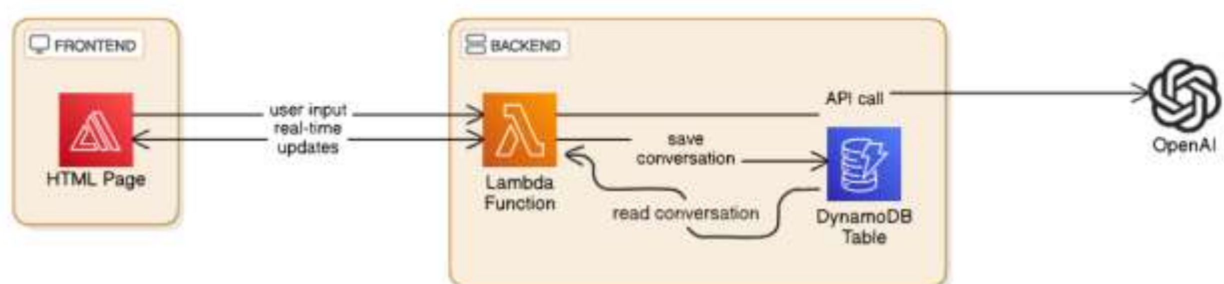
Lab: AWS Lambda Function for Open AI API

Lab overview and objectives

In this lab, you will create an AWS Lambda function using Python that will call Open AI APIs.

After completing this lab, you should be able to:

- Create a Python based **AWS Lambda** function from the **AWS Management Console** that will call Open AI APIs.
- Create an **AWS Lambda layer**.
- Test the functions from **AWS Lambda Test** tab.
- Execute the functions from a Web Browser using **Function URL**.
- Execute the functions from and HTML page using **Function URL**.
- Create a **DynamoDB** table to keep conversation history.
- Enhance the Lambda API and HTML to support chat history.



Scenario

You will create a Lambda function that receives a JSON payload with three parameters in the request body (instead of query parameters in GET). Based on these parameters, a calculation will be performed, and the result will be returned as a JSON response.

Accessing the AWS Management Console

For this lab we will use the **AWS Academy learner** lab and will be using the **AWS Management Console**.

Task 1: Create Python chat using Open AI API

In this task, you will write a Python program that interacts with the OpenAI API to generate responses based on user input. You will also learn about the API key and how it is used in the context of the OpenAI API.

The API key is a unique identifier used to authenticate requests associated with your OpenAI account. It is essential to keep your API key secure and not expose it in public repositories or client-side code. In this task, you will use the API key to initialize the OpenAI client.

1. *Write the next python code:*
    - *Import the OpenAI library:*
        1. *Start by importing the `OpenAI` class from the `openai` module.*
    - *Initialize the OpenAI client:*
        1. *Create an instance of the `OpenAI` client using your API key.*
    - *Set up the conversation history:*
        1. *Initialize an empty list to store the conversation history.*
    - *Create a loop to interact with the user:*
        1. *Use a `while` loop to continuously prompt the user for input until they type 'exit'.*
        2. *Append each user input to the conversation history.*
    - *Generate a response from the AI:*
        1. *Use the `responses.create` method of the OpenAI client to generate a response based on the conversation history.*
    - *Print the AI's response:*
        1. *Print the AI's response to the console.*
        2. *Append the AI's response to the conversation history.*
    - *Use a low-cost model like 'gpt-4.1-nano' model*

Note about the conversation history: This conversation history is stored only in memory and will reset every time the Lambda function is invoked. In later tasks, we will store chat history in DynamoDB so that the Lambda function can remember previous interactions and pass them to OpenAI.

2. Alternatively, use the following code, make sure to replace `<OPEN_AI_API_KEY>` with your own key:

```
from openai import OpenAI

# Initialize OpenAI client
client = OpenAI(api_key="YOUR_API_KEY")

# Conversation history
conversation = []

print("AI Chat started. Type 'exit' to quit.\n")

while True:

    # Prompt user for input
    user_input = input("You: ")

    if user_input.lower() == "exit":
        print("Goodbye!")
        break

    # Add user message to conversation history
    conversation.append({
        "role": "user",
        "content": user_input
    })

    # Generate AI response
    response = client.responses.create(
        model="gpt-4.1-nano",
        input=conversation
    )

    # Extract the AI text response
    ai_reply = response.output_text

    # Print response
    print("AI:", ai_reply)

    # Add AI response to conversation history
    conversation.append({
        "role": "assistant",
        "content": ai_reply
    })
```

3. Run the code, verify it's working by asking any question:

```
Enter your prompt (or 'exit' to quit): what is Open AI API Key?
AI: OpenAI API Key is a unique authentication key provided by OpenAI, a
leading artificial intelligence research lab, that allows developers to
securely access and use their AI models and tools through their API
(Application Programming Interface). The API Key acts as a secure token
that verifies the identity of the user and grants them access to the
resources and services provided by OpenAI. Developers can use this key
to integrate OpenAI's powerful AI capabilities into their own
applications, tools, and services.
Enter your prompt (or 'exit' to quit):
```

Task 2: Create a Python AWS Lambda function

In this task, we will write a lambda function in python, based on the code we wrote in task 1.

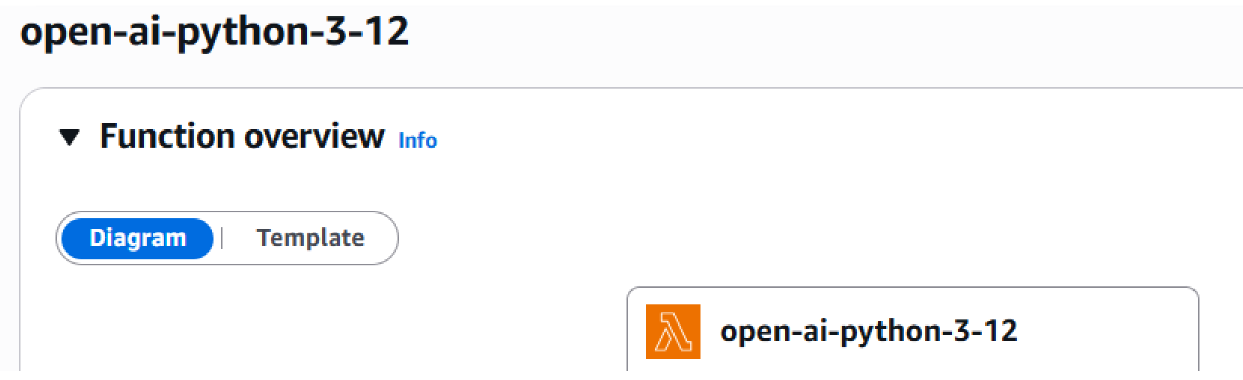
1. *Write the next AWS Lambda function in python code:*
   - *Import the OpenAI library:*
     1. *Import necessary libraries:*
        1. *Import the `json` module for handling JSON data.*
        2. *Import the OpenAI class from the openai library to interact with the OpenAI API.*
   - *Define the Lambda handler function:*
     1. *Create a function named `lambda_handler` that takes `event` and `context` as parameters.*
   - *Initialize the OpenAI client:*
     1. *Create an instance of the `OpenAI` client using your API key.*
   - *Parse the Request Body:*
     1. *Try to parse the body of the event as JSON. If parsing fails, return a 400 status code with an error message.*
     2. *Extract the user_prompt from the parsed body.*
   - *Validate the user prompt:*
     1. *Check if the `user_prompt` is present. If not, return a 400 status code with an error message.*
   - *Prepare the Conversation History:*
     1. *Initialize an empty list for conversation_history.*
     2. *Append the user_prompt to the conversation_history.*
   - *Generate a response from the AI:*

