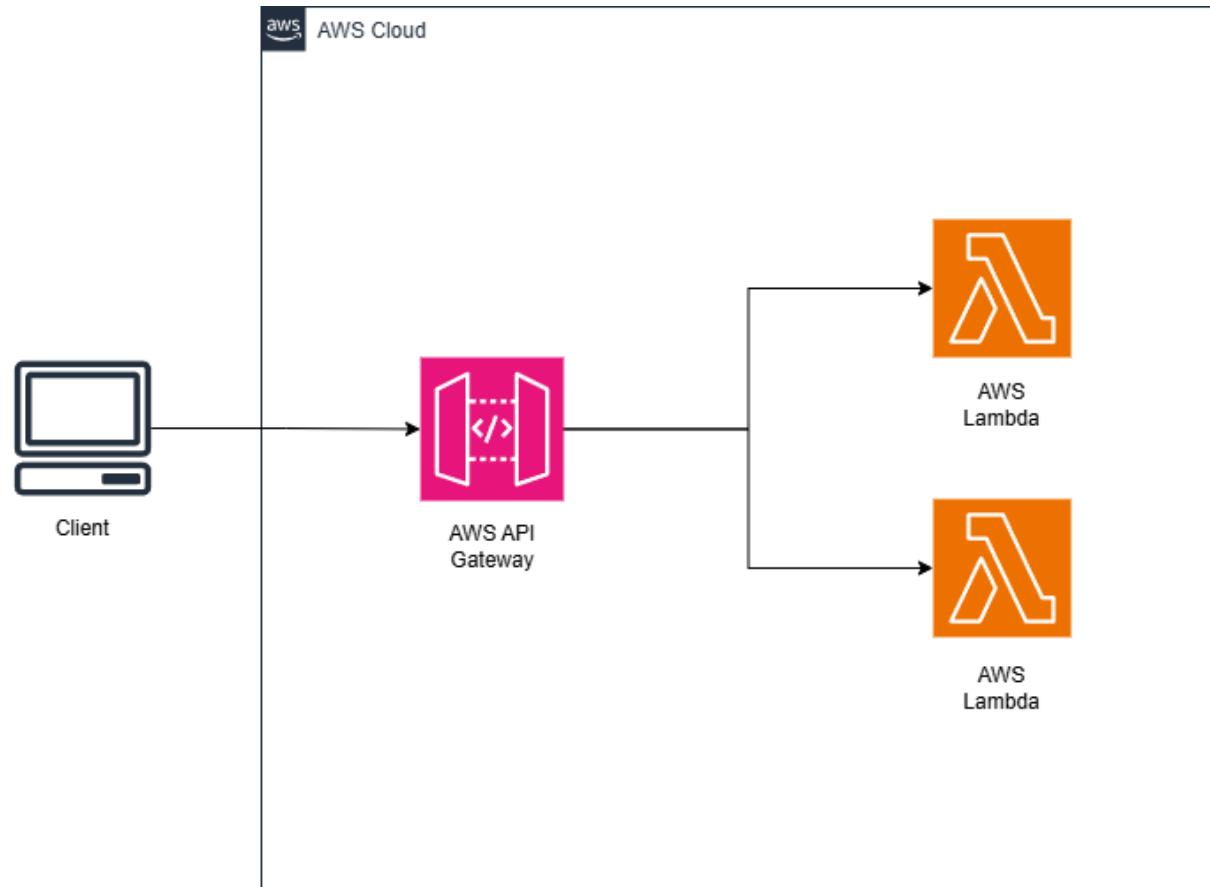


## Integrate AWS with Lambda and API Gateway and Monitoring

Architecture :



Service used :

- AWS Lambda
- AWS API Gateway

Walkthrough:

## PART 1 : Creating the Lambda and monitor the function logs

1. Open up the AWS Console and open the Lambda services, choose to create a new Function.

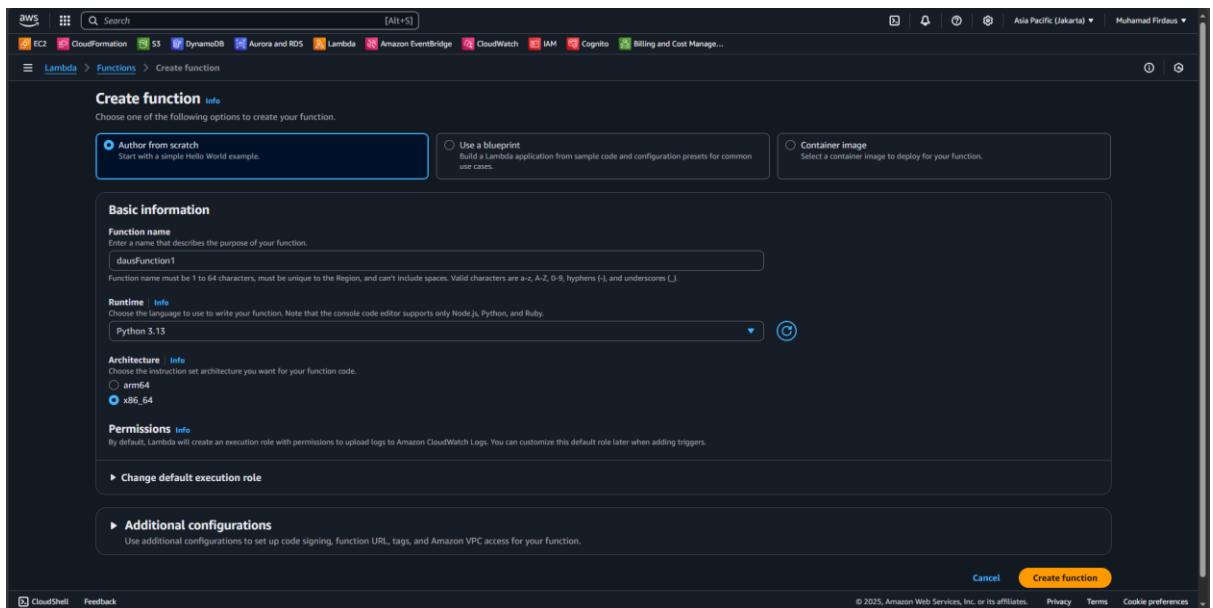
The screenshot shows the AWS Console interface with the Lambda service selected. The left sidebar lists various AWS services, and the main content area is dedicated to Lambda. It displays sections for 'Welcome to AWS' (with links to getting started and training), 'AWS Health' (showing 0 open issues and scheduled changes), and 'Cost and usage' (showing current month costs at \$0.00 and forecasted end-of-month costs at \$0.01). A large central area is titled 'AWS Lambda' with the sub-headline 'lets you run code without thinking about servers.' It includes a 'Get started' button and a code editor window titled 'How it works' showing Python sample code:

```
s = "def lambda_handler(event, context):\n2     print(event)\n3     return 'Hello from Lambda!'\n4"
```

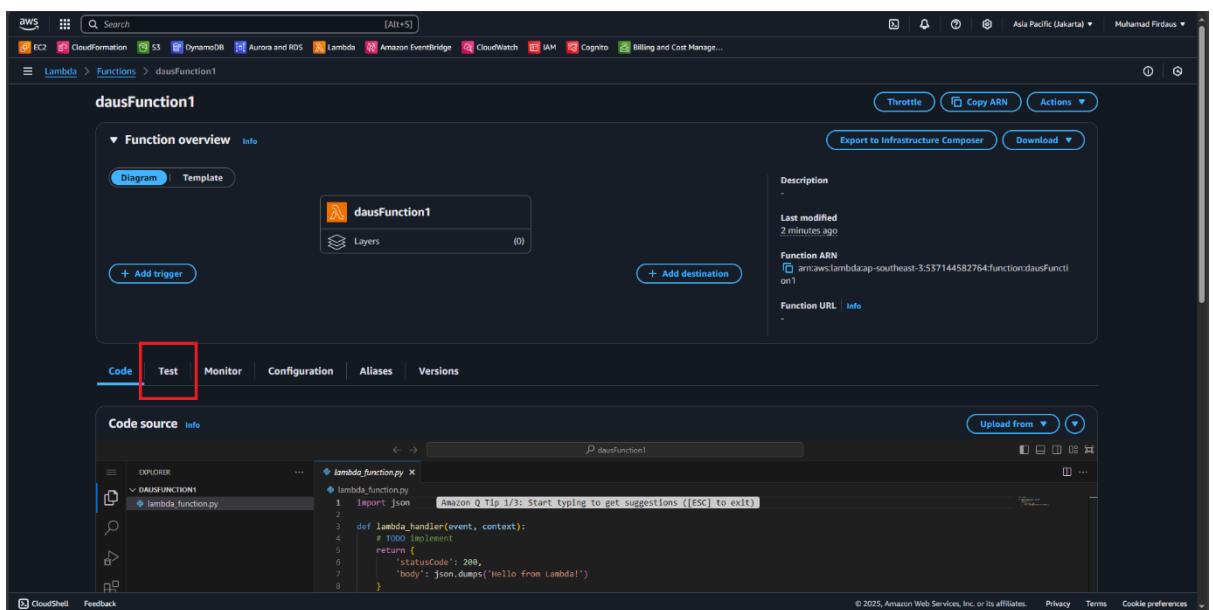
The code editor has tabs for .NET, Java, Node.js, Python (selected), Ruby, and Custom runtime. Buttons for 'Run' and 'Next: Lambda responds to events' are visible. The bottom of the screen shows standard AWS navigation and footer links.

- Fill the function details with information that you want, in this case I am using below options to fill the details and after that choose to create the function :

Option : Author from scratch  
 Function name : dausFunction1  
 Runtime : Python  
 Architecture : x64



- A pop up will show after the service created the new function, after the pop up choose the “Test” button on the bottom screen to open the section.



4. On the “Test” section, edit the details with the following details so we can use it to test the function and after that click on the orange “Test” button on the right above screen.

```
Test event action : Create new event
Event name : TestEvent
Event sharing settings : Private
Template : Hello World
```

The screenshot shows the AWS Lambda Test event configuration interface. At the top, there are tabs for Code, Test (which is selected), Monitor, Configuration, Aliases, and Versions. Below the tabs, there's a section for "Test event" with an "Info" link. A note says "To invoke your function without saving an event, configure the JSON event, then choose Test." There are two buttons: "CloudWatch Logs Live Tail" and "Save". An orange "Test" button is located at the top right of the main form area. The main form contains the following fields:

- Test event action:** Create new event (radio button selected)
- Event name:** TestEvent
- Event sharing settings:** Private (radio button selected)
- Template - optional:** Hello World

Below the form is a "Event JSON" section containing the following code:

```
1 * []
2   "key1": "value1",
3   "key2": "value2",
4   "key3": "value3"
5 []
```

There are "Format JSON" and "Copy" buttons next to the JSON code.

5. The result of the function test will then pop up to give us the details regarding the function testing.

The screenshot shows the AWS Lambda Test event results interface. At the top, there's a "Test event" section with an "Info" link and a note: "To invoke your function without saving an event, configure the JSON event, then choose Test." Below this is a "Logs" section with a green checkmark and the message "Executing function: succeeded (logs)". The "Logs" section contains a collapsed "Details" section showing a JSON object with "statusCode": 200 and "body": "\"Hello from Lambda!\"".

