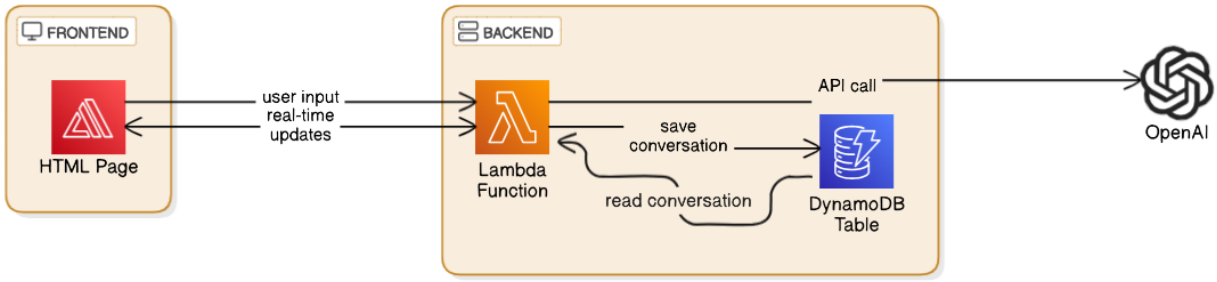
# Project: ChatBot development using OPEN AI API & AWS Services - AWS Lambda, AWS Amplify & DynamoDB

## 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 `chat.completions.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.*

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

client = OpenAI(
    api_key="<OPEN_AI_API_KEY>")
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})

    chat_completion = client.chat.completions.create(
        messages=conversation_history,
        model="gpt-3.5-turbo",
        max_tokens=100  # Limit the number of tokens in the reply
    )

    ai_reply = chat_completion.choices[0].message.content
    print("AI:", ai_reply)

    conversation_history.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

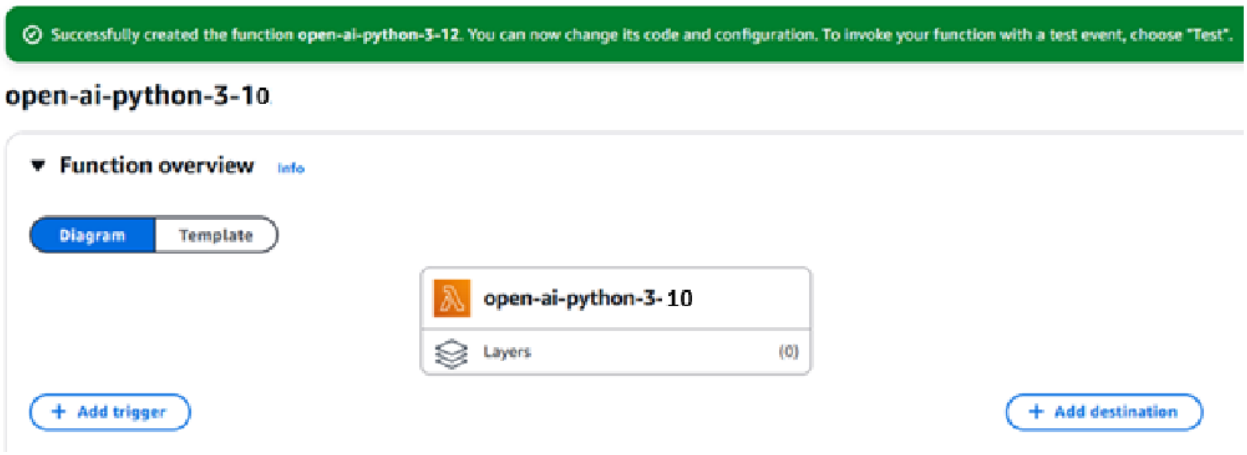
In this task, we will write a lambda function in python, based on the code we wrote in task 1.