    1. *Use the `client.responses` method of the OpenAI client to generate a response based on the conversation history.*
- *Return the AI's response:*
    1. *Return a 200 status code with the AI's response in the body.*
- *Use a low-cost model like 'gpt-4.1-nano' model*

2. Alternatively, use the following code, make sure to replace <OPEN_AI_API_KEY> with your own key:

```
import json
from openai import OpenAI

def lambda_handler(event, context):

    client = OpenAI(api_key="YOUR_API_KEY")

    # Parse request body
    try:
        body = json.loads(event.get("body", "{}"))
    except Exception:
        return {
            "statusCode": 400,
            "body": json.dumps({"error": "Invalid JSON in request
body"})
        }

    user_prompt = body.get("user_prompt")

    # Validate prompt
    if not user_prompt:
        return {
            "statusCode": 400,
            "body": json.dumps({"error": "Missing 'user_prompt'
parameter"})
        }

    conversation_history = []
    conversation_history.append({
        "role": "user",
        "content": user_prompt
    })

    try:
        response = client.responses.create(
            model="gpt-4.1-nano",
            input=conversation_history
        )

        ai_reply = response.output_text

        return {
            "statusCode": 200,
            "body": json.dumps({
                "response": ai_reply
            })
        }

    except Exception as e:
        return {
            "statusCode": 500,
            "body": json.dumps({"error": str(e)})
        }
```

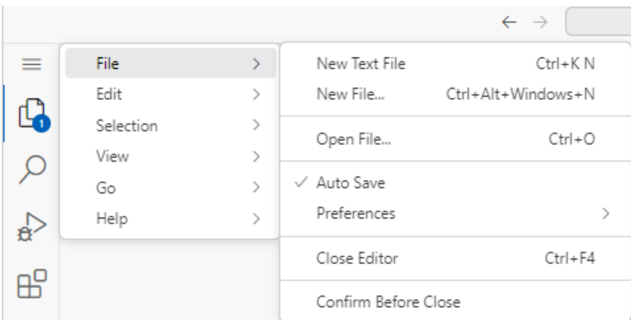
3. In **AWS Management Console**, In the search box to the right of **Services**, search for and choose **Lambda** to open the AWS Lambda console.
4. Choose **Create a function**.
5. In the **Create function** screen, configure these settings:
   - Choose **Author from scratch**
   - Function name: `open-ai-python-3-12`
   - Runtime: **Python 3.12**
   - Choose **Change default execution role**
   - Execution role: **Use an existing role**
   - Existing role: From the dropdown list, choose **LabRole**
6. Choose **Create function**.
7. The below should appear.



Task 3: Configure the Lambda function

In this task, you will paste the lambda function code you wrote in task 2.

12. Below the **Function overview** pane, choose **Code**, and then choose *lambda_function.py* to display and edit the Lambda function code.
13. In the **Code source** pane, delete the existing code, and paste the code you wrote in task 2.
14. By default, file saving is marked as **Auto Save**. To verify this, Choose the **File** menu and look for the **V** next to the **Auto Save** option.

    Alternatively, **Save** the changes.

14. In the **Code source** box, choose **Deploy**.

    Your Lambda function is now fully configured.

    Successfully updated the function **open-ai-python-3-10**

Task 4: Verify that the Lambda function works

19. Choose **Test** tab, and update the next JSON in the **Event JSON** panel:

```
{
  "body": "{\"user_prompt\": \"Who are you?\"}"
}
```

20. Press on **Test**. The function will execute, and error will occur:

Executing function: failed (logs)

Details

The area below shows the last 4 KB of the execution log.

```
{
  "errorMessage": "Unable to import module 'lambda_function': No module named 'openai'",
  "errorType": "Runtime.ImportModuleError",
  "requestId": "",
  "stackTrace": []
}
```

The error message "Unable to import module 'lambda_function': No module named 'openai'" indicates that the AWS Lambda environment cannot find the openai module. This happens because the openai package is not included in the deployment package. In the next task, we will resolve this.

Task 5: Create AWS Lambda layer for openai

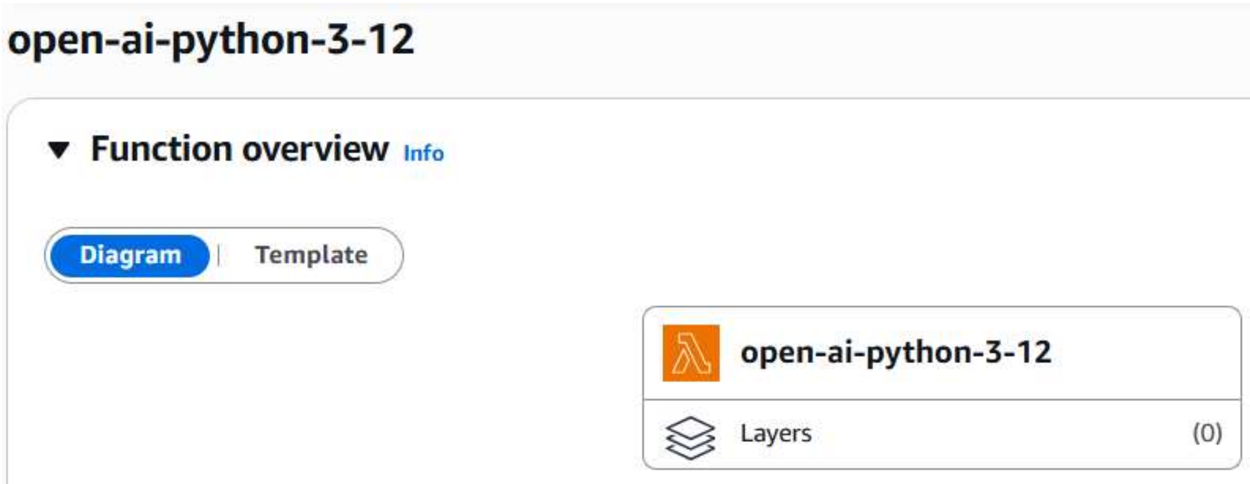
We will use a pre-created openai lambda layer zip file which supports **python 12**.

openai-2.24.0-layer-python-12.zip

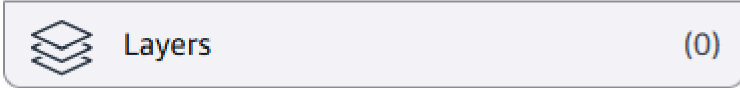
Task 6: Configure AWS Lambda layers

In this task you will solve the issue with missing dependencies using AWS Lambda layers.