The main results section is titled "Summary" and includes the following details:

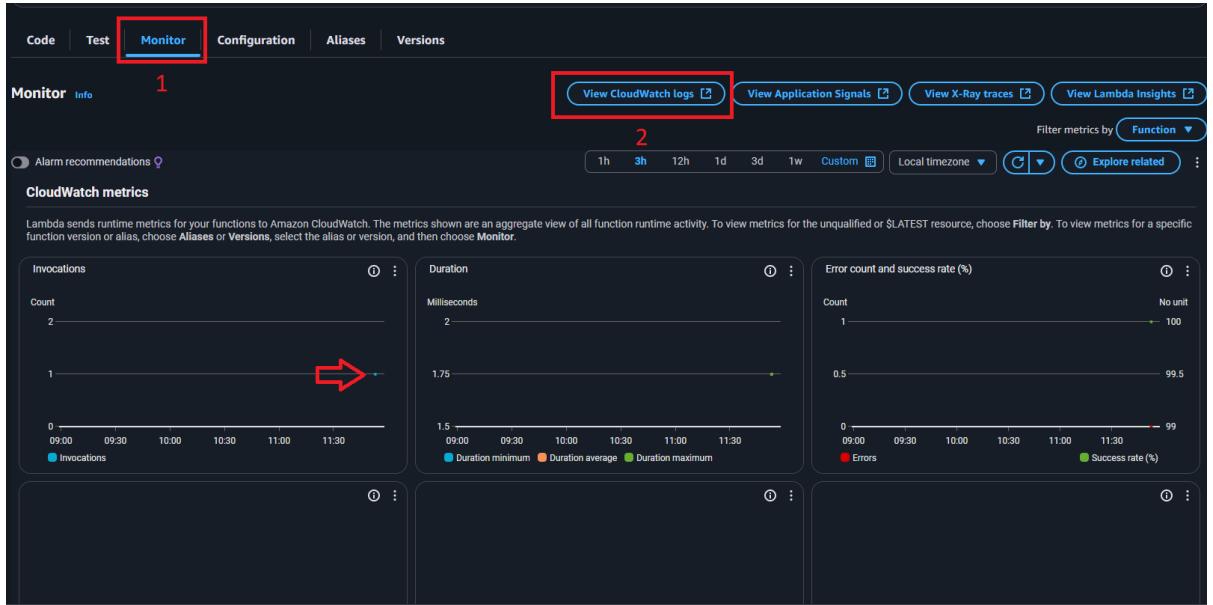
Code SHA-256	Execution time
HAPq9EReJVECSgLaVtc/gyd5vZtd9eiUGF952t0jBxY=	1 minute ago
Function version	Request ID
\$LATEST	24f188bf-ed93-4bc5-b341-8677390fe91c
Duration	Billed duration
1.75 ms	2 ms
Resources configured	Max memory used
128 MB	34 MB
Init duration	
76.70 ms	

Below the summary is a "Log output" section with the message: "The area below shows the last 4 KB of the execution log. Click here to view the corresponding CloudWatch log group." It contains the following log entries:

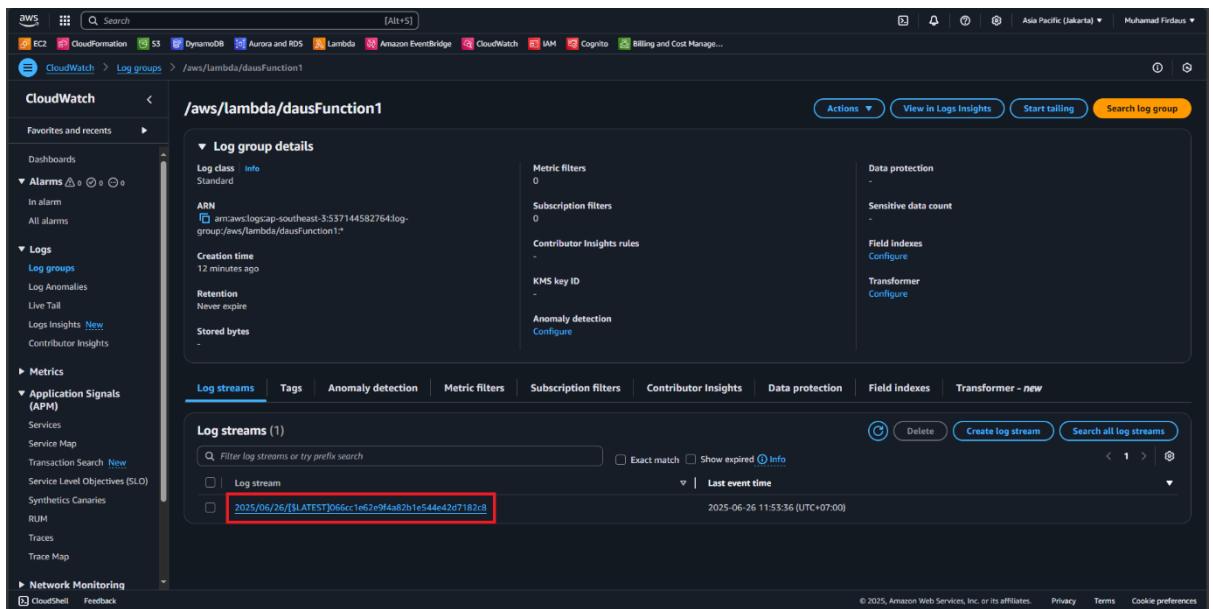
```
START RequestId: 24f188bf-ed93-4bc5-b341-8677390fe91c Version: $LATEST
END RequestId: 24f188bf-ed93-4bc5-b341-8677390fe91c
REPORT RequestId: 24f188bf-ed93-4bc5-b341-8677390fe91c Duration: 1.75 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 34 MB Init Duration: 76.70 ms
```

At the bottom, there are "CloudWatch Logs Live Tail", "Save", and "Test" buttons.

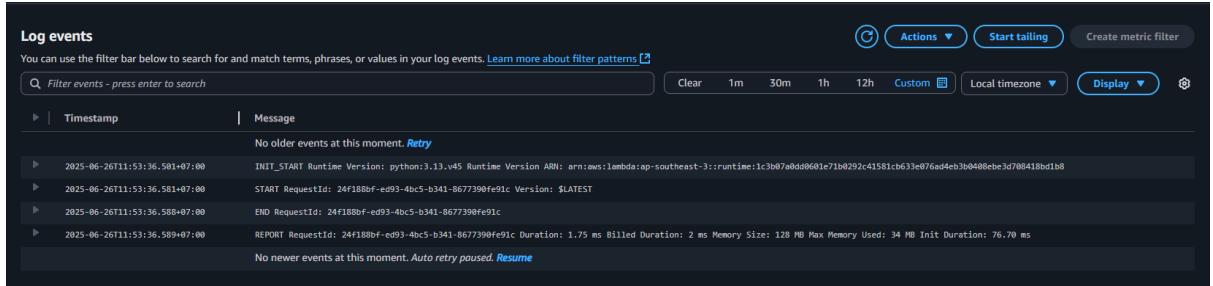
6. Now we can see the CloudWatch logs regarding the function testing on the CloudWatch section, click on “Monitoring” and choose “View CloudWatch Logs”. On the monitoring tab you can also see a dot that list our testing history activity.



7. On the CloudWatch pop up, choose the newest log name on the log stream (the blue one) and click it to get redirected to the logs menu.



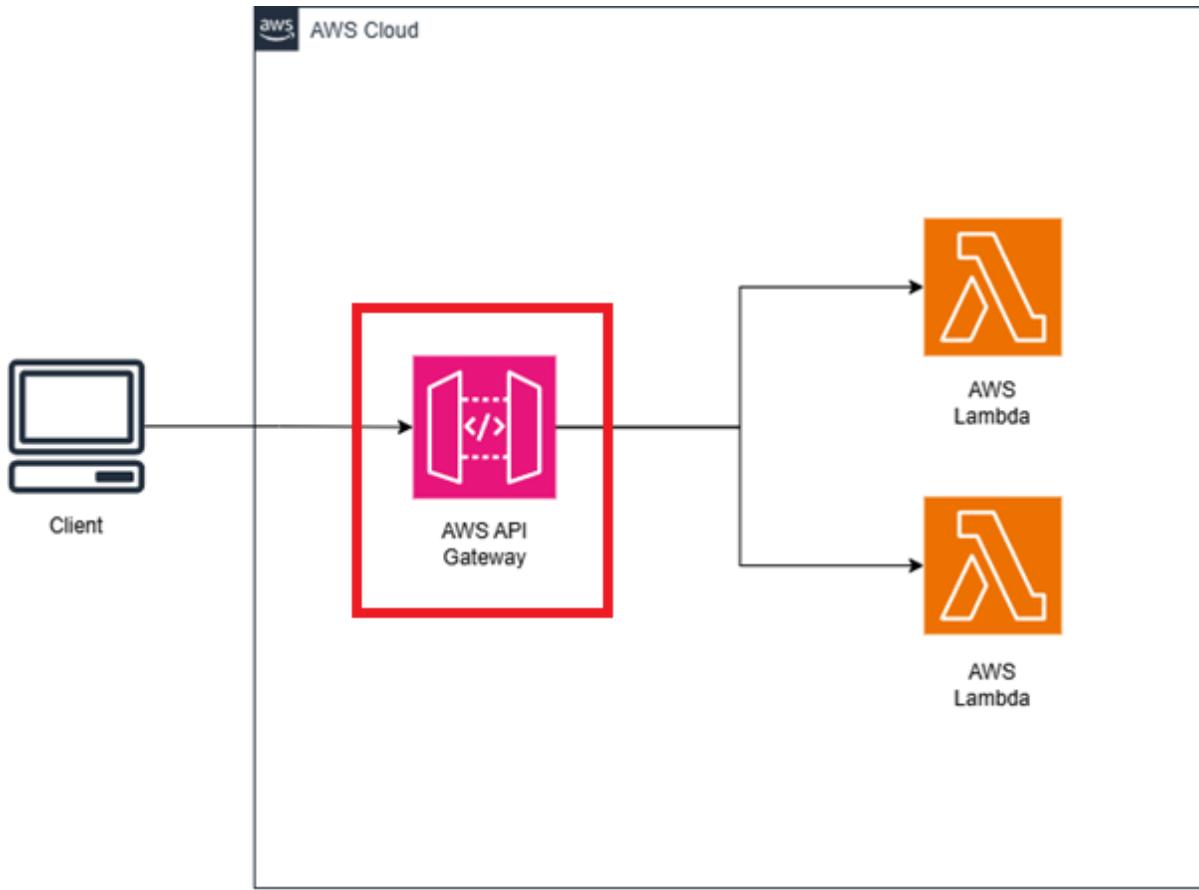
- The log event then will show the detail regarding the running function process that we can see.



A screenshot of the AWS Lambda Log Events interface. The interface has a header with 'Actions', 'Start tailing', 'Create metric filter', and a search bar. Below the header is a toolbar with 'Clear', time ranges ('1m', '30m', '1h', '12h', 'Custom'), 'Local timezone', 'Display' (with a dropdown), and a refresh icon. A table below the toolbar shows log events. The first row is a header with 'Timestamp' and 'Message'. The second row contains a timestamp '2025-06-26T11:53:36.501+07:00' and a message starting with 'INIT\_START Runtime Version: python:3.13.v45 Runtime Version ARN: arn:aws:lambda:ap-southeast-3::runtime:1c3b07a0dd0601e71b0292c41581cb633e076ad4eb3b0408eb3d708418bd1b8'. The third row contains a timestamp '2025-06-26T11:53:36.581+07:00' and a message starting with 'START RequestId: 24f188bf-e93-4bc5-b341-8677390fe91c Version: \$LATEST'. The fourth row contains a timestamp '2025-06-26T11:53:36.588+07:00' and a message starting with 'END RequestId: 24f188bf-e93-4bc5-b341-8677390fe91c'. The fifth row contains a timestamp '2025-06-26T11:53:36.589+07:00' and a message starting with 'REPORT RequestId: 24f188bf-e93-4bc5-b341-8677390fe91c Duration: 1.75 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 34 MB Init Duration: 76.70 ms'. At the bottom of the table, it says 'No newer events at this moment. Auto retry paused. Resume'.