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 `chat.completions.create` 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.*

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="<OPEN_AI_API_KEY>")

    print("event = ", event)

    try:
        body = json.loads(event.get('body', '{}'))
        print("body = ", body)
    except json.JSONDecodeError:
        return {"statusCode": 400, "body": json.dumps({"error":
"Invalid JSON"})}

    conversation_history = []
    user_prompt = body.get('user_prompt')

    if not user_prompt:
        return {"statusCode": 400, "body": json.dumps({"error":
"user_prompt is required"})}

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

    chat_completion = client.chat.completions.create(
        messages=conversation_history,
        model="gpt-3.5-turbo",
        max_tokens=100
    )

    ai_reply = chat_completion.choices[0].message.content

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

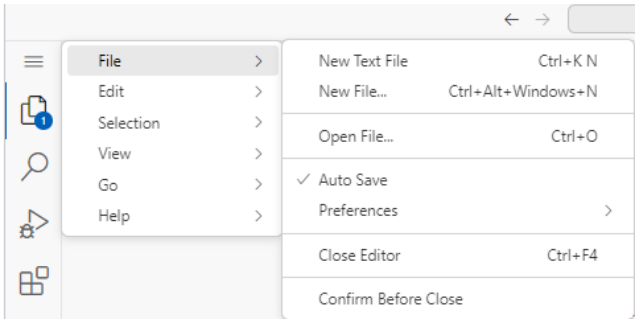
3. In **AWS Management Console**, In the search box to the right of **Services**, search for and choose **Lambda** to open the AWS Lambda console.
4. Choose **Create a function**.
5. In the **Create function** screen, configure these settings:
    - Choose **Author from scratch**
    - Function name: open-ai-python-3-10
    - Runtime: **Python 3.10**
    - 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.

# 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:


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

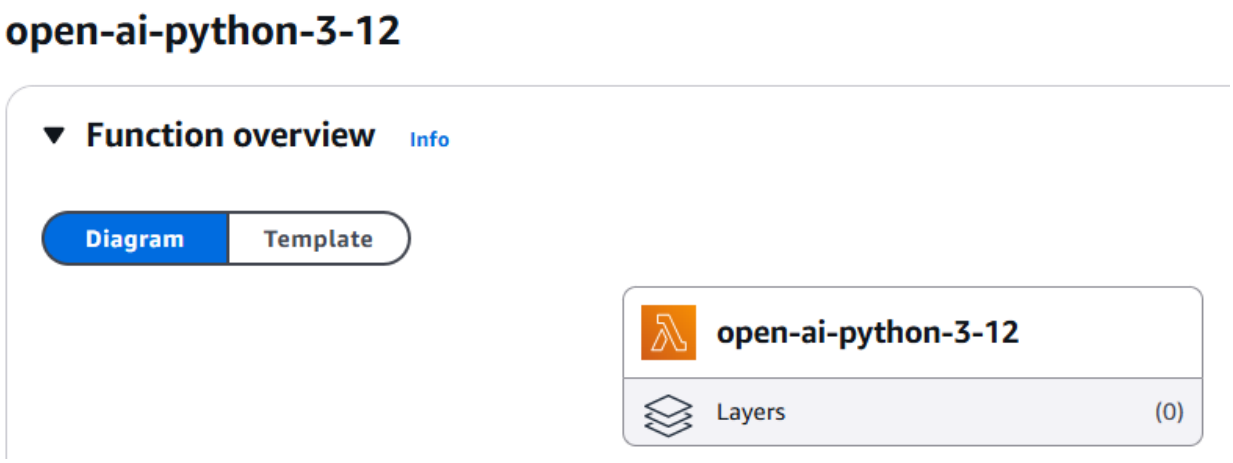
We will use a pre created openai lambda layer zip file which support python 10.