1. In the Lambda **Function overview** pane, you will see that there is currently no layer.



2. In the Lambda Function overview pane, you will see that there are currently no Layers. Press on the Layers link to navigate to the Layers pane:

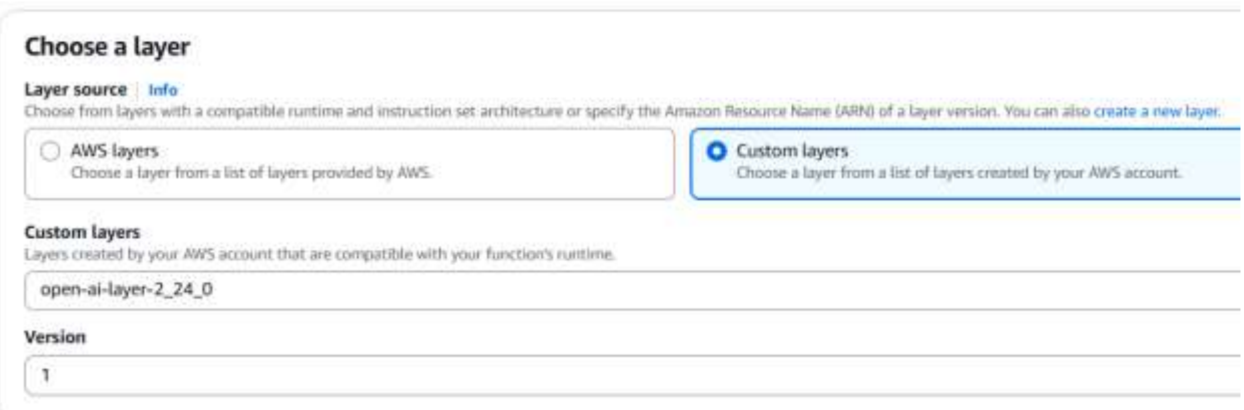


3. In the **Layers** pane, press on **Add a Layer.**

19. In the **Add layer** page, in the **Choose a layer** pane, press right click on the **create a new layer link,** and select Open link in new tab.

20. Navigate to the new tab, and In the **Create layer** window, set the **Name** as **openai-layer**, press on the **Upload** button, and select the **openai-lambda-layer.zip** file. Select the **Compatible runtimes** drop down, select Python 12, and press **Create**. Your layer has now successfully created.

21. Navigate back to the **Create layer** window and refresh the page (F5), in the **Choose a layer** pane, select the **Custom Layers** radio button. Then, open the **Custom Layers** drop down and select **openai-layer**, and in the Version drop down, select **1,** and press on **Add**.



Task 7: Re Verify that the Lambda function works

19. Choose **Test** tab, and update the next JSON in the **Event JSON** panel:

```
{
  "body": "{\"user_prompt\": \"Who are you?\"}"
}
```

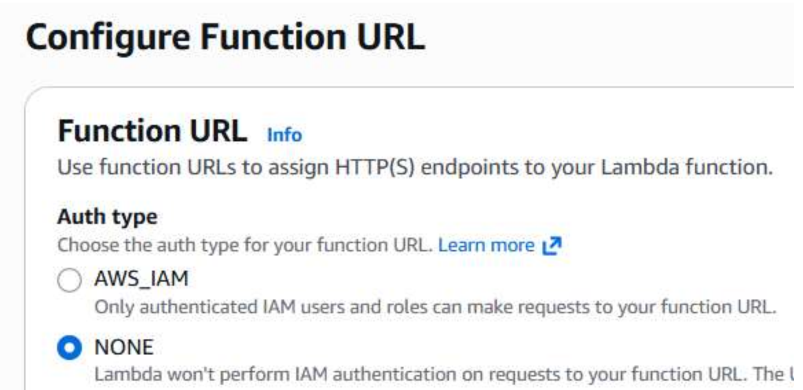
20. Press on **Test**. The function will execute and will finish successfully.

22. Press on the **Details** drop down and inspect the execution. You should see the response, Summary and Log output where you will also see the reply of the OpenAI API:
{
  "statusCode": 200,
  "body": "{\"ai_reply\": \" **I am an AI digital assistant designed to provide information and assist with various tasks. How can I help you today?**\"}"
}

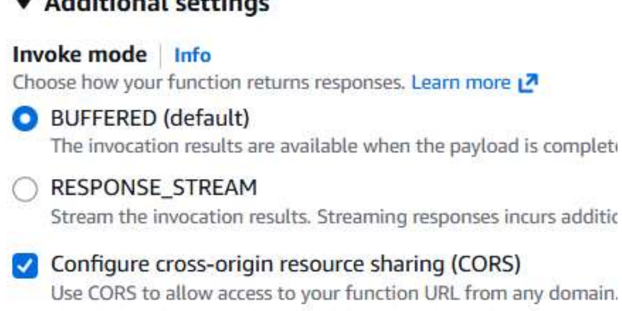
Task 8: Execute the Lambda function from a Web Browser web address URL.

15. Choose **Configuration** tab, **Function URL** (a dedicated HTTP(S) endpoint) option and press on **Create Function URL**.