Timestamp	Message
2025-06-26T11:53:36.501+07:00	INIT_START Runtime Version: python:3.13.v45 Runtime Version ARN: arn:aws:lambda:ap-southeast-3::runtime:1c3b07a0dd0601e71b0292c41581cb633e076ad4eb3b0408eb3d708418bd1b8
2025-06-26T11:53:36.581+07:00	START RequestId: 24f188bf-e93-4bc5-b341-8677390fe91c Version: \$LATEST
2025-06-26T11:53:36.588+07:00	END RequestId: 24f188bf-e93-4bc5-b341-8677390fe91c
2025-06-26T11:53:36.589+07:00	REPORT RequestId: 24f188bf-e93-4bc5-b341-8677390fe91c Duration: 1.75 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 34 MB Init Duration: 76.70 ms

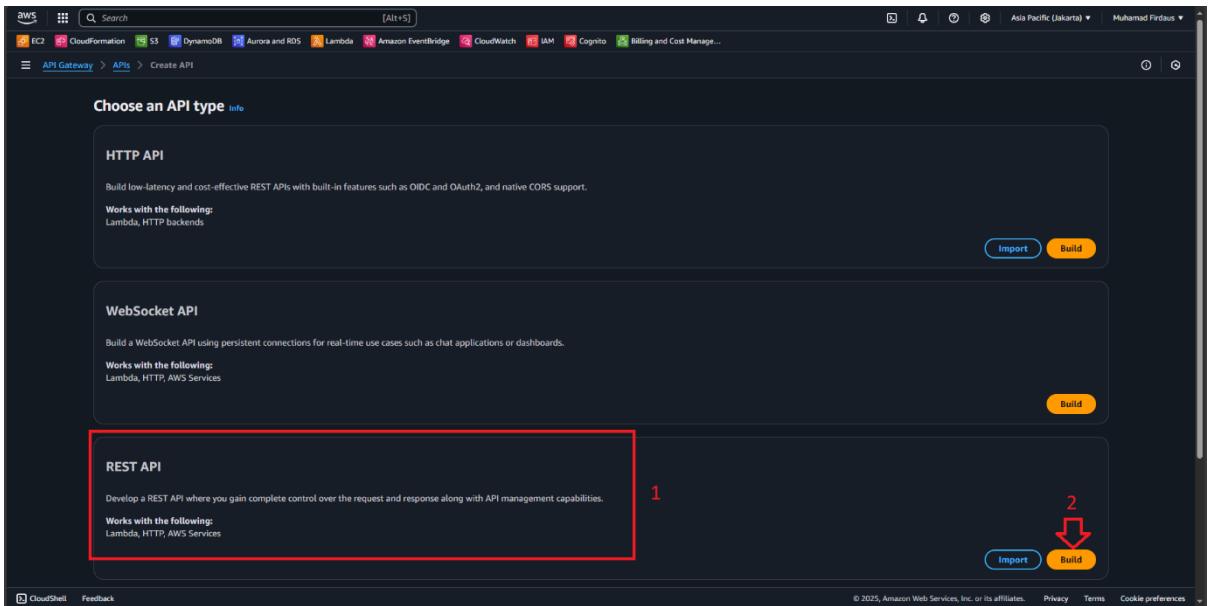
## PART 2 : Connecting Lambda function with the API Gateway



1. On the AWS Console, search for API Gateway service and enter the service. We will create the API Gateway (I will mention it further as APIGW) to connect to the lambda function. Choose to create a new API.

The screenshot shows the AWS API Gateway service page. The top navigation bar includes links for EC2, CloudFormation, S3, DynamoDB, Aurora and RDS, Lambda, Amazon EventBridge, CloudWatch, IAM, Cognito, Billing and Cost Management, and more. The main content area is titled 'API Gateway' and 'Create and manage APIs at scale'. It features a 'Get started' button and a 'How it works' section with a diagram showing various clients (Connected users and streaming dashboards, Web and mobile applications, IoT devices, Private applications: VPC and on-premises) connecting to an 'Amazon API Gateway' (represented by an orange box with a double door icon), which then connects to an 'Amazon CloudFront' and an 'Amazon Gateway cache'. The 'Pricing' section explains the cost model for different API types. The 'Resources' section provides links to Getting Started Guide, Developer Guide, and API References. At the bottom, there are links for CloudShell, Feedback, and a footer with copyright information.

- On the creation page, you will be asked with what kind of API you want to create. We will continue by using REST API (not the private one).



- On the API creation page, choose to fill the details as you see fit. In this case I will be using below details to continue and choose to create the API :

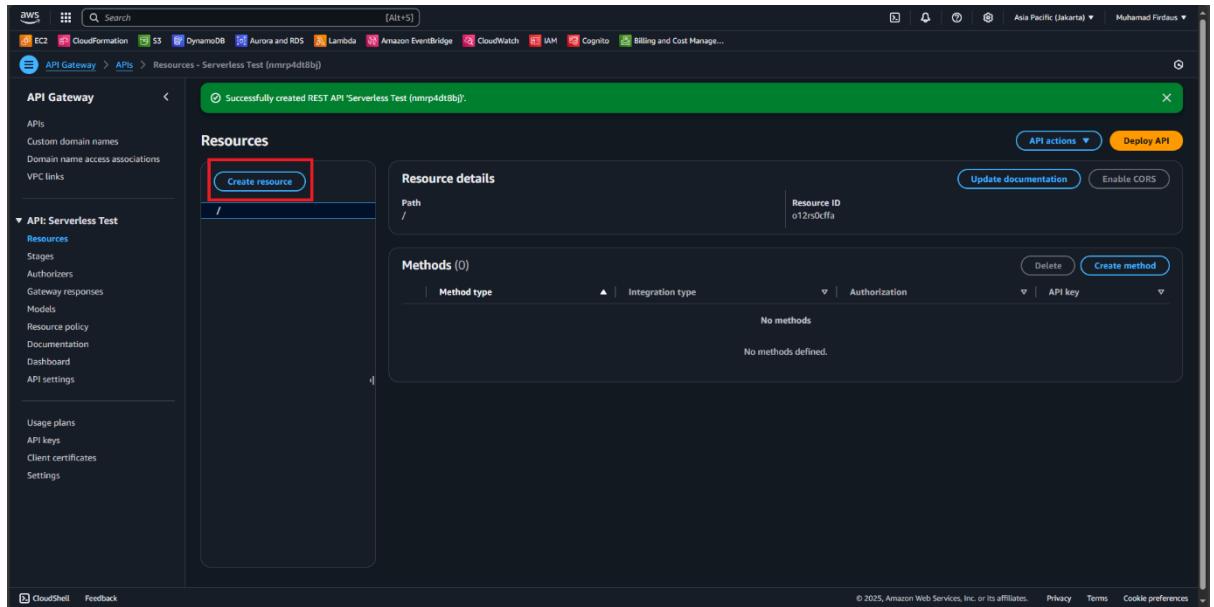
API Details : New API
API name : Serverless Test
Description : Testing Serverless Service
API endpoint type : Regional
IP address type : IPv4

The screenshot shows the 'Create REST API' page. The 'API details' section includes:

- API details**:
  - New API: Create a new REST API.
  - Clone existing API: Create a copy of an API in this AWS account.
  - Import API: Import an API from an OpenAPI definition.
  - Example API: Learn about API Gateway with an example API.
- API name**: Serverless Test
- Description - optional**: Testing Serverless Service
- API endpoint type**: Regional
- IP address type**:
  - IPv4: Supports only edge-optimized and Regional API endpoint types.
  - Dualstack: Supports all API endpoint types.

At the bottom right, there are 'Cancel' and 'Create API' buttons.

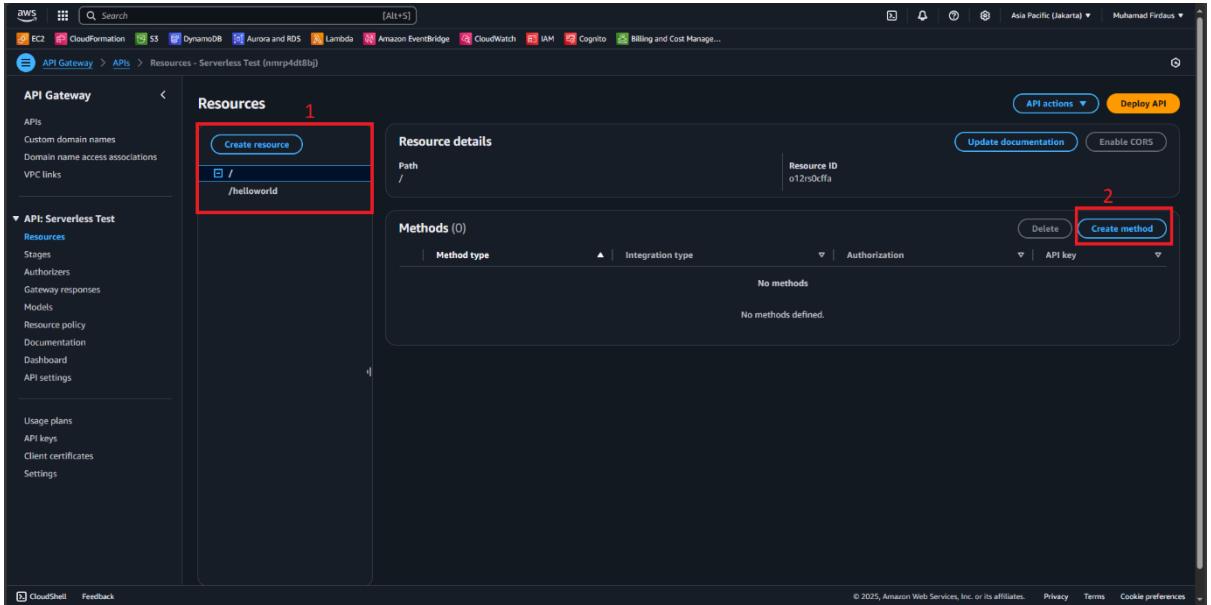
4. A page will then popup showing you the finished process of creating the new API, now on that popup page we will choose the “Create Resource” option to create a new resources.



5. On the create resource section, fill in the details with default setting while adding the name of the resource as “helloworld”.

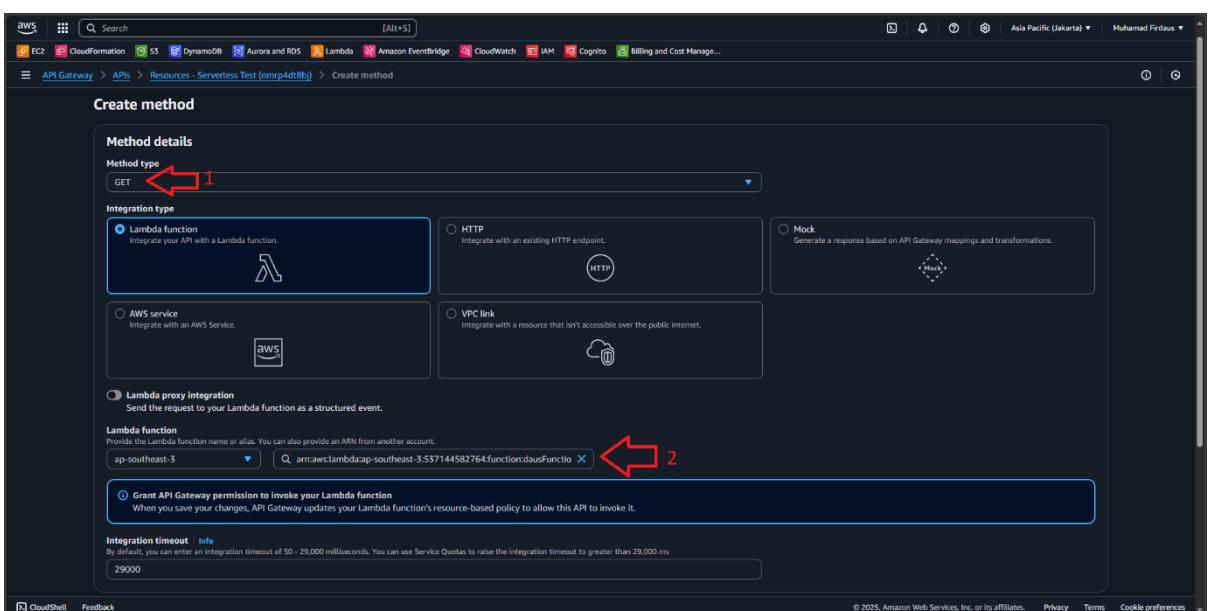
The screenshot shows the 'Create resource' dialog box. Under the 'Resource details' section, there is a 'Proxy resource info' link and a note about proxy resources. Below that, there is a 'Resource path' input field containing '/' and a 'Resource name' input field containing 'helloworld'. At the bottom right of the dialog are 'Cancel' and 'Create resource' buttons.

- After creating the resource, a new pop up will show us that the resource is successfully created. And then we choose the “create method” option AFTER we click on the /helloworld to create a new method for our resource under the /helloworld resource.



- On the create method page, we fill the details with the below information to proceed with our method creation and then choose create method.