openai-lambda-layer.zip

# Task 6: Configure AWS Lambda layers

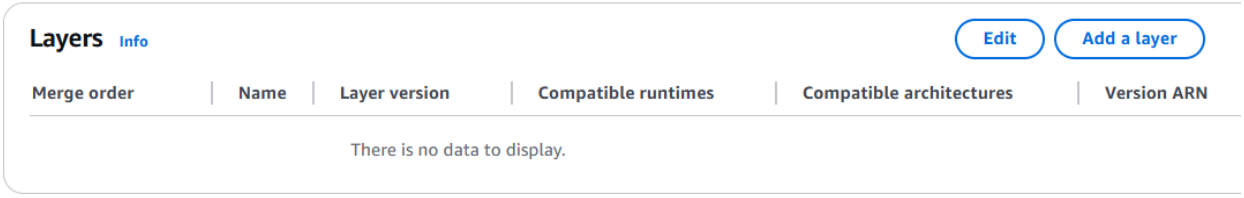
In this task you will solve the issue with missing dependencies using AWS Lambda layers.

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

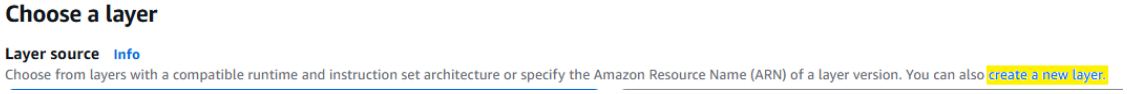



20. In the Lambda Function overview pane, you will see that there are currently no Layers. Press on the Layers link to navigate to the Layers pane:

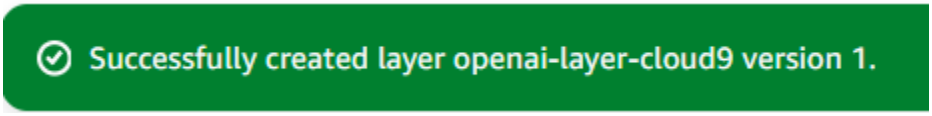

21. In the **Layers** pane, press on **Add a Layer.**



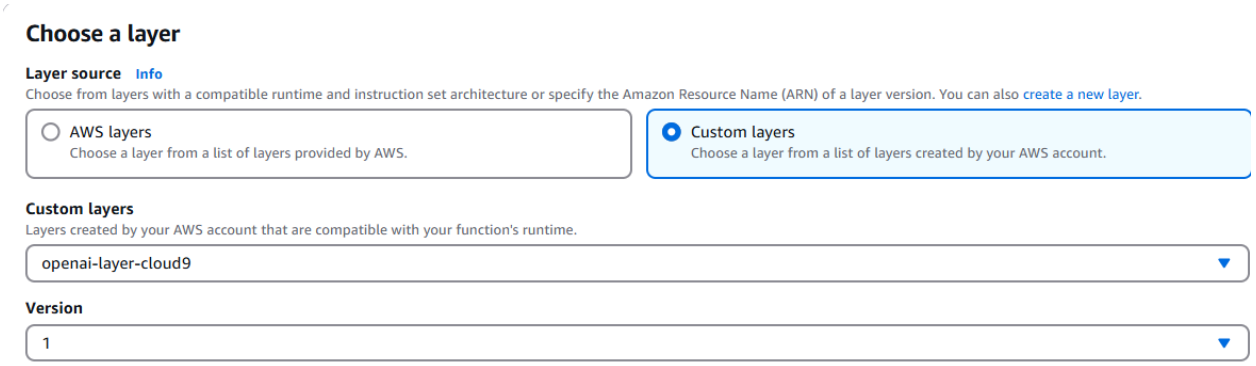

19. In the **Add layer** page, in the **Choose a layer** pane, press right click on the **create a new layer link,** and select Open link in new tab.



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 10, and press **Create**. Your layer has now successfully created.