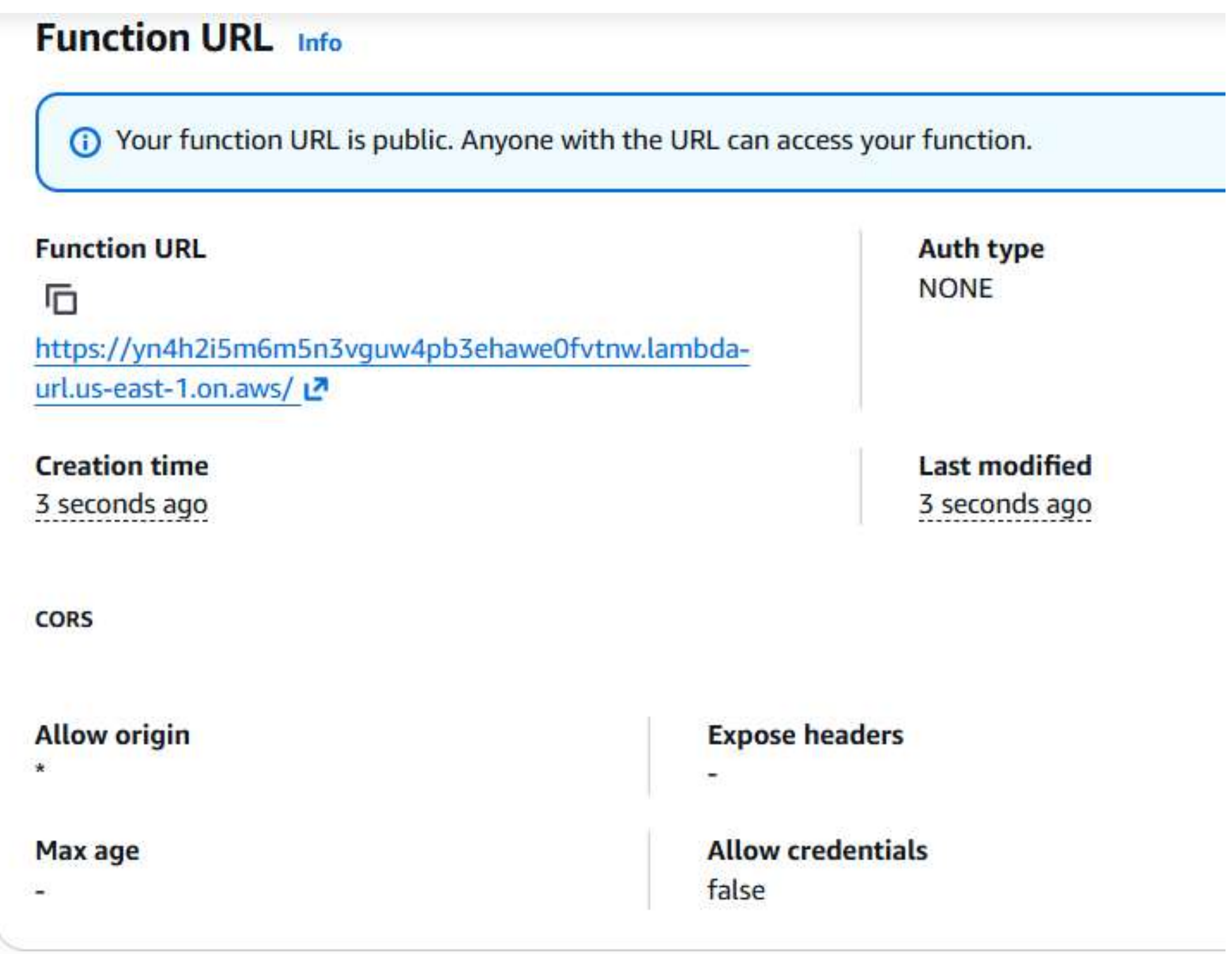
16. In the **Configure Function URL** section, under **Function URL** section select **None**



17. Open the **Additional settings** section and select **Configure cross-origin resource sharing (CORS)**



18. Press **Save**. Your Function URL should now be created.



19. Copy your function URL, and from your preferred web browser, execute it. You should probably get an error of missing user_prompt parameter.

    Note: You might get a different error depending on your function error handling.

    Pretty-print ☐

    ```
    {"error": "user_prompt is required"}
    ```

    Because our Lambda function supports POST request, we cannot execute it by copying and pasting the URL into a browser's address bar like with a GET request. POST requests require a request body, which cannot be included in a URL.

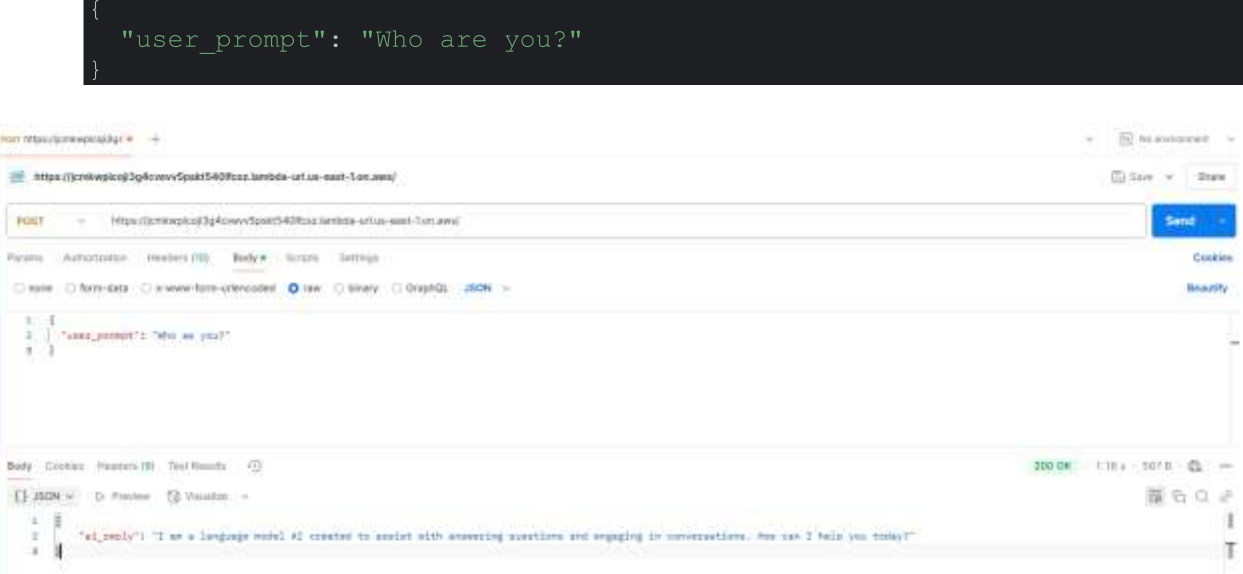
Task 9: Execute the Lambda function from postman.

1. Open **postman.com** and select to **Send an API request**.

   Get started

2. Set the request type to POST.
3. Enter the URL of your deployed Lambda function endpoint.
4. Set the headers:
   Key: Content-Type
   Value: application/json
5. Set the body of the request to raw JSON and include the following content:

```
{
  "user_prompt": "Who are you?"
}
```



Task 10: Execute the Lambda function from an HTML page.

1. Create a basic HTML page that will execute your lambda function. HTML page will implement a basic chat window. Make sure to execute POST method when calling your Lambda function.