Method type : GET  
 Integration type : Lambda function  
 Lambda proxy integration : off  
 Lambda function : our created function earlier in my case it is dausFunction1  
 Integration timeout : left it default



8. After the method was created, the GET method will appear under the /helloworld on the resource tab, confirm this to proceed. Then click on the “Deploy API” so we can begin to prepare to deploy our API

The screenshot shows the AWS API Gateway interface. On the left, there's a sidebar with 'API Gateway' selected. Under 'APIs', 'Custom domain names', 'Domain name access associations', and 'VPC links' are listed. Below that, 'AP: Serverless Test' has 'Resources' expanded, showing 'Stages', 'Authorizers', 'Gateway responses', 'Models', 'Resource policy', 'Documentation', 'Dashboard', and 'API settings'. Under 'Usage plans', 'API keys', 'Client certificates', and 'Settings' are listed. The main area is titled 'Resources 1' and shows a tree structure with a 'Create resource' button. A red box highlights the '/helloworld' node under the '/' node. To the right, a detailed view of the '/helloworld - GET - Method execution' is shown. It includes an ARN (arn:aws:execute-api:south-east-1:537144582764:nnmp4dt8bj/\*/GET/helloworld), a Resource ID (a1oydc), and a flow diagram showing the request path from Client to Lambda integration. The 'Method request' tab is selected. At the top right, a 'Deploy API' button is highlighted with a red box.

9. A popup will appear for us to configure the API deployment, use below option to fill the required field and then “Deploy”.

- 10. Stage : \*New stage\* (Its a dropdown menu)
- 11. Stage name : PROD
- 12. Description : PRODv1

**Deploy API**

Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. [Learn more](#)

**Stage**

\*New stage\*

**Stage name**

PROD

ⓘ A new stage will be created with the default settings. Edit your stage settings on the [Stage](#) page.

**Deployment description**

PRODv1

**Cancel** **Deploy**

13. After we clicked the deploy button, a new page pop up showing us the stages of the method, open all the stages until you get to the GET stages, click on it to reveal the method content and then copy the “Invoke URL” for us to be able to use it on API testing tools.

open all the way until "GET"

Invoke URL  
https://nmp4dt8bj.execute-api.ap-southeast-3.amazonaws.com/PROD/helloworld

14. Open up your API testing tools such as Postman API or Insomnia, then click on a new testing environment, use the GET method and then paste the Invoke URL we previously copied on the AWS API Gateway and then click on send request.

Enter URL or paste text

Send

15. If the configuration is correct, we will get the response from the AWS API Gateway with status 200 OK and the Hello from lambda message.

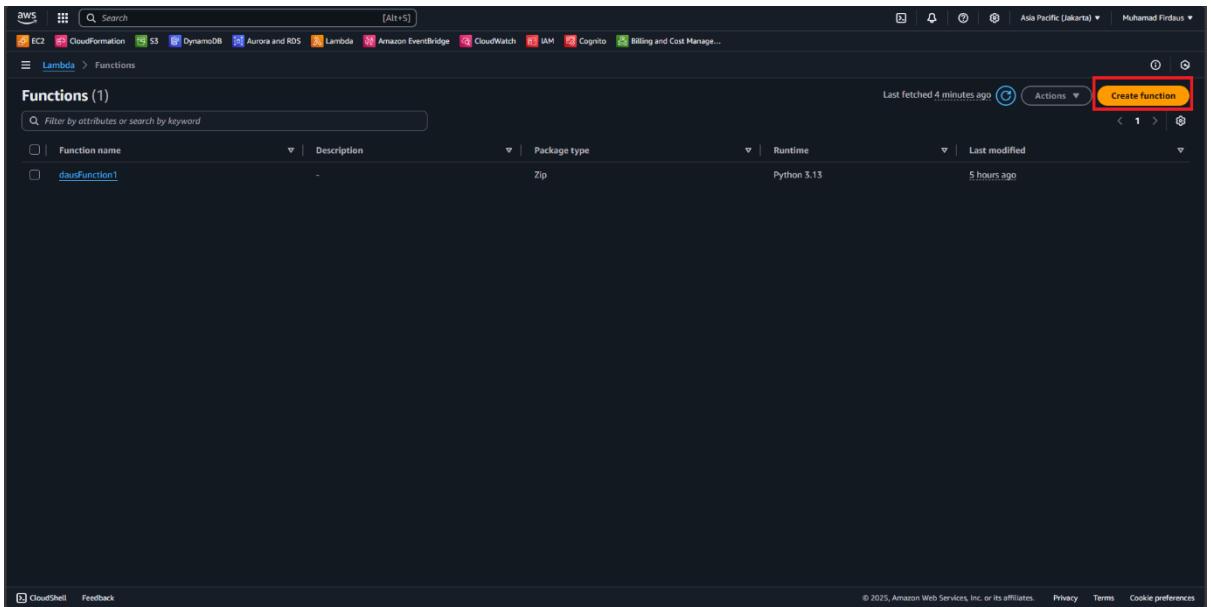
The screenshot shows the Postman application interface. On the left, there's a sidebar with 'My Workspace' selected, showing options for Collections, Environments (which is highlighted), Flows, and History. The main area has a 'GET https://nmrp4dt8bj.execute-api.ap-southeast-3.amazonaws.com/PROD/helloworld' request. The 'Body' tab of the response panel is selected, displaying a JSON response with a red border around it. The response body is:

```
{ } JSON ▾
1 {
2   "statusCode": 200,
3   "body": "\"Hello from Lambda!\""
4 }
```

The status bar at the bottom indicates a 200 OK response with 53 ms and 389 B.

## PART 3 : Implement API Canary on the API Gateway

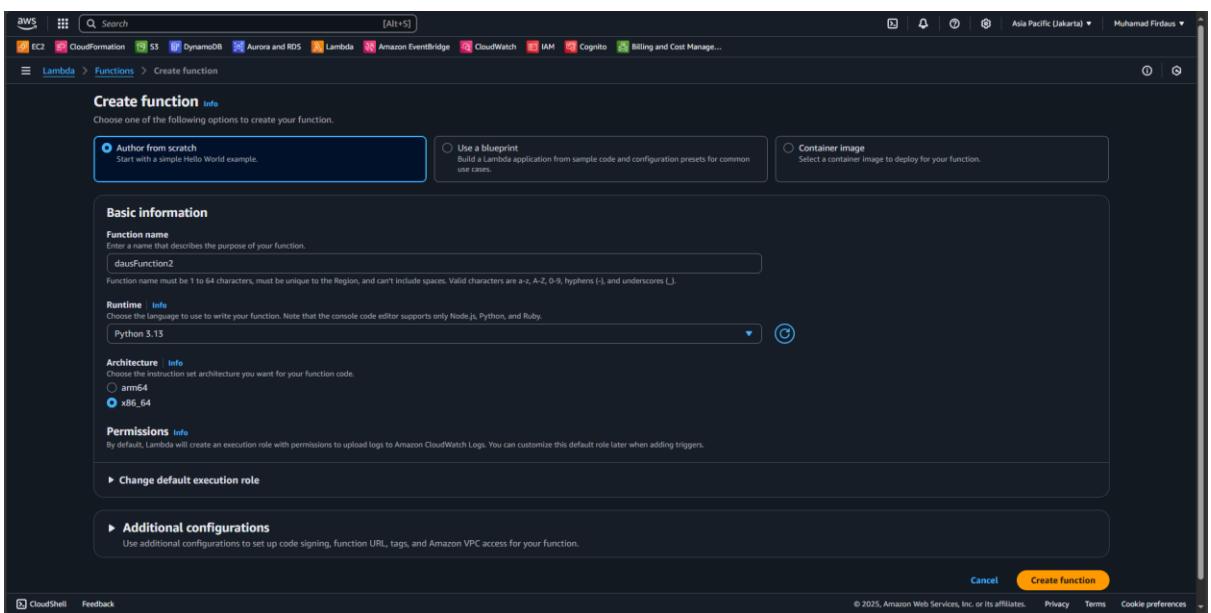
1. Open up the lambda service menu, we will create a new function this time.



The screenshot shows the AWS Lambda service interface. In the top navigation bar, the 'Lambda' icon is selected. Below it, the 'Functions' section displays one item: 'dausFunction1'. The function details are shown in a table: Function name (dausFunction1), Package type (Zip), Runtime (Python 3.13), and Last modified (5 hours ago). At the top right of the table, there is a 'Create function' button, which is highlighted with a red box.

2. Fill out the required information for our second function, I use the specification below for my second function.

Create function : Author from scratch  
Function name : dausFunction2  
Runtime : Python  
Architecture : x64



The screenshot shows the 'Create function' wizard. The first step, 'Basic information', is displayed. Under 'Function name', the value 'dausFunction2' is entered. Under 'Runtime', 'Python 3.15' is selected. Under 'Architecture', 'x64' is selected. There are three options at the top: 'Author from scratch' (selected), 'Use a blueprint' (unselected), and 'Container image' (unselected). The 'Create function' button is located at the bottom right of the form.

3. On the code source, we change the successful message a little bit. You can edit it however you want and DON'T FORGET TO DEPLOY THE CHANGES FIRST before proceeding.

```
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Greetings from Firdaus!')
    }
```

The screenshot shows the AWS Lambda Code Source Editor interface. The code editor window displays the following Python function:

```
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Greetings from Firdaus!')
    }
```

The sidebar on the left shows the project structure under 'EXPLORER' with a file named 'lambda\_function.py'. Below the code editor, there are sections for 'DEPLOY [UNDEPLOYED CHANGES]' and 'TEST EVENTS [NONE SELECTED]'. The status bar at the bottom indicates the code has 9 lines, 1 space, and is in UTF-8 LF Python mode.

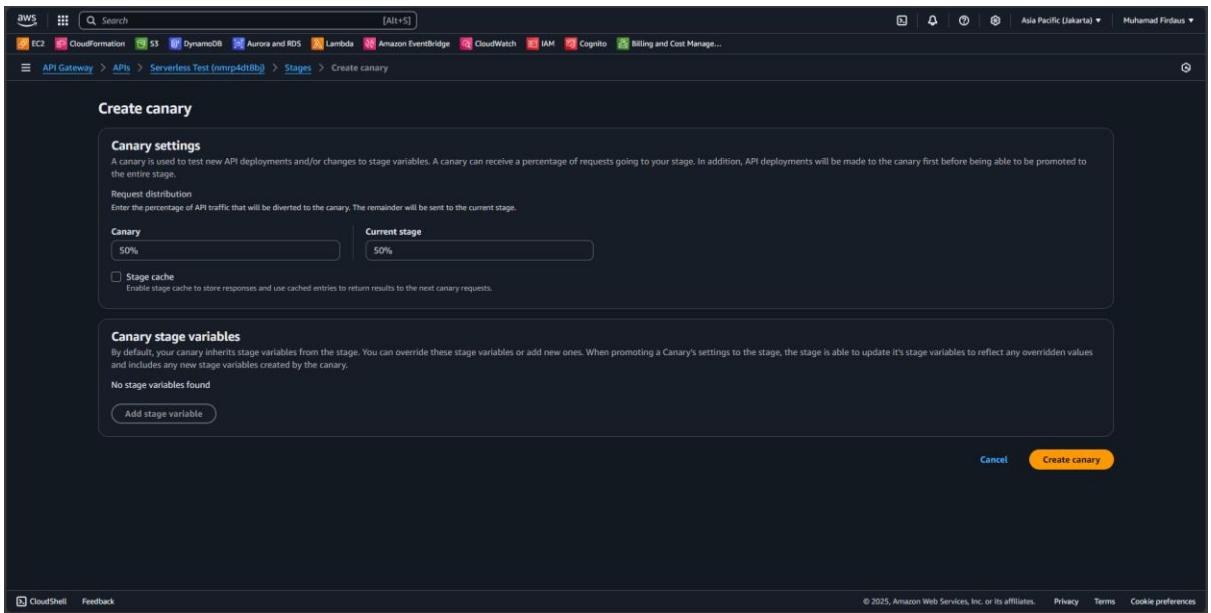
- On the stages section, click on the “PROD” option, and choose “canary” menu on the right bottom of the page.

The screenshot shows the AWS API Gateway interface. On the left, there's a sidebar for 'API Gateway' with sections like 'APIs', 'Custom domain names', 'Domain name access associations', 'VPC links', and 'API: Serverless Test'. Under 'API: Serverless Test', it lists 'Resources', 'Stages', 'Authorizers', 'Gateway responses', 'Models', 'Resource policy', 'Documentation', 'Dashboard', and 'API settings'. The 'Stages' section is expanded, showing 'Stages' with '1' and 'PROD' with '7' resources. Below 'PROD' is a 'GET /helloworld' resource. To the right, there are two main sections: 'Stage details' and 'Log and tracing'. 'Stage details' includes fields for 'Stage name' (PROD), 'Cache cluster info' (inactive), 'Default method-level caching' (inactive), and an 'Invoke URL' (https://nmp4dt8bj.execute-api.ap-southeast-3.amazonaws.com/PROD). 'Log and tracing' includes 'CloudWatch logs' (inactive), 'X-Ray tracing' (inactive), and 'Custom access logging' (inactive). At the bottom, tabs for 'Stage variables', 'Deployment history', 'Documentation history', 'Canary' (which is highlighted with a red underline), and 'Tags' are present. A red box labeled '1' is around the 'PROD' stage, and a red arrow labeled '2' points from the 'Canary' tab to the 'Create canary' button.