21. Navigate back to the **Create layer** window, 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**.



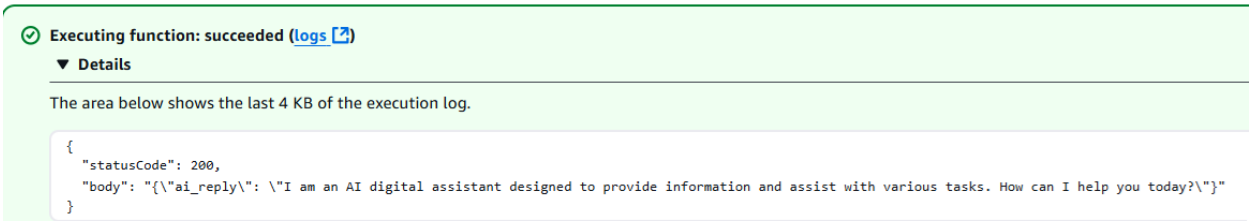
# Task 7: Re Verify that the Lambda function works

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

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

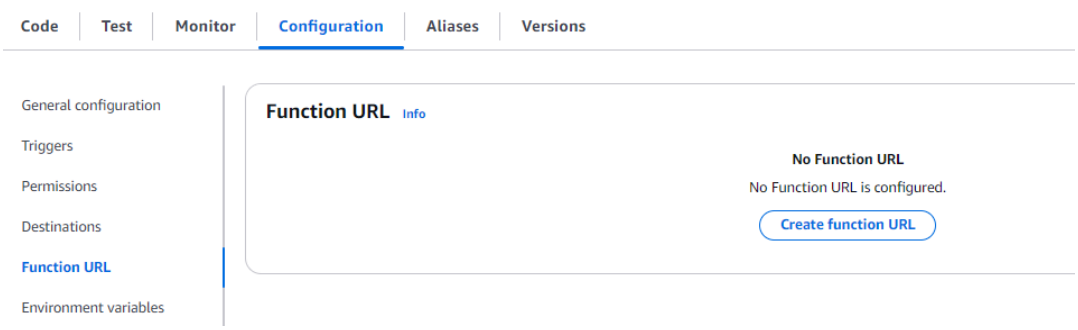
20. Press on **Test**. The function will execute and will finish successfully.

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?**\"}"
    }

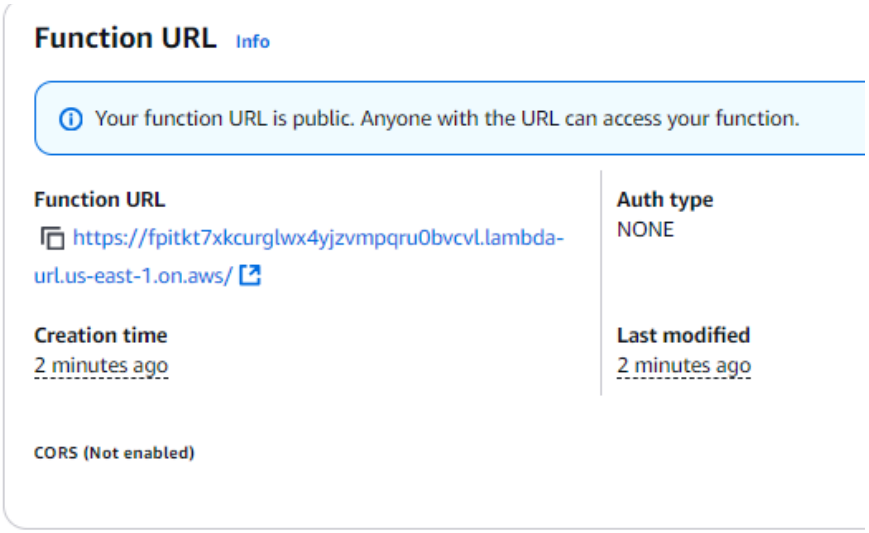



# Task 8: Execute the Lambda function from a Web Browser web address URL.

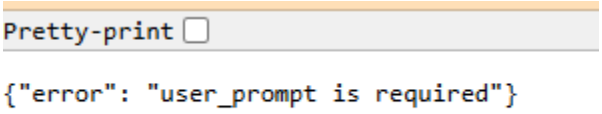
15. Choose **Configuration** tab, **Function URL** (a dedicated HTTP(S) endpoint) option and press on **Create Function URL**.