   You can use the next prompt:

   > *create a basic chat HTML that will call AWS Lambda function.*
   >
   > *1. It will use a lambda function in the next URL:*
   >
   > *https://yn4h2i5m6m5n3vguw4pb3ehawe0fvtnw.lambda-url.us-east-1.on.aws/*
   >
   > *2. the lambda function will get the user prompt in a parameter called: user_prompt*
   >
   > *3. the Lambda function support POST request only*

   Alternatively, use the following code. Make sure to replace the <AWS Lambda Function URL> to your own AWS Lambda Function URL.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>MTA Chat</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            flex-direction: column;
            align-items: center;
            justify-content: center;
            height: 100vh;
            margin: 0;
            background-color: #f4f4f4;
        }
        .chat-container {
            width: 50%;
            max-width: 600px;
            background: white;
            padding: 20px;
```

```
            border-radius: 10px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }
        .chat-box {
            height: 300px;
            overflow-y: auto;
            border-bottom: 1px solid #ccc;
            margin-bottom: 10px;
            padding: 10px;
        }
        .input-container {
            display: flex;
        }
        input {
            flex: 1;
            padding: 10px;
            border: 1px solid #ccc;
            border-radius: 5px;
            margin-right: 10px;
        }
        button {
            padding: 10px 15px;
            border: none;
            background: blue;
            color: white;
            border-radius: 5px;
            cursor: pointer;
        }
    </style>
</head>
<body>
    <div class="chat-container">
        <h2>MTA Chat</h2>
        <div class="chat-box" id="chat-box"></div>
        <div class="input-container">
            <input type="text" id="user-input" placeholder="Type your
message..." autofocus>
            <button id="send-btn">Send</button>
        </div>
    </div>

    <script>
        document.getElementById("user-input").addEventListener("keypress",
function(event) {
            if (event.key === "Enter") {
                document.getElementById("send-btn").click();
            }
        });

document.getElementById("send-btn").addEventListener("click", async
function() {
    const userInput = document.getElementById("user-input").value.trim();
    if (!userInput) return;

    const chatBox = document.getElementById("chat-box");
    chatBox.innerHTML += `<div><strong>You:</strong> ${userInput}</div>`;
    document.getElementById("user-input").value = "";
```

```
    try {
        const response = await fetch("<AWS Lambda Function URL>", {
            method: "POST",
            headers: {
                "Content-Type": "application/json"
            },
            body: JSON.stringify({ user_prompt: userInput })
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        chatBox.innerHTML += `<div><strong>AI:</strong>
${data.ai_reply}</div>`;
        chatBox.scrollTop = chatBox.scrollHeight;
    } catch (error) {
        console.error("Fetch error:", error);
        chatBox.innerHTML += `<div style="color:
red;"><strong>Error:</strong> ${error.message}</div>`;
    }
});
    </script>
</body>
</html>
```

2. Open the HTML page, add the parameters values and execute it. You should get an error: **ChatGPT: Unable to connect to the server**.

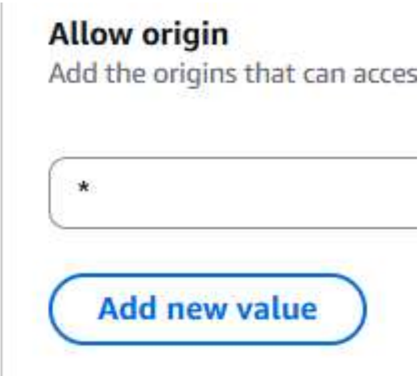
```
You: who are you?
ChatGPT: Unable to connect to the server.
```

This is due to CORS Policy. Our Lambda function is not supporting to allow cross-origin requests. When you call a Lambda function directly from a web browser by entering the URL, it works because the browser does not enforce Cross-Origin Resource Sharing (CORS) policies for direct URL access. However, when you call the Lambda function from an HTML page using JavaScript (e.g., fetch), the browser enforces CORS policies to prevent security issues. Let's fix this.

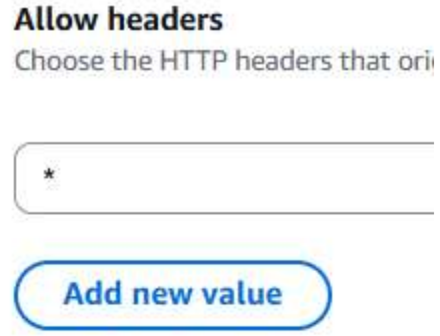
3. Add support to CORS headers in the Lambda function from Function URL Configuration.

    - In the **AWS Lambda function** window, Choose **Configuration** tab, **Function URL** and **Edit**.

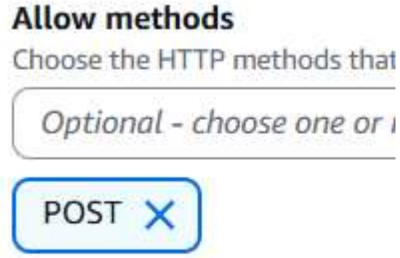
    - In the **Additional settings** section:
        - Make sure that **Allow origin** has the value of *****.



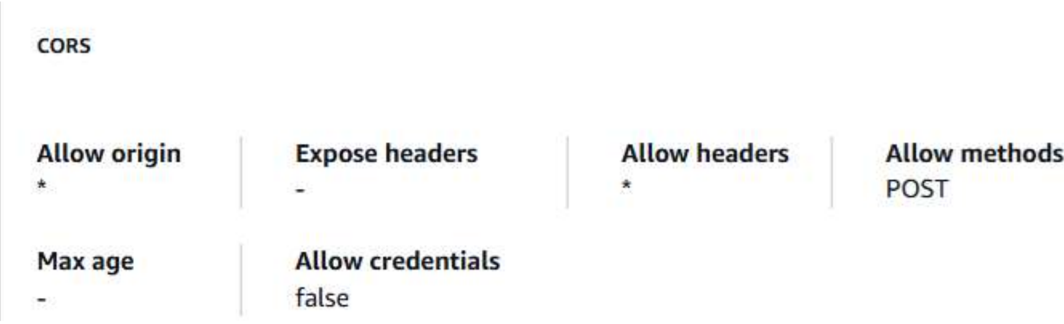
        - Set the Allow headers to * by pressing **Add new value**.



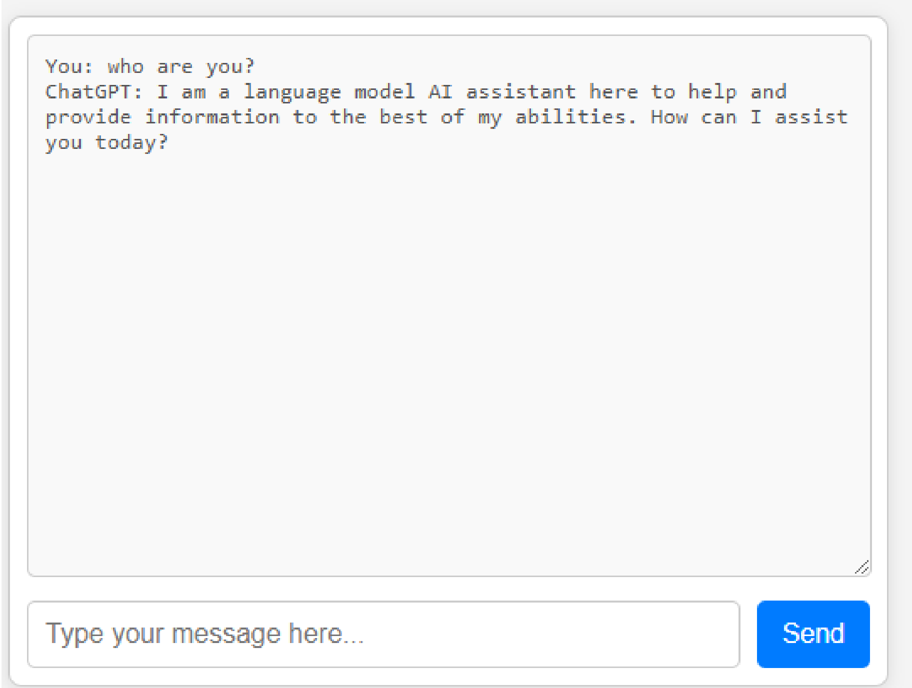
- Open the **Allow methods** drop down, and select **POST**



- Press **Save**. Your Function URL should now be updated.



4. Re Open the HTML page and execute it. IT should now work:



- **Note:** Alternative option for core support, instead of Function URL configuration is to change the lambda code to support it.
    - Step 1, set headers dictionary:

```
# CORS headers
headers = {
    'Access-Control-Allow-Origin': '*',
    'Access-Control-Allow-Methods': 'OPTIONS,POST',
    'Access-Control-Allow-Headers': 'Content-Type'
}
```

    - Step 2, if HTTP method is OPTIONS, return 200:

```
if event.get("requestContext", {}).get("http", {}).get("method") == "OPTIONS":
        return {
            "statusCode": 200,
            "headers": headers,
            "body": ""
        }
```

    - Step 3, add `'headers': headers` to all returns, example:

```
return {
    "statusCode": 200,
    'headers': headers,
    "body": json.dumps({"ai_reply": ai_reply})
}
```

Task 11: Create Python chat using lambda function

In this task, you will write a Python program that interacts with a lambda function to generate responses based on user input.

1. *Write a python script that will implement a chat using an AWS lambda function (Which calls open AI API) that support HTTP Method POST*.
   Alternatively, use the following code, make sure to replace <LAMBDA_FUNCTION_URL> with your lambda function URL:

```
import requests

lambda_url = "<LAMBDA_FUNCTION_URL>/"

conversation_history = []

while True:
    user_prompt = input("Enter your prompt (or 'exit' to quit): ")
    if user_prompt.lower() == 'exit':
        break

    conversation_history.append({"role": "user", "content": user_prompt})

    try:
        ##response = requests.get(lambda_url, params={"user_prompt": user_prompt})
        response = requests.post(lambda_url, json={"user_prompt":
user_prompt})
        response.raise_for_status()
        data = response.json()
        ai_reply = data["ai_reply"]
    except requests.exceptions.RequestException as e:
        print(f"Error: {e}")
        continue

    print("AI:", ai_reply)
    conversation_history.append({"role": "assistant", "content": ai_reply})
```

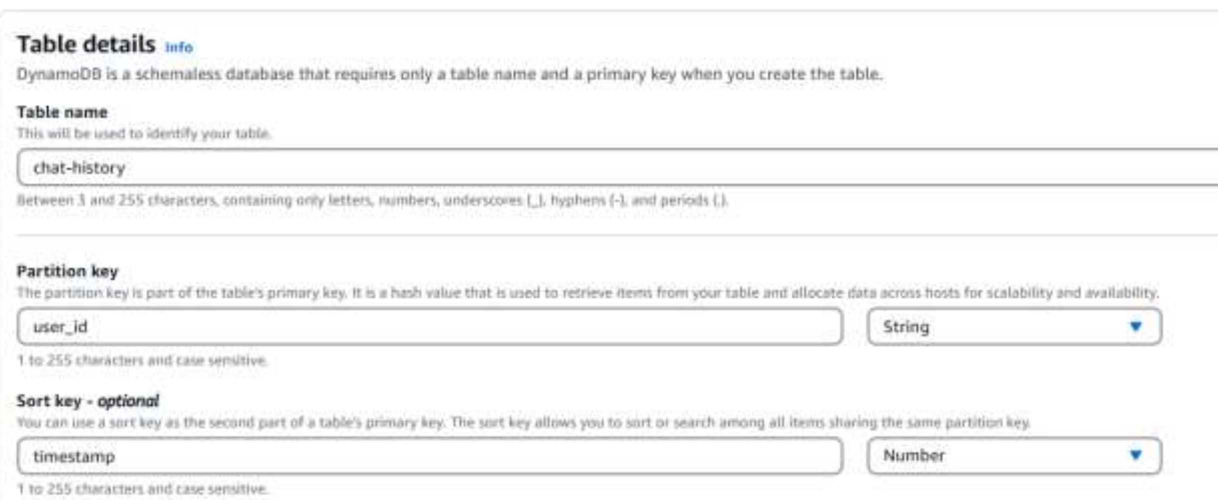
2. Run the code, verify it's working by asking any question:

   Enter your prompt (or 'exit' to quit): hello
   AI: Hello! How can I assist you today?
   Enter your prompt (or 'exit' to quit):

Task 12: Enhance the Lambda function to support conversation history

At this point, the Lambda function is not supporting conversation history, so open-ai will not reply to the prompt based on session context. As Lambda function is stateless, we will now add DynamoDB table to keep the conversation history and enhance the Lambda function to read and write to the table per needs. In addition, we will enhance the HTML page to pass to the Lambda API user_id.

1. Create a DynamoDB Table.
    - In the AWS Console search box to the right of **Services**, search for and choose **DynamoDB** to open the **DynamoDB** console.
    - Choose **Create table**.
    - In the **Create table** screen, configure these settings:
        1. Table name: `chat-history:` A clear name representing stored chat messages.
        2. Partition key name: `user_id (String):` Groups messages by user.
        3. Sort key name: `timestamp (Number):` Ensures messages are ordered chronologically per user.



- Choose **Create table**.
- The table will now be created, this operation might take a few minutes.

Creating the chat-history table. It will be available for use shortly.

- Once the table is ready to use, the following message will appear:

2. *Enhance the Lambda function to support conversation_history using Dynamo DB:*
    - *In addition to* 'user_prompt', *it will also get* 'user_id'.
    - *It will create a resource to* 'dynamodb' to table: 'chat-history'.
    - *It will read the last 6 messages from a DynamoDB table and send it to open-ai API sorted by timestamp using the conversation_history*
    - *List.*
    - *I will then save the last prompt and reply in the table. Remember also to get from the body the user_id value which is needed for the table.*

Alternatively, use the following code, make sure to replace < OPEN_AI_API_KEY> with your lambda function URL:

```
import json
import time
import boto3
from openai import OpenAI

client = OpenAI(api_key="YOUR_OPENAI_KEY")

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('chat-history')


def lambda_handler(event, context):

    # Parse body
    try:
        body = json.loads(event.get("body", "{}"))
    except Exception:
        return {
            "statusCode": 400,
            "body": json.dumps({"error": "Invalid JSON"})
        }

    user_prompt = body.get("user_prompt")
    user_id = body.get("user_id")

    if not user_prompt or not user_id:
        return {
            "statusCode": 400,
            "body": json.dumps({"error": "Missing user_prompt or
```

```
user_id"})
        }

    try:

        # 1 Get last 6 messages from DynamoDB
        response = table.query(

KeyConditionExpression=boto3.dynamodb.conditions.Key('user_id').eq(user
_id),
            ScanIndexForward=False,
            Limit=6
        )

        items = response.get("Items", [])

        # 2 Sort messages by timestamp
        items_sorted = sorted(items, key=lambda x: x["timestamp"])

        # 3 Build conversation history
        conversation_history = []

        for item in items_sorted:
            conversation_history.append({
                "role": item["role"],
                "content": item["content"]
            })

        # Add current user prompt
        conversation_history.append({
            "role": "user",
            "content": user_prompt
        })

        # 4 Send to OpenAI
        ai_response = client.responses.create(
            model="gpt-4.1-nano",
            input=conversation_history
        )

        ai_reply = ai_response.output_text

        timestamp = int(time.time())

        # 5 Save user prompt
        table.put_item(
            Item={
                "user_id": user_id,
                "timestamp": timestamp,
                "role": "user",
                "content": user_prompt
            }
        )

        # 6 Save AI reply
```

```
        table.put_item(
            Item={
                "user_id": user_id,
                "timestamp": timestamp + 1,
                "role": "assistant",
                "content": ai_reply
            }
        )

        return {
            "statusCode": 200,
            "body": json.dumps({
                "response": ai_reply
            })
        }

    except Exception as e:
        return {
            "statusCode": 500,
            "body": json.dumps({"error": str(e)})
        }
```

3. Choose **Test** tab, and use the next 4 prompts, one by one, to test your code:

```
{
  "body": "{\"user_prompt\": \"How much is 1+1?\", \"user_id\": \"user1\"}"
}
```

```
{
  "body": "{\"user_prompt\": \"How much is 2+2?\", \"user_id\": \"user1\"}"
}
```

```
{
  "body": "{\"user_prompt\": \"How much is 3+3?\", \"user_id\": \"user1\"}"
}
```

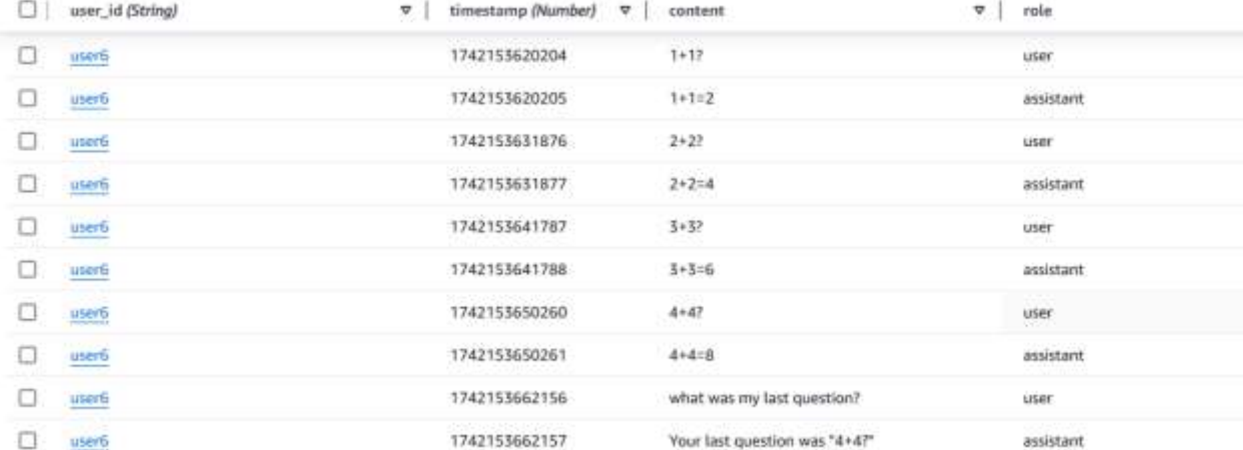
```
{
  "body": "{\"user_prompt\": \"How much is 4+4?\", \"user_id\": \"user1\"}"
}
```

```
{
  "body": "{\"user_prompt\": \"What did I just asked you?\", \"user_id\":
\"user1\"}"
}
```

4. Verify the last reply you got:

```
{
  "statusCode": 200,
  "body": "{\"response\": \"You asked, \\\"How much is 4+4?\\\"\"}" }
```

5. Verify the content of **DynamoDB** Table:



user_id (String)	timestamp (Number)	content	role
user6	1742153620204	1+1?	user
user6	1742153620205	1+1=2	assistant
user6	1742153631876	2+2?	user
user6	1742153631877	2+2=4	assistant
user6	1742153641787	3+3?	user
user6	1742153641788	3+3=6	assistant
user6	1742153650260	4+4?	user
user6	1742153650261	4+4=8	assistant
user6	1742153662156	what was my last question?	user
user6	1742153662157	Your last question was "4+4?"	assistant

6. *Enhance the HTML chat page to send also the user_id. You can select one of the next 3 options:*
    - *Hard code the user_id value*
    - *Generate a random user_id*
    - *Ask the user for his name and use it as a user_id*

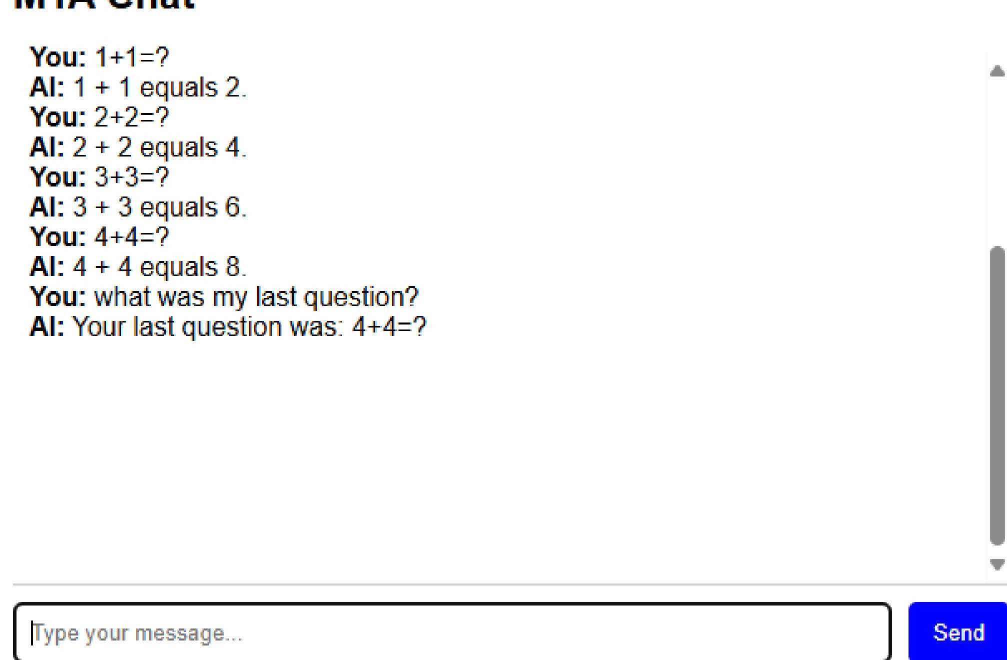
   Alternatively, use the following code which generates a random user_id. Make sure to replace the <AWS Lambda Function URL> to your own AWS Lambda Function URL.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>MTA Chat</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            display: flex;
            flex-direction: column;
            align-items: center;
            justify-content: center;
            height: 100vh;
            margin: 0;
            background-color: #f4f4f4;
        }
        .chat-container {
            width: 50%;
            max-width: 600px;
            background: white;
```

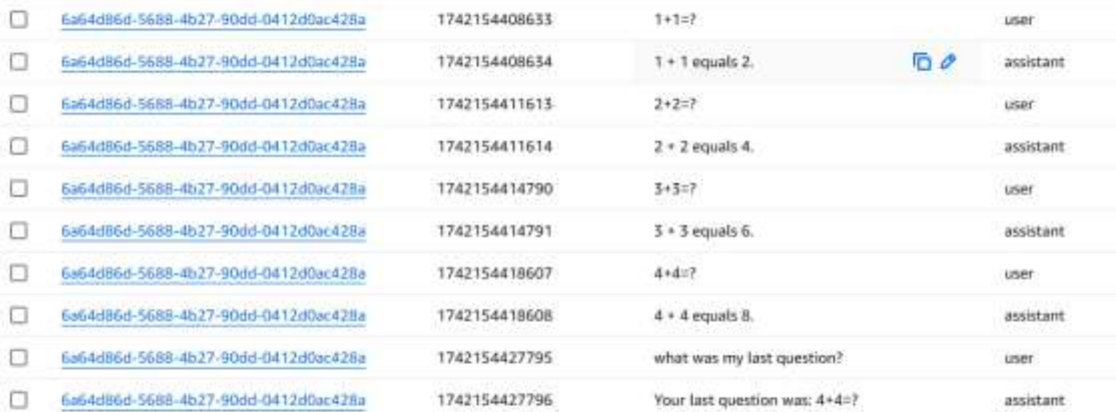
```
            padding: 20px;
            border-radius: 10px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }
        .chat-box {
            height: 300px;
            overflow-y: auto;
            border-bottom: 1px solid #ccc;
            margin-bottom: 10px;
            padding: 10px;
        }
        .input-container {
            display: flex;
        }
        input {
            flex: 1;
            padding: 10px;
            border: 1px solid #ccc;
            border-radius: 5px;
            margin-right: 10px;
        }
        button {
            padding: 10px 15px;
            border: none;
            background: blue;
            color: white;
            border-radius: 5px;
            cursor: pointer;
        }
    </style>
</head>
<body>
    <div class="chat-container">
        <h2>MTA Chat</h2>
        <div class="chat-box" id="chat-box"></div>
        <div class="input-container">
            <input type="text" id="user-input" placeholder="Type your
message..." autofocus>
            <button id="send-btn">Send</button>
        </div>
    </div>

    <script>

        // Generate or retrieve user_id from localStorage
        let userId = localStorage.getItem("user_id") ||
crypto.randomUUID();
        localStorage.setItem("user_id", userId);

        document.getElementById("user-
input").addEventListener("keypress", function(event) {
            if (event.key === "Enter") {
                document.getElementById("send-btn").click();
            }
        });

document.getElementById("send-btn").addEventListener("click", async
```

```
function() {
    const userInput = document.getElementById("user-
input").value.trim();
    if (!userInput) return;

    const chatBox = document.getElementById("chat-box");
    chatBox.innerHTML += `<div><strong>You:</strong>
${userInput}</div>`;
    document.getElementById("user-input").value = "";

    try {
        const response = await fetch("<AWS Lambda Function URL>", {
            method: "POST",
            headers: {
                "Content-Type": "application/json"
            },
            body: JSON.stringify({
                        user_id: userId,  // Added user_id
                        user_prompt: userInput
                    })
        });

        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }

        const data = await response.json();
        chatBox.innerHTML += `<div><strong>AI:</strong>
${data.ai_reply}</div>`;
        chatBox.scrollTop = chatBox.scrollHeight;
    } catch (error) {
        console.error("Fetch error:", error);
        chatBox.innerHTML += `<div style="color:
red;"><strong>Error:</strong> ${error.message}</div>`;
    }
});
    </script>
</body>
</html>
```

7. Ask the 4 questions as noted above and verify the last one is correct:



8. Similarly, review your table:



☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154408633	1+1=?	user
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154408634	1 + 1 equals 2.	assistant
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154411613	2+2=?	user
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154411614	2 + 2 equals 4.	assistant
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154414790	3+3=?	user
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154414791	3 + 3 equals 6.	assistant
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154418607	4+4=?	user
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154418608	4 + 4 equals 8.	assistant
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154427795	what was my last question?	user
☐	6a64d86d-5688-4b27-90dd-0412d0ac428a	1742154427796	Your last question was: 4+4=?	assistant

Activity complete

Congratulations! You have completed the activity.