\_BUCKET}

        )

        execution\_id = response['QueryExecutionId']

        # Wait for query to complete

        max\_wait = 30

        for \_ in range(max\_wait):

            status = athena.get\_query\_execution(QueryExecutionId=execution\_id)

            state = status['QueryExecution']['Status']['State']

            if state in ['SUCCEEDED', 'FAILED', 'CANCELLED']:

                break

            time.sleep(2)

        if state != 'SUCCEEDED':

            reason = status['QueryExecution']['Status'].get('StateChangeReason', 'Unknown error')

            return respond\_with\_text(f"❌ Athena query failed. Reason: {reason}")

        # Retrieve results

        result\_set = athena.get\_query\_results(QueryExecutionId=execution\_id)

        rows = result\_set['ResultSet']['Rows']

        if not rows or len(rows) < 2:

            return respond\_with\_text("✅ Query executed successfully, but no data found.")

        # Extract headers

        headers = [col.get('VarCharValue', '') for col in rows[0]['Data']]

        table\_lines = [' | '.join(headers)]

        # Extract data rows

        for row in rows[1:]:

            values = [col.get('VarCharValue', '') for col in row['Data']]

            table\_lines.append(' | '.join(values))

        # Join all into a string

        table\_string = "\n".join(table\_lines)

        # Output summary

        summary = f"✅ SQL executed successfully. Output preview:\n\n{table\_string}"

        return respond\_with\_text(summary)

    except Exception as e:

        logger.error(f"Error: {str(e)}")

        return respond\_with\_text(f"❌ Lambda error: {str(e)}")

def respond\_with\_text(message: str):

    return {

        "messageVersion": "1.0",

        "response": {

            "actionGroup": "execute\_sql\_queries",

            "function": "query\_executor",

            "functionResponse": {

                "responseBody": {

                    "TEXT": {

                        "body": message

                    }

                }

            }

        }

    }