16. In the **Configure Function URL** section, select **None** and press **Save**. Your Function URL should now be created.




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



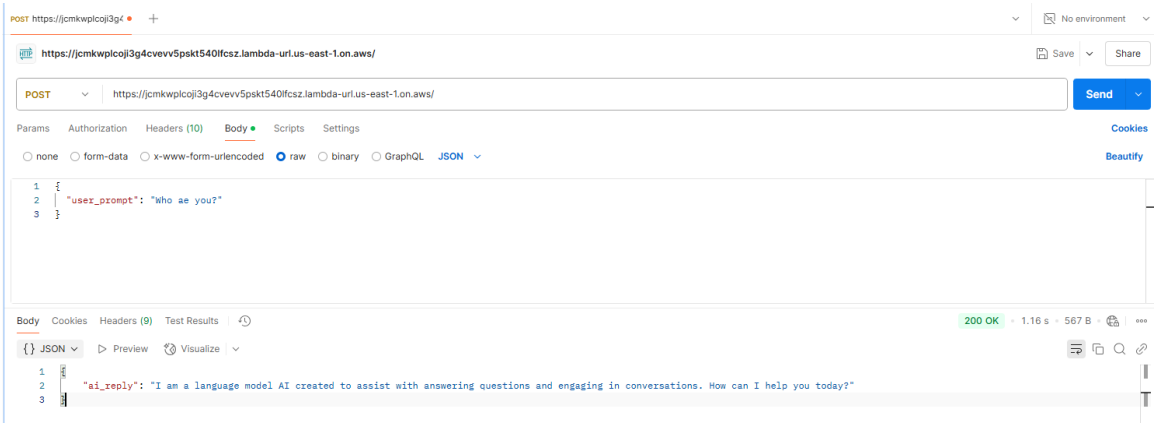

Because our Lambda function supports POST request, we cannot execute it by copying and pasting the URL into a browser's address bar like with a GET request. POST requests require a request body, which cannot be included in a URL.

# Task 9: Execute the Lambda function from postman.

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


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.

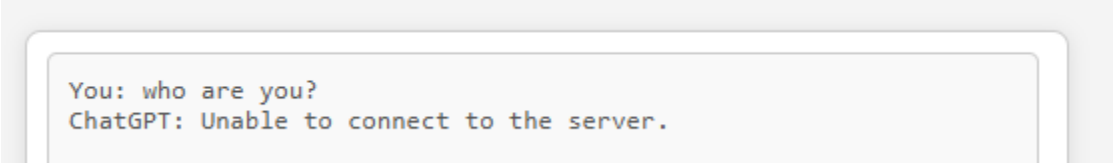
1. *Create a basic chat HTML page.*
   1. *It will use an AWS lambda function URL of HTTP POST Method.*
   2. *The Lambda response will be in a parameter called: ai_reply*
   3. *The page title will be: “”” MTA Chat “””*
   4. *In the page, when I press Enter, I want it to press the SEND button*
   5. *create chat.html from this content so I can download it.*

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**.