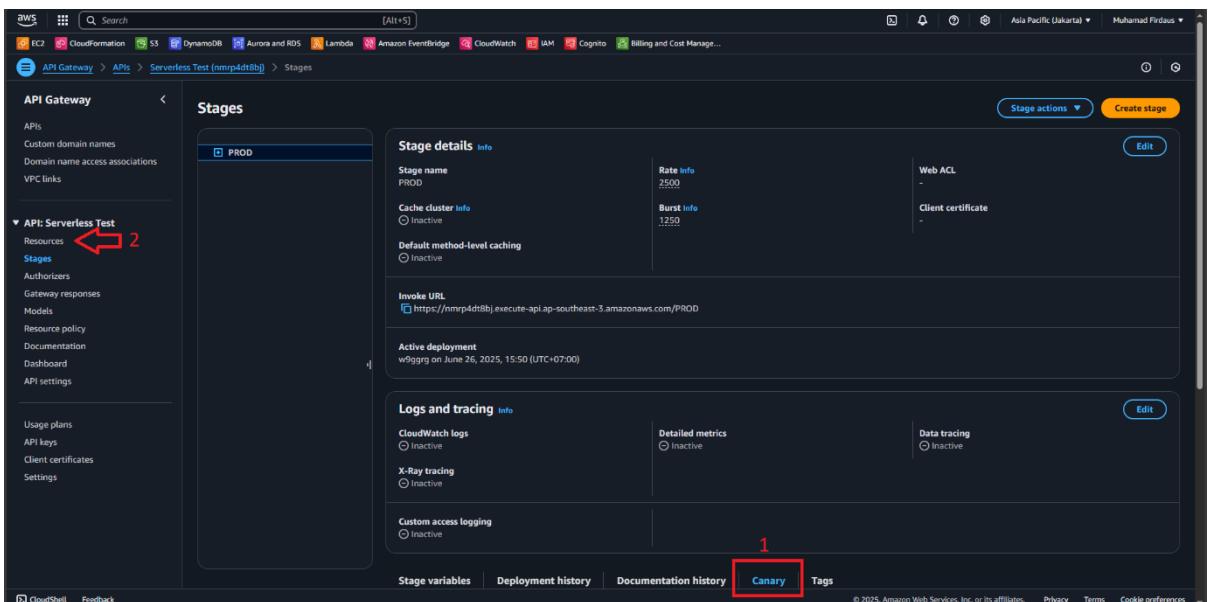
- Choose to create a new canary by clicking on “create canary” button.

The screenshot shows the 'Canary settings' page. At the top, tabs for 'Stage variables', 'Deployment history', 'Documentation history', 'Canary' (which is highlighted with a red underline), and 'Tags' are visible. Below, a section titled 'Canary settings' shows a message 'No canary associated with the stage.' and a large 'Create canary' button, which is highlighted with a red box.

- On the create canary menu, I will be splitting the canary and the current stages with equal percentages of 50%, this will split the traffic in equal way towards both stages.



- Back to the stages menu, check if your new canary is already created, and then click on the resource tab as we will create a new resource for the canary stages.



- Click on the “Create resource” button to open up the creation menu.

The screenshot shows the AWS API Gateway interface. On the left, there's a sidebar with 'APIs' and 'API: Serverless Test' sections. The main area is titled 'Resources' and shows a single resource entry: '/ /helloworld' with a 'GET' method. At the top right, there are 'API actions' and 'Deploy API' buttons. A prominent red arrow points to the 'Create resource' button at the top left of the main content area.

9. On the resource detail page, enter “v2” or any other resource name you want to name it, still within the “/” path, then choose to create this resource.

The screenshot shows the 'Create resource' dialog box. In the 'Resource details' section, the 'Proxy resource info' option is selected. The 'Resource path' field contains a single slash ('/'). The 'Resource name' field is filled with 'v2'. At the bottom right, there are 'Cancel' and 'Create resource' buttons.

10. Click on the “/v2” to choose it as the current path and then click again on the create resource menu to create a new resource under the “v2” path.

The screenshot shows the AWS API Gateway Resources page. A green success message at the top says "Successfully created resource '/v2'". On the left, a sidebar for the "API: Serverless Test" API shows various settings like Stages, Authorizers, and Models. The main area displays a tree view of resources under the root "/". One node, "/v2", is highlighted with a red box and labeled "1". Another red box highlights the "Create resource" button at the top left of the main content area, labeled "2". To the right, a "Resource details" panel shows the path "/v2" and a "Methods (0)" section. At the bottom right of the main area, there are "Delete", "Update documentation", and "Enable CORS" buttons.

11. Make sure to check the resource path is on the “v2” and then input the name you want to be using as the resource name, I will be choosing another “helloworld”.

The screenshot shows the "Create resource" dialog box. In the "Resource details" section, the "Proxy resource info" option is selected. Below it, the "Resource path" field contains "/v2/" and the "Resource name" field contains "helloworld". There is also a checked "CORS (Cross Origin Resource Sharing) info" checkbox. At the bottom right of the dialog are "Cancel" and "Create resource" buttons.

12. When the “helloworld” resource is created inside the “v2”, click on it and then choose to create a new method inside the helloworld resource.

The screenshot shows the AWS API Gateway interface. On the left, the navigation pane is open with 'APIs' selected. Under 'APIs', 'Custom domain names', 'Domain name access associations', and 'VPC links' are listed. Below that, under 'API: Serverless Test', 'Resources' is selected, followed by 'Stages', 'Authorizers', 'Gateway responses', 'Models', 'Resource policy', 'Documentation', 'Dashboard', and 'API settings'. On the right, the main panel shows 'Resources' with a 'Create resource' button. A tree view shows a root node '/' with a child node '/helloworld'. Under '/helloworld', there is a 'GET' method and a 'v2' folder. Inside 'v2', there is another '/helloworld' node. The 'Methods (0)' section has a 'Create method' button highlighted with a red box. Other tabs like 'Method type', 'Integration type', 'Authorization', and 'API key' are visible but not selected.

13. We'll be using the GET method again, and this time we choose the “dausFunction2” as the lambda function. Create the method after leaving all the settings in default.

The screenshot shows the 'Create method' dialog for a GET method on the '/helloworld' resource. The 'Method details' section shows 'Method type' set to 'GET'. The 'Integration type' section has 'Lambda function' selected, which is highlighted with a blue background. Other options like 'HTTP', 'Mock', 'AWS service', and 'VPC link' are shown but not selected. Below the integration type, 'Lambda proxy integration' is checked. The 'Lambda function' field contains 'arn:aws:lambda:ap-southeast-3:537144582764:function:dausFunction2'. A tooltip for this field says: 'Provide the Lambda function name or alias. You can also provide an ARN from another account.' The 'Grant API Gateway permission to' field contains 'arn:aws:lambda:ap-southeast-3:537144582764:function:dausFunction2'. A tooltip for this field says: 'When you save your changes, API Gateway will grant the specified Lambda function permission to invoke it.' At the bottom, the 'Integration timeout' field is set to 29000. The entire dialog is contained within a modal window.

14. Click on the GET method inside the v2 helloworld, then choose to deploy a new API inside of it.

The screenshot shows the AWS API Gateway Resources page. On the left, there's a sidebar with options like APIs, Stages, Authorizers, and Models. The main area shows a tree structure of resources under 'APIs'. A specific node, '/v2/helloworld - GET - Method execution', is selected. At the top right of this node, there's a 'Deploy API' button, which is highlighted with a red box. Below the tree, there's a flow diagram showing the request and response paths from a 'Client' to a 'Lambda integration'. Underneath the tree, there's a section for 'Method request settings' with fields for Authorization (set to NONE), Request validator (set to None), and Request paths (empty). The ARN of the resource is also visible at the top.

15. On the stage, we'll be choosing “PROD” it will mention that canary is now enabled if we choose it, fill the description however you like and create the API.

The screenshot shows the 'Deploy API' dialog box. At the top, it says 'Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. [Learn more](#)'. Below this, there's a warning message in a yellow box: '⚠️ When you deploy an API to an existing stage, you immediately overwrite the current stage configuration with a new active deployment.' Underneath, there's a 'Stage' dropdown menu set to 'PROD (Canary enabled)'. Below that is a 'Deployment description' input field containing 'Canary Deployment'. At the bottom right, there are 'Cancel' and 'Deploy' buttons, with 'Deploy' being highlighted with a yellow background.

16. On the stages menu, open all the stages until you click the GET section so you can see the invoke url, copy it so we can use it to send a request to the canary gateway.

The screenshot shows the AWS API Gateway interface. In the left sidebar, under 'APIs', there's a section for 'Serverless Test'. Under 'Stages', 'PROD' is selected. Below the stages tree, there's a 'Method overrides' section with a note: 'By default, methods inherit stage-level settings. To customize settings for a method, configure method overrides.' A note below says 'This method inherits its settings from the 'PROD' stage.' At the bottom, the 'Invoke URL' is displayed as a link: <https://nmrp4dt8bj.execute-api.ap-southeast-3.amazonaws.com/PROD/helloworld>.

17. Open up your API testing tools, again I will be using Postman, enter a new task, choose HTTP and use the GET method, then paste the Invoke URL we copied before (don't forget to add the /v2/ path before the helloworld since we havent configured it to be automatically invoked on the URL) then choose to sent your request. If your configuration is correct then you will see your message in the different setup than the first lambda function as seen here.

The screenshot shows the Postman interface. On the left, there's a sidebar with 'My Workspace', 'Collections', 'Environments' (which is selected), 'Flows', and 'History'. The main area shows a 'Globals' section with a character icon. Below it, a message says 'You don't have any environments.' and 'An environment is a set of variables that allows you to switch the context of your requests.' There's a 'Create Environment' button. The central part of the screen shows a task configuration: 'URL' is set to 'https://nmrp4dt8bj.execute-api.ap-southeast-3.amazonaws.com/PROD/v2/helloworld', 'Method' is 'GET', 'Headers' has '(7)' items, and 'Body' is set to 'JSON' with the following content: { "statusCode": 200, "body": "\"Greetings from Firdaus!\""}'. The status bar at the bottom right indicates a successful response: '200 OK'.

18. Return to the AWS console and click on the PROD menu, and choose on canary option.

The screenshot shows the AWS API Gateway Stages page. On the left, there's a sidebar with 'APIs', 'Custom domain names', 'Domain name access associations', 'VPC links', and a expanded section 'API: Serverless Test' containing 'Stages', 'Authorizers', 'Gateway responses', 'Models', 'Resource policy', 'Documentation', 'Dashboard', and 'API settings'. The 'Stages' section is currently active. In the main area, there's a tree view under 'Stages' with 'PROD' expanded, showing ' /' and '/helloworld' with a 'GET' method. To the right of the tree view is a 'Stage details' section for 'PROD' with tabs for 'Info', 'Logs and tracing', and 'Custom access logging'. Below these tabs are buttons for 'Stage actions' (dropdown), 'Create stage' (button), and 'Edit' (button). At the bottom of the page are buttons for 'CloudShell', 'Feedback', and links for '© 2025, Amazon Web Services, Inc. or its affiliates.', 'Privacy', 'Terms', and 'Cookie preferences'.

19. Now after confirming that the canary is reachable, we can promote the canary by clicking on the “Promote Canary” button inside the canary option menu.

The screenshot shows the 'Canary' tab of the 'Canary settings' page. It has sections for 'Requests directed to canary' (50%), 'Requests directed to 'PROD'' stage (50%), 'Deployment date' (June 26, 2025, 15:50 (UTC+07:00)), 'Description' (empty), 'Cache' (disabled), and 'Stage variables (0)'. There's also a table for 'Name', 'Value', and 'Canary override' with a note 'No variables'. A red box highlights the 'Promote canary' button at the top right of the page.

20. A new pop up will show and ask you regarding the configuration for the canary promotion, checks all the box to update the settings of the gateway and then click on “Promote canary” to promote the new developed canary.