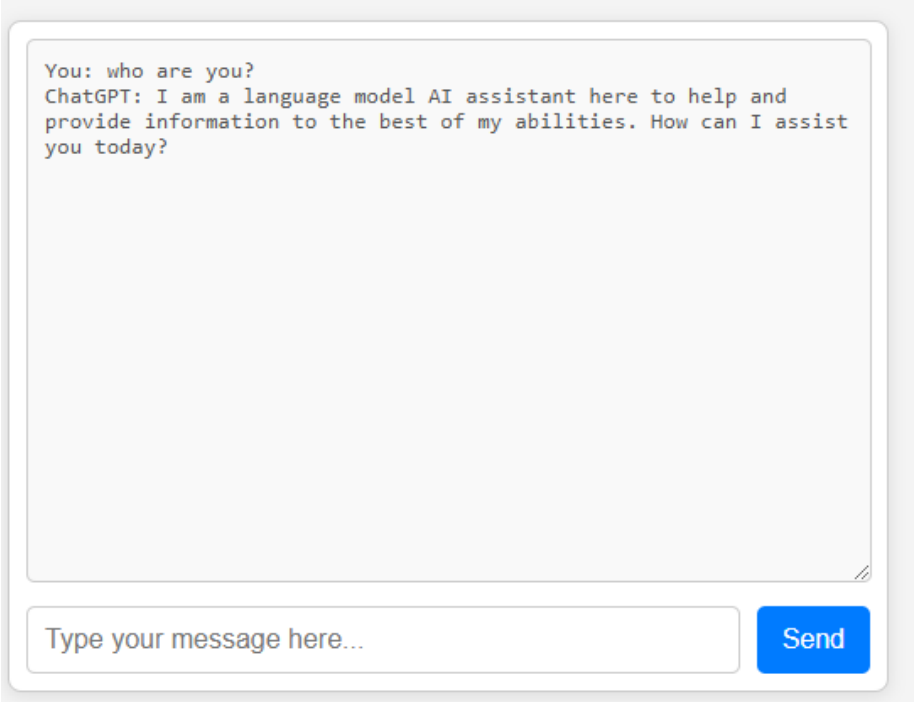

This is due to CORS Policy. Our Lambda function is not supporting to allow cross-origin requests. When you call a Lambda function directly from a web browser by entering the URL, it works because the browser does not enforce Cross-Origin Resource Sharing (CORS) policies for direct URL access. However, 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.*
4. *In addition, the Lambda function should support OPTIONS method so it can be called from HTML page.*
5. Alternatively, use the following code, make sure to replace <OPEN_AI_API_KEY> with your own key.
   Note: The new code is highlighted.

```
import json
from openai import OpenAI

def lambda_handler(event, context):
    client = OpenAI(
        api_key="<OPEN_AI_API_KEY>")

    print("event = ", event)

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

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

    try:
```

```
        body = json.loads(event.get('body', '{}'))
        print("body = ", body)
    except json.JSONDecodeError:
        return {"statusCode": 400, 'headers': headers, "body":
json.dumps({"error": "Invalid JSON"})}

    conversation_history = []
    user_prompt = body.get('user_prompt')

    if not user_prompt:
        return {"statusCode": 400, 'headers': headers, "body":
json.dumps({"error": "user_prompt is required"})}

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

    chat_completion = client.chat.completions.create(
        messages=conversation_history,
        model="gpt-3.5-turbo",
        max_tokens=100
    )

    ai_reply = chat_completion.choices[0].message.content

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

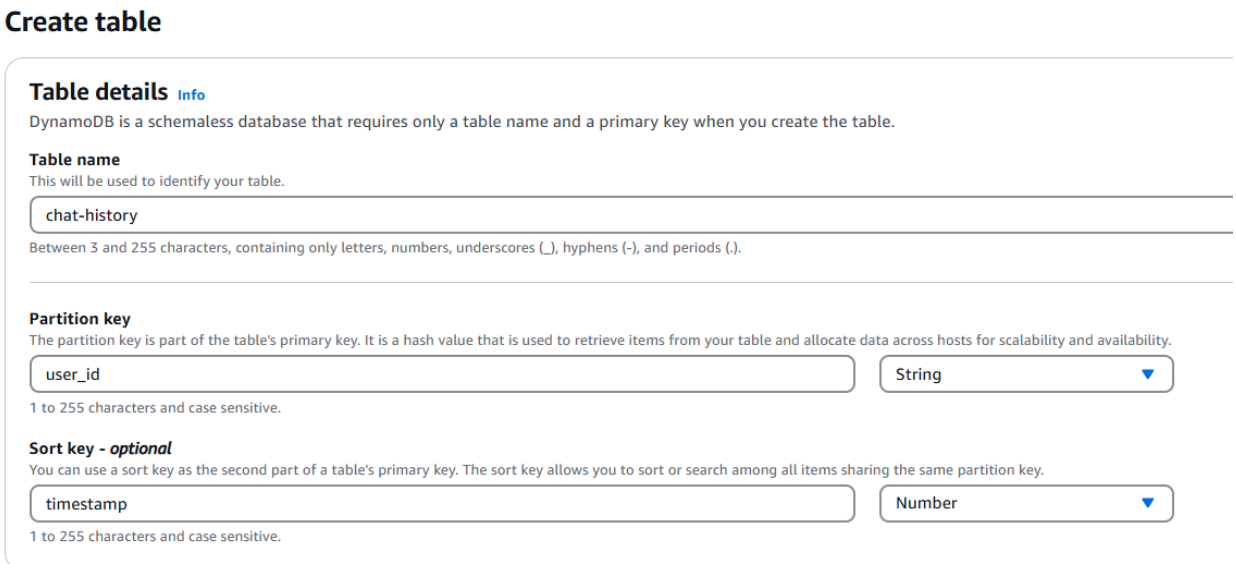
6. Re Open the HTML page and execute it. IT should now work:



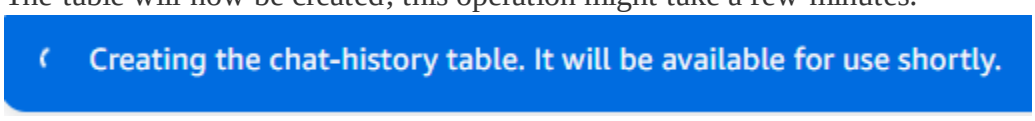
# Task 11: Enhance the Lambda function to support conversation history

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.

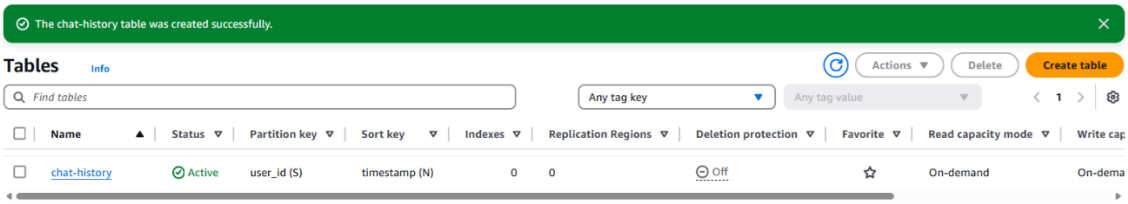
1. Create a DynamoDB Table.
    - In the AWS Console search box to the right of **Services**, search for and choose **DynamoDB** to open the **DynamoDB** console.
    - Choose **Create table**.
    - In the **Create table** screen, configure these settings:
        1. Table name: `chat-history:` A clear 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.



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




2. *Enhance the Lambda function to read the last 6 messages from the database table and send it to open-ai API sorted by timestamp, then to 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
from openai import OpenAI
import boto3
from boto3.dynamodb.conditions import Key
import time

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('chat-history')

def lambda_handler(event, context):
    client = OpenAI(
        api_key="<OPEN_AI_API_KEY>")

    print("event = ", event)

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

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

    try:
        body = json.loads(event.get('body', '{}'))
        print("body = ", body)
    except json.JSONDecodeError:
```

```
        return {"statusCode": 400, 'headers': headers, "body":
json.dumps({"error": "Invalid JSON"})}

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

    if not user_prompt:
        return {"statusCode": 400, 'headers': headers, "body":
json.dumps({"error": "user_prompt is required"})}

    if not user_id:
        return {"statusCode": 400, 'headers': headers, "body":
json.dumps({"error": "user_id is required"})}

    # Retrieve the last 6 messages from DynamoDB
    response = table.query(
        KeyConditionExpression=Key('user_id').eq(user_id),
        ScanIndexForward=False,
        Limit=6
    )
    conversation_history = response.get('Items', [])
    print("conversation_history we got:",conversation_history)

    conversation_history = sorted(conversation_history, key=lambda x:
x['timestamp'])
    print("conversation_history after sort",conversation_history)

    print("remove unnecessary fields from conversation_history")
    # Remove unnecessary fields
    for message in conversation_history:
        del message['user_id']
        del message['timestamp']

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

    print("conversation_history after adding
prompt",conversation_history)


    chat_completion = client.chat.completions.create(
        messages=conversation_history,
        model="gpt-3.5-turbo",
        max_tokens=100
    )

    ai_reply = chat_completion.choices[0].message.content

    # Save only the last prompt and reply to DynamoDB
    timestamp = int(time.time() * 1000)
    table.put_item(
        Item={
            'user_id': user_id,
            'timestamp': timestamp,
            'role': 'user',
            'content': user_prompt
        }
```