21. After a successful promotion we can see the GET method from the v2/helloworld is now live and deployed to the PROD. This means the helloworld will be changed with the v2 version of it.

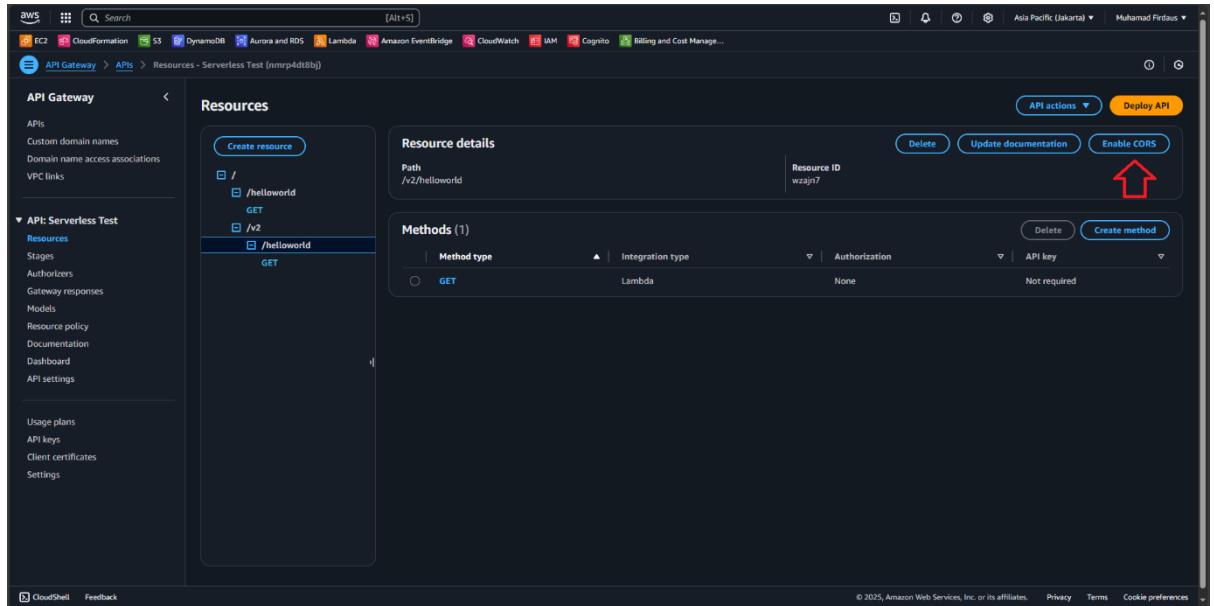
A screenshot of the API Gateway Stages and Stage details interface. The left panel shows the "Stages" list with "PROD" expanded, showing endpoints for "/" and "/helloworld" (both GET methods) and "/v2/helloworld" (GET method). The right panel shows the "Stage details" for "PROD", including "Stage name: PROD", "Cache cluster: Inactive", and "Default method: Inactive".

Stages	
<input type="checkbox"/> PROD	
<input type="checkbox"/> /	
<input type="checkbox"/> /helloworld	GET
<input type="checkbox"/> /v2/helloworld	GET

Stage details	
Stage name	PROD
Cache cluster	(-) Inactive
Default method	(-) Inactive

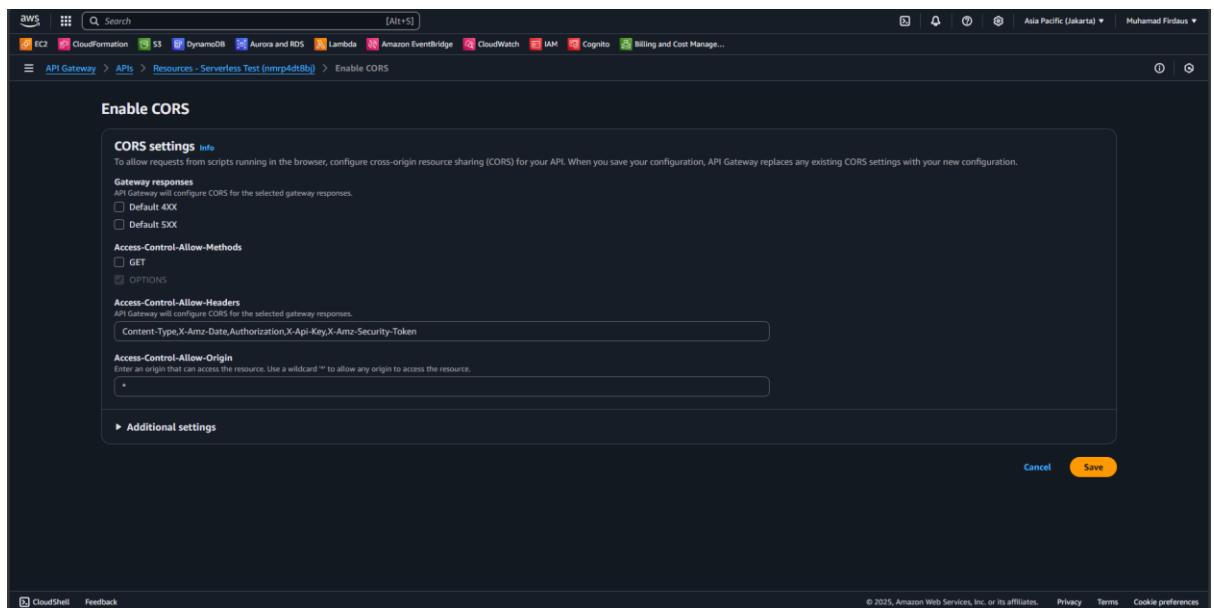
## PART 4 : Enabling CORS for the API Gateway

1. On the “Resources” menu, click on the resource you want to use to enable the CORS method. Click on the “Enable CORS” button on the top right section.



The screenshot shows the AWS API Gateway interface. In the left sidebar, under the 'APIs' section, there is a tree view of resources. One node is expanded to show methods: '/helloWorld' (GET) and '/v2/helloworld' (GET). In the top right corner of the main content area, there is a row of buttons: 'Delete', 'Update documentation', and 'Enable CORS'. A red arrow points to the 'Enable CORS' button.

2. A pop up will show the CORS enabling menu, just choose the default and save this configuration to proceed (We will be using the OPTIONS method).



The screenshot shows the 'Enable CORS' configuration dialog box. It has several sections:

- CORS settings**: A note stating "To allow requests from scripts running in the browser, configure cross-origin resource sharing (CORS) for your API. When you save your configuration, API Gateway replaces any existing CORS settings with your new configuration."
- Gateway responses**: Options for "Default 4XX" and "Default 5XX".
- Access-Control-Allow-Methods**: Options for "GET" and "OPTIONS". "OPTIONS" is checked.
- Access-Control-Allow-Headers**: A list containing "Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token".
- Access-Control-Allow-Origin**: A text input field with a placeholder "Enter an origin that can access the resource. Use a wildcard '\*' to allow any origin to access the resource." It contains a single asterisk (\*)
- Additional settings**: A link to another section.

At the bottom right are 'Cancel' and 'Save' buttons, with 'Save' being highlighted.

3. An OPTION resource will appear under the V2 and GET method we created earlier, this confirms that the CORS is already activated but not yet deployed on the PROD since we enable canary.

The screenshot shows the AWS API Gateway console. In the left sidebar, under 'APIs', 'Custom domain names', 'Domain name access associations', and 'VPC links' are visible. Under 'API: Serverless Test', 'Resources', 'Stages', 'Authorizers', 'Gateway responses', 'Models', 'Resource policy', 'Documentation', 'Dashboard', and 'API settings' are listed. On the right, a green banner at the top says 'Successfully enabled CORS'. Below it, the 'Resources' section shows a tree view with '/ /helloworld /GET /v2 /helloworld /GET /OPTIONS'. The 'Resource details' panel shows the path '/v2/helloworld' and the resource ID 'wzajn7'. The 'Methods' table lists two methods: 'GET' (Integration type: Lambda, Authorization: None, API key: Not required) and 'OPTIONS' (Integration type: Mock, Authorization: None, API key: Not required). Buttons for 'Delete', 'Update documentation', and 'Enable CORS' are also present.

4. Before we deploy to the PROD env, click on the OPTION resource and choose the “intergration response” then “edit” to edit the response as we will use this after this one.

The screenshot shows the AWS API Gateway console. The left sidebar is identical to the previous screenshot. The main area shows the 'Resources' section with the same tree structure. A red arrow labeled '1' points to the 'OPTIONS' method under the /v2/helloworld resource. A large red arrow labeled '2' points to the 'Integration response' tab in the bottom navigation bar. A red arrow labeled '3' points to the 'Edit' button in the 'Default - Response' section. The 'Integration responses' table shows a single row for 'Default - Response' with 'HTTP status regex' set to '' and 'Method response status code' set to '200'. The 'Content handling' dropdown is set to 'Passthrough'. The 'Default mapping' checkbox is checked. The 'Header mappings' table has three entries: 'Name' and 'Mapping value'. Buttons for 'Create response', 'Edit', and 'Delete' are available.

- Immediately scroll to the bottom section and click the “Mapping Template” to open the advanced menu, then fill the content type name as below or however you like.

application/json

The screenshot shows the 'Edit integration response' interface in the AWS Management Console. The 'Content type' field is set to 'application/json'. A red arrow labeled '1' points to the 'Mapping templates' section, and another red arrow labeled '2' points to the 'Content type' field.

- Scroll to the bottom again until you see the “Template body” section, fill it with the message you want to get during another API testing we will be having after this edit.

```
{
  "message": "CORS preflight response success"
}
```

The screenshot shows the 'Edit integration response' interface in the AWS Management Console. The 'Template body' section contains the following JSON code:

```

1 + {
2   "message": "CORS preflight response success"
3 }
4

```

A red arrow points to the 'Template body' section.

- After successfully editing the response, finally its time to deploy this version to the current PROD env. Choose the “Deploy API” to deploy this version of the API to the gateway.

The screenshot shows the AWS API Gateway Resources page. On the left, there's a sidebar with 'API Gateway' and 'API: Serverless Test' sections. The main area shows a tree view of resources: / (with /helloworld, GET), /v2 (with /helloworld, GET, OPTIONS). A red arrow labeled '1' points to the 'OPTIONS' method under /v2/helloworld. On the right, there's a detailed view of the /v2/helloworld - OPTIONS - Method execution. It includes a flow diagram from Client to Method request to Integration request to Integration response to Method response. At the top right of this panel is a 'Deploy API' button, which has a red box and arrow labeled '2' pointing to it. Other buttons like 'Update documentation' and 'Delete' are also visible.

- Confirm the changes and updates.

The screenshot shows the 'Deploy API' dialog box. It has a title 'Deploy API' and a sub-instruction: 'Create or select a stage where your API will be deployed. You can use the deployment history to revert or change the active deployment for a stage. [Learn more](#)'. Below this is a warning message in a box: '⚠️ When you deploy an API to an existing stage, you immediately overwrite the current stage configuration with a new active deployment.' The 'Stage' dropdown is set to 'PROD (Canary enabled)'. The 'Deployment description' text area contains the text 'CORS Enabling'. At the bottom are 'Cancel' and 'Deploy' buttons, with 'Deploy' being highlighted by a red box.

- Finally, we go to the canary section and use that Promote canary button to promote this API version to the PROD env.

The screenshot shows the 'Canary' tab of the AWS Lambda console. It displays 'Requests directed to canary' at 0% and 'Requests directed to 'PROD' stage' at 100%. The 'Deployment date' is June 27, 2025, 18:33 (UTC+07:00). The 'Cache' setting is disabled. Below this, there's a 'Stage variables (0)' section with a search bar and a table showing 'No variables'. At the top right, there are 'Edit', 'Delete', and 'Promote canary' buttons, with a red arrow pointing to the 'Promote canary' button.

- Promote the canary using the configurations you want.

The screenshot shows the 'Promote canary' dialog box. It contains three checked options:

- Update stage with canary's deployment
- Update stage with canary's variables
- Set canary percentage to 0.0%

At the bottom, there are 'Cancel' and 'Promote canary' buttons.

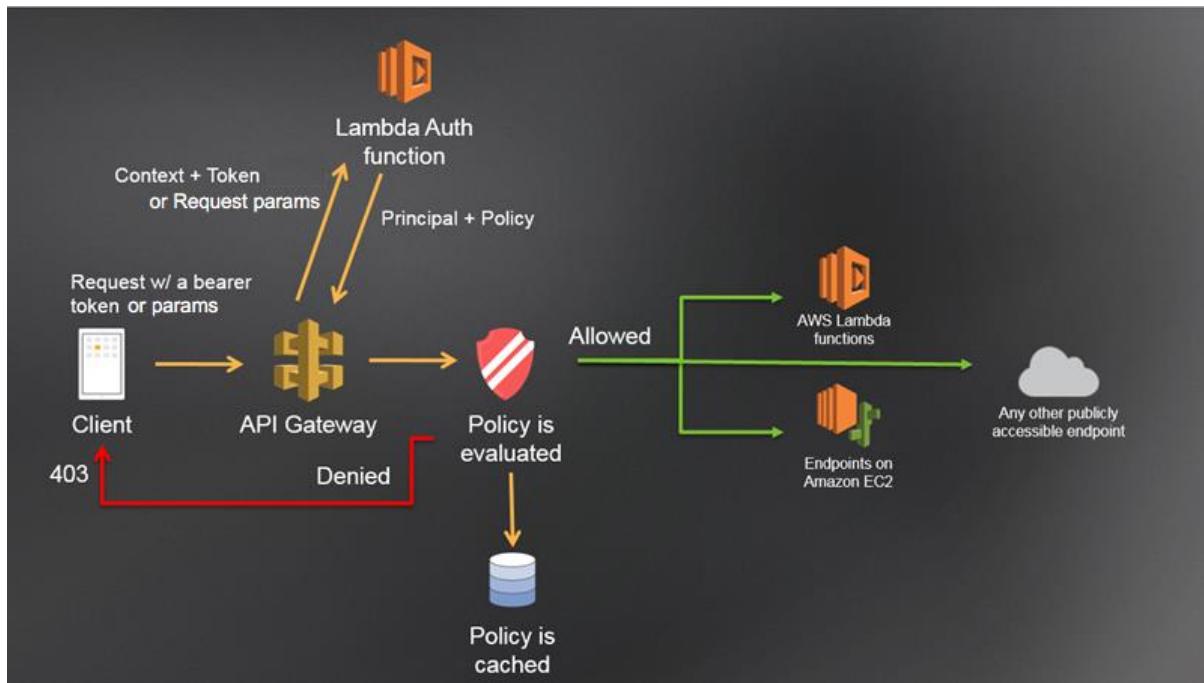
- Check it via curl to see if the API Gateway is responding with the 200 OK message.

```
< HTTP/2 200
< date: Fri, 27 Jun 2025 11:36:04 GMT
< content-type: application/json
< content-length: 0
< x-amzn-requestid: 54a93839-2ec0-458b-9169-35f6bbb133c9
< access-control-allow-origin: *
< access-control-allow-headers: Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Security-Token
< x-amz-apigw-id: M0lF1E1giMoECyw=
< access-control-allow-methods: OPTIONS
<
* Connection #0 to host nmrp4dt8bj.execute-api.ap-southeast-3.amazonaws.com left intact
ashurafirdaus@Ashura-Firdaus:/mnt/c/Users/zordl$ -
```

12. Or use Postman API or any API testing tools you want to use to check the changes and see the response now confirming that the CORS is now already applied to the PROD env using the OPTIONS method.

The screenshot shows the Postman application interface. In the top navigation bar, 'My Workspace' is selected. The main workspace shows a placeholder message: 'You don't have any environments.' Below this, there's a note: 'An environment is a set of variables that allows you to switch the context of your requests.' A 'Create Environment' button is available. The central area displays an API request configuration. The method is set to 'OPTIONS', the URL is 'https://nmp4dt8bj.execute-api.ap-southeast-3.amazonaws.com/PROD/v2/helloworld', and the 'Headers' tab is selected. The Headers section contains a single entry: 'Content-Type: application/json'. The 'Body' tab is also visible. At the bottom of the interface, the status bar shows '200 OK', '44 ms', '447 B', and other connectivity icons.

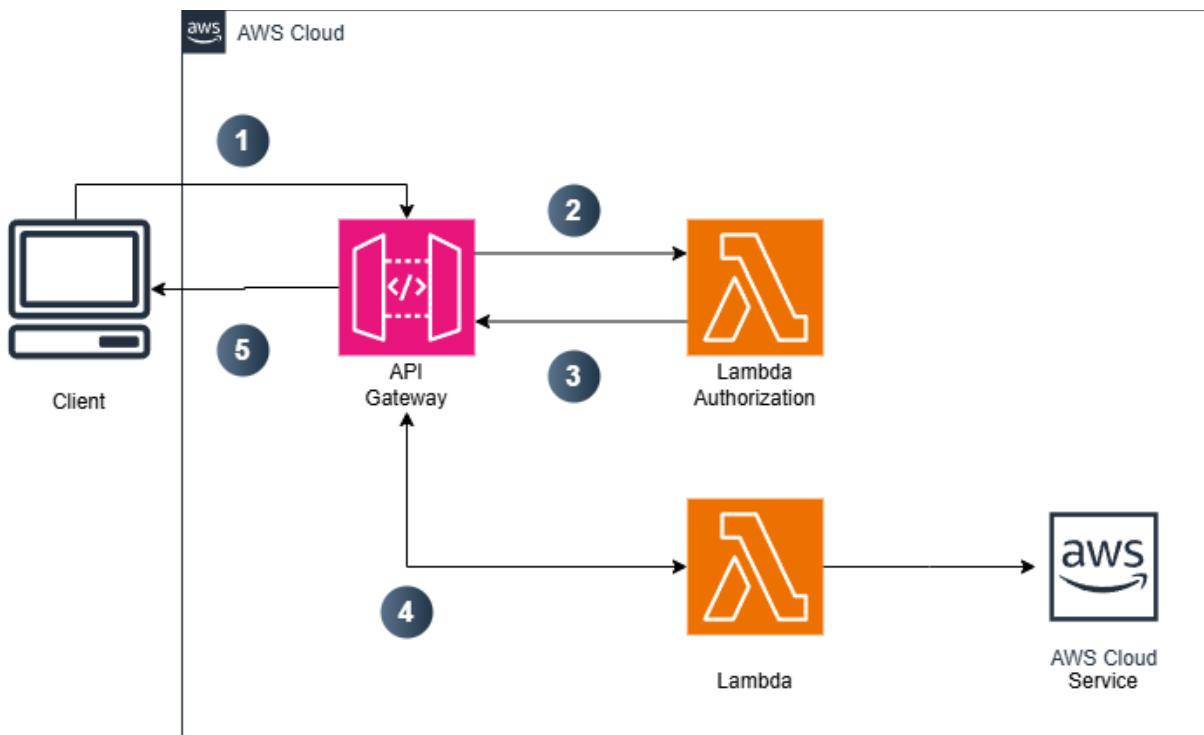
## PART 5 : Control your access using Lambda Authorizer



The diagram shows the authorization workflow for a Lambda authorizer.

(Source : <https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-use-lambda-authorizer.html>)

Architecture :



1. Open up command line to use the openssl rand command, in this case I'll be using my WSL environment to create the secret key for the AWS lambda authorizer.

```
openssl rand 256 | base64
```



The screenshot shows a terminal window with the following text:

```
ashurafirdaus@Ashura-Firdaus: /mnt/c/Users/zordl
C:\Users\zordl>wsl -d ubuntu
ashurafirdaus@Ashura-Firdaus:/mnt/c/Users/zordl$ openssl rand 256 | base64
PmvCtfr1Hy4mAQhTARjywokZRh4KjEyGfkXdGvULMVMhJHj79h3L14CTwB5WN5n9PtMfdazGSgN
RJlb97ni7GB1phKCo6rLT4SN36kBi7UIIkW6H8sSgS5WmmdcG7UT7t8LFIGbIxJ9nzEOH5uNiZhW
FLQKdaNIHckEdF1E8cDfjDfbvrSENvLCq2NNFOYh5M0dDrXy5gmh10de12T5VmV0eC9WlcT2M1wK
jdSKZWUZwzivVN3xrZ4EUM3DIcv1rWVYeKSF8hp7SJbEhT52rtHwuUI1jxGvHF7XtVHtzXWUV8oK
dbPTuVuZkVA8LQEKeSuHnndNj8AGcmCxDp0yGiA==
ashurafirdaus@Ashura-Firdaus:/mnt/c/Users/zordl$
```

2. Copy the secret and go to the jwt.io website to and choose the JWT Encoder and fill the value such as below.

Header: Algorithm & Token Type

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload: Data

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

Sign JWT: Secret

fill with your secret

The screenshot shows the JWT Debugger interface. On the left, there are three main sections: 'HEADER: ALGORITHM & TOKEN TYPE', 'PAYLOAD: DATA', and 'SIGN JWT: SECRET'. The 'Header' section contains a JSON object with 'alg': 'HS256' and 'typ': 'JWT'. The 'Payload' section contains a JSON object with 'sub': '1486200124', 'name': 'Muhamad Firdaus', 'admin': true, 'lat': 1748718860, and 'exp': 1780246800. The 'SIGN JWT: SECRET' section contains a secret key and encoding options. On the right, the 'JSON WEB TOKEN' section displays the generated token: eyJhbGciOiJIUzI1NiBhY2tlbmQKVC30... followed by a long string of characters.

- For the iat, we can use the freeformatter tools below and fill with the date format you'll be using.

<https://www.freeformatter.com/epoch-timestamp-to-date-converter.html>

The screenshot shows the FreeFormatter.com Epoch & Unix Timestamp Converter. On the left, there's a sidebar with various tools like Formatters, Validators, and Converters. The main area has several input fields: 'Current Unix epoch time' (1751120012), 'Current Unix epoch time in milliseconds' (1751119991781), 'Convert epoch timestamp to date' (Epoch in seconds or mills: 1748710800), and 'Convert date to epoch timestamp' (Year: 2025, Month: 06 - June, Date: 01, Hours: 00, Minutes: 00, Seconds: 00, Time zone: Local). A red arrow points to the 'Convert' button in the 'Convert date to epoch timestamp' section.

4. Open up AWS console and proceed to the lambda section to create a new function.

The screenshot shows the AWS Lambda Functions list page. On the left, there's a sidebar with navigation links like Dashboard, Applications, Functions, Additional resources, and Related AWS resources. The main area displays a table titled 'Functions (2)'. The table has columns for Function name, Description, Package type, Runtime, and Last modified. Two functions are listed: 'dausfunction1' and 'dausfunction2', both created 2 days ago and using Python 3.11. At the top right of the table, there's a 'Create function' button, which is highlighted with a large red arrow pointing towards it.

5. Fill out the new function to set it up as our lambda authorizer.

The screenshot shows the 'Create function' wizard. The first step, 'Choose one of the following options to create your function.', has three options: 'Author from scratch' (selected and highlighted with a red arrow), 'Use a blueprint' (unselected), and 'Container image' (unselected). Below this, there are sections for 'Basic information', 'Runtime', 'Architecture', 'Permissions', and 'Additional configurations'. At the bottom right of the wizard, there are 'Cancel' and 'Create function' buttons.

- Create a new directory to set up as a zip file that we will be using later and install the PyJWT library inside of it (Your directory will look empty at first).

```
pip install PyJWT -t .
```

```
File Edit Selection View Go Run Terminal Help
EXPLORER dausAuthorizer.py U x client_testpy U
OPEN EDITORS
dausAuthorizer.py Part5\lambdabs-auth... U
MAMIDA-APIGW-AND-MONITOR
client_testpy Part5\lambdabs-auth... U
> git
> application
> Docs
> Part1
> Part3
> Part5
lambda-authorizer package
> jwt
> PyJWT-2.10.1.dist-info
dausAuthorizer-old.py
dausAuthorizer.py
client_testpy
lambda-authorizer.zip
.glgignore
README.md
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS CODE REFERENCE LOG
C:\Users\zordi\OneDrive\Documents\Project\lambda-api-gw-and-monitor\Part5\lambda-authorizer-package>pip install PyJWT -t .
cmd + v
File Edit Selection View Go Run Terminal Help
EXPLORER dausAuthorizer.py U x client_testpy U
OPEN EDITORS
dausAuthorizer.py Part5\lambdabs-auth... U
MAMIDA-APIGW-AND-MONITOR
client_testpy Part5\lambdabs-auth... U
> git
> application
> Docs
> Part1
> Part3
> Part5
lambda-authorizer package
> jwt
> PyJWT-2.10.1.dist-info
dausAuthorizer-old.py
dausAuthorizer.py
client_testpy
lambda-authorizer.zip
.glgignore
README.md
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS CODE REFERENCE LOG
C:\Users\zordi\OneDrive\Documents\Project\lambda-api-gw-and-monitor\Part5\lambda-authorizer-package>pip install PyJWT -t .

```

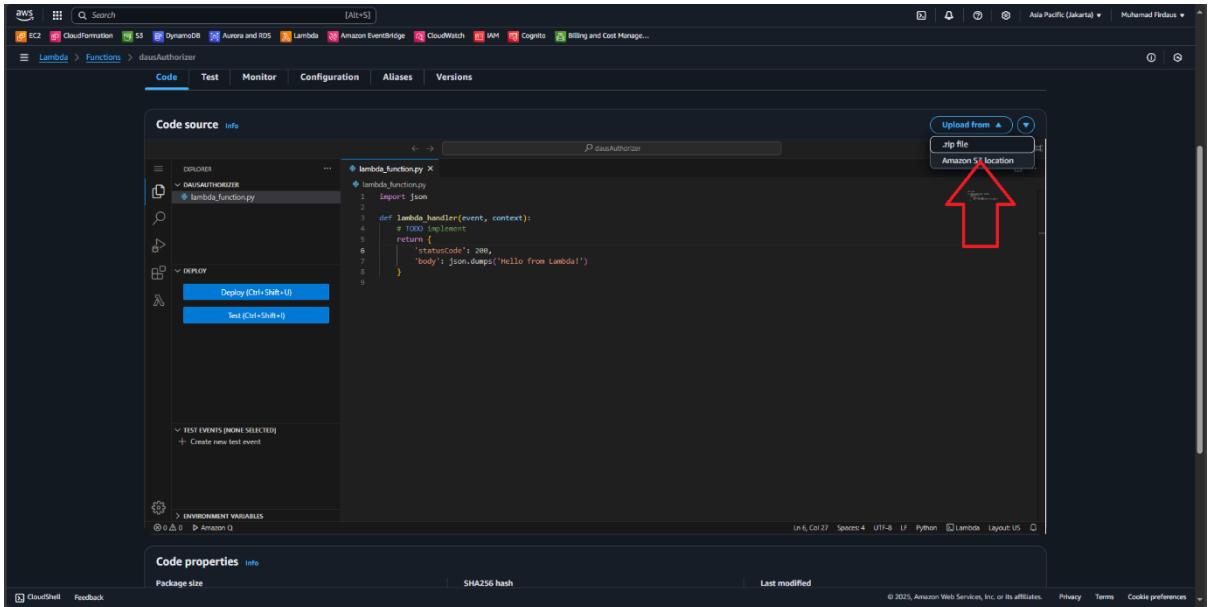
- After the library has show up inside the directory, we proceed to zip all the directories using below command.

```
zip -r ..\lambda-authorizer.zip .
```

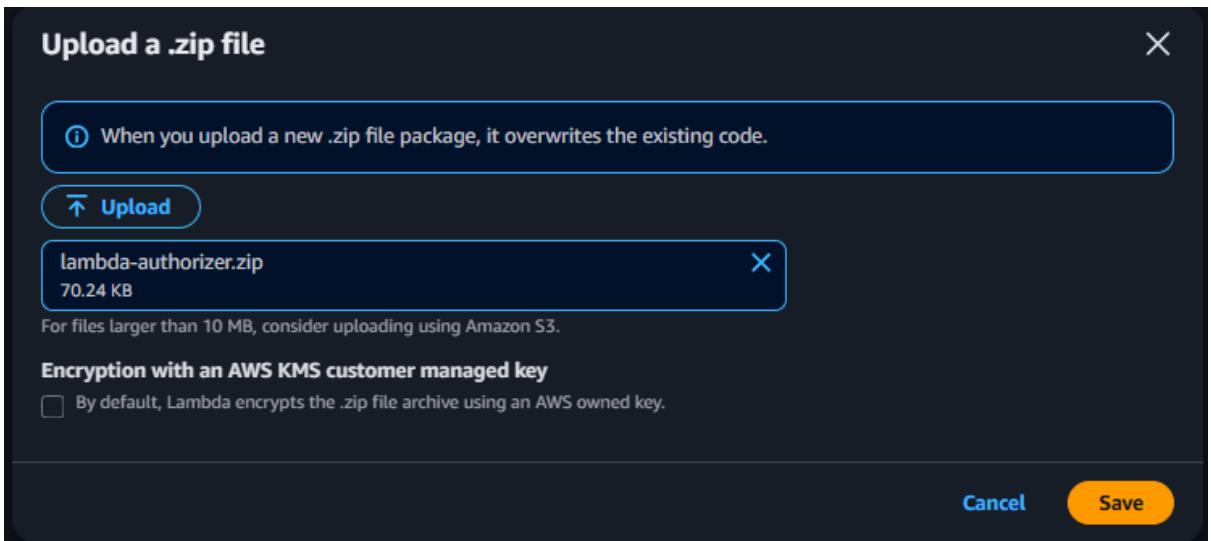
```
File Edit Selection View Go Run Terminal Help
EXPLORER dausAuthorizer.py U x client_testpy U
OPEN EDITORS
dausAuthorizer.py Part5\lambdabs-auth... U
MAMIDA-APIGW-AND-MONITOR
client_testpy Part5\lambdabs-auth... U
> git
> application
> Docs
> Part1
> Part3
> Part5
lambda-authorizer package
> jwt
> PyJWT-2.10.1.dist-info
dausAuthorizer-old.py
dausAuthorizer.py
client_testpy
lambda-authorizer.zip
.glgignore
README.md
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS CODE REFERENCE LOG
C:\Users\zordi\OneDrive\Documents\Project\lambda-api-gw-and-monitor\Part5\lambda-authorizer-package>zip -r ..\lambda-authorizer.zip .
cmd + v
File Edit Selection View Go Run Terminal Help
EXPLORER dausAuthorizer.py U x client_testpy U
OPEN EDITORS
dausAuthorizer.py Part5\lambdabs-auth... U
MAMIDA-APIGW-AND-MONITOR
client_testpy Part5\lambdabs-auth... U
> git
> application
> Docs
> Part1
> Part3
> Part5
lambda-authorizer package
> jwt
> PyJWT-2.10.1.dist-info
dausAuthorizer-old.py
dausAuthorizer.py
client_testpy
lambda-authorizer.zip
.glgignore
README.md
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS CODE REFERENCE LOG
C:\Users\zordi\OneDrive\Documents\Project\lambda-api-gw-and-monitor\Part5\lambda-authorizer-package>zip -r ..\lambda-authorizer.zip .

```

- Back at the newly launched lambda authorizer function, choose to upload from the zip file.



- Confirm that the uploaded file is the zip we just created earlier.

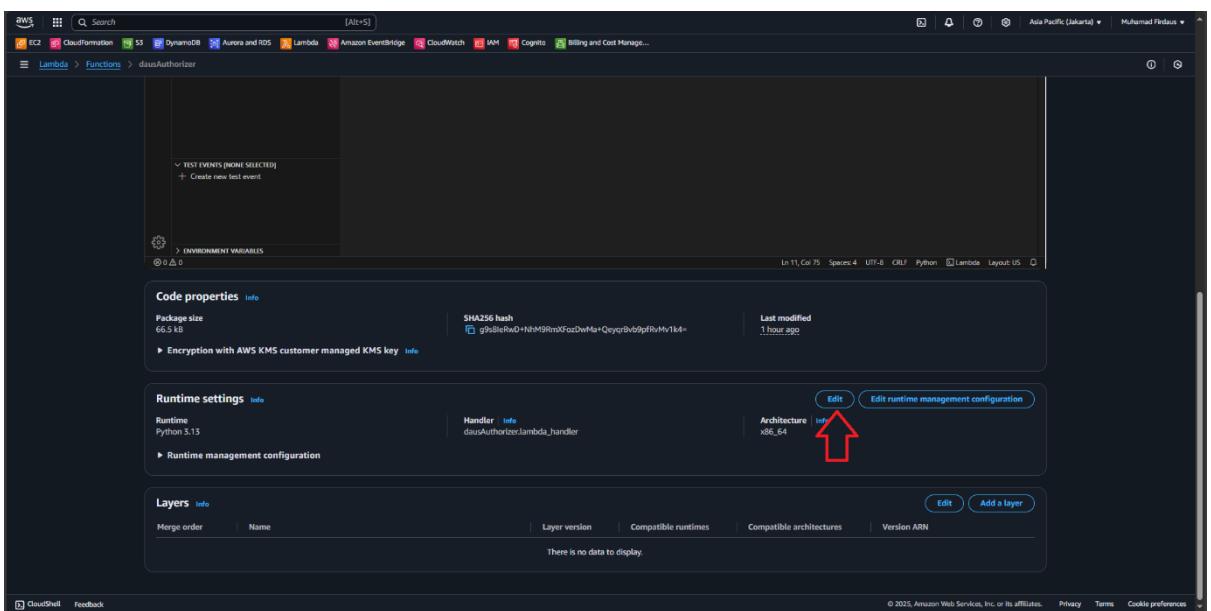


10. The python code then will show up and check if its correct.

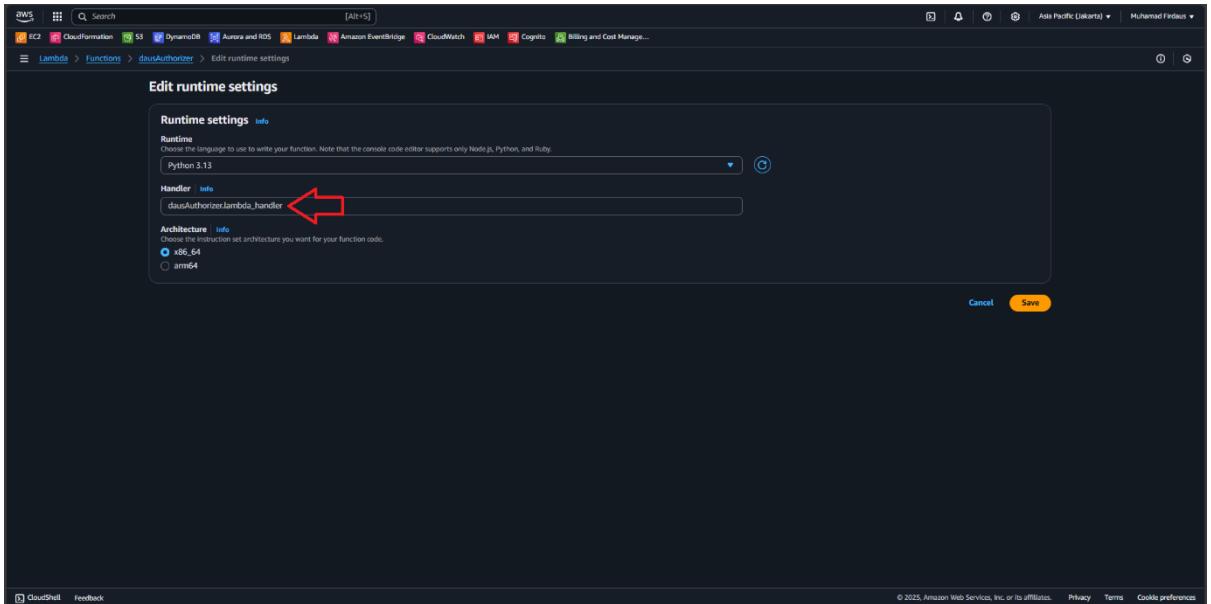
```
dausAuthorizerpy
1 import jwt
2
3 def lambda_handler(event, context):
4     secret = "PmWt-2.10.iotinfo"
5     try:
6         token = event.get('authorizationToken') # <- FIX: seharusnya event langsung
7         decoded_token = jwt.decode(token, secret, algorithms=['HS256'])
8         user_id = decoded_token.get('user_id', 'anonymous')
9     except Exception:
10         raise Exception("Unauthorized") # HARUS pakai raise, bukan return
11
12     return {
13         'principalId': user_id,
14         'policyDocument': {
15             'Version': '2012-10-17',
16             'Statement': [
17                 {
18                     'Action': 'execute-api:Invoke',
19                     'Effect': 'Allow',
20                     'Resource': event['methodArn']
21                 }
22             ],
23             'Context': {
24                 'user_id': user_id
25             }
26         }
27     }
```

Successfully updated the function dausAuthorizer.