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


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

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,
  "headers": {
    "Access-Control-Allow-Origin": "*",
    "Access-Control-Allow-Methods": "OPTIONS,POST",
    "Access-Control-Allow-Headers": "Content-Type"
  },
```

"body": "{\"ai_reply\": \"**The last question you asked me was \\\"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 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*

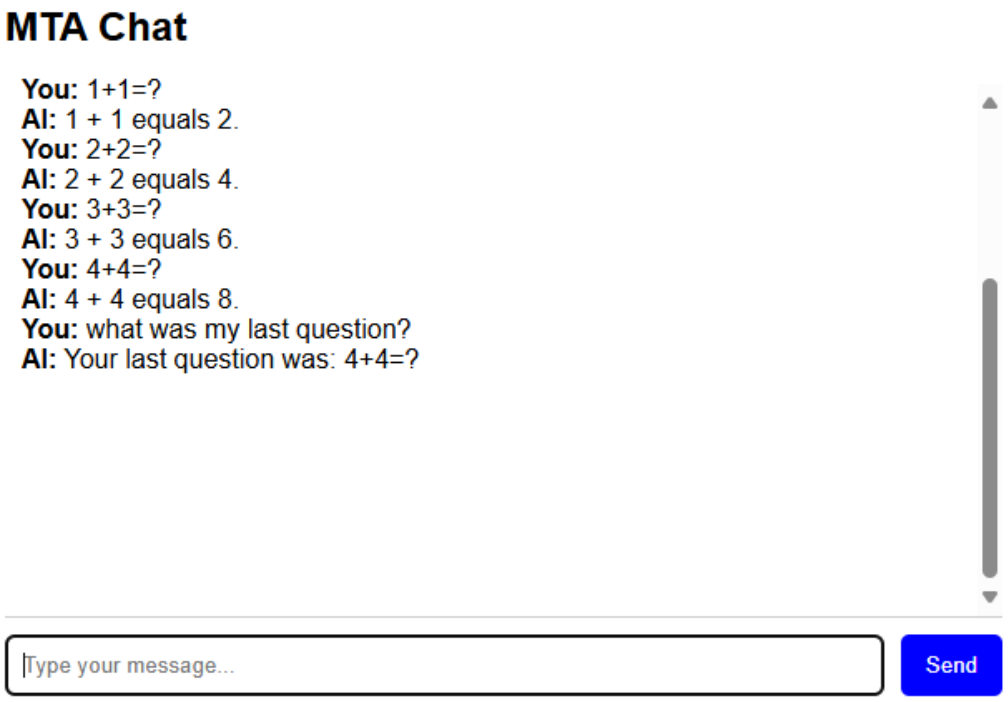
   Alternatively, use the following code which generates a random user_id. Make sure to replace the <AWS Lambda Function URL> to your own AWS Lambda Function URL.

```
<!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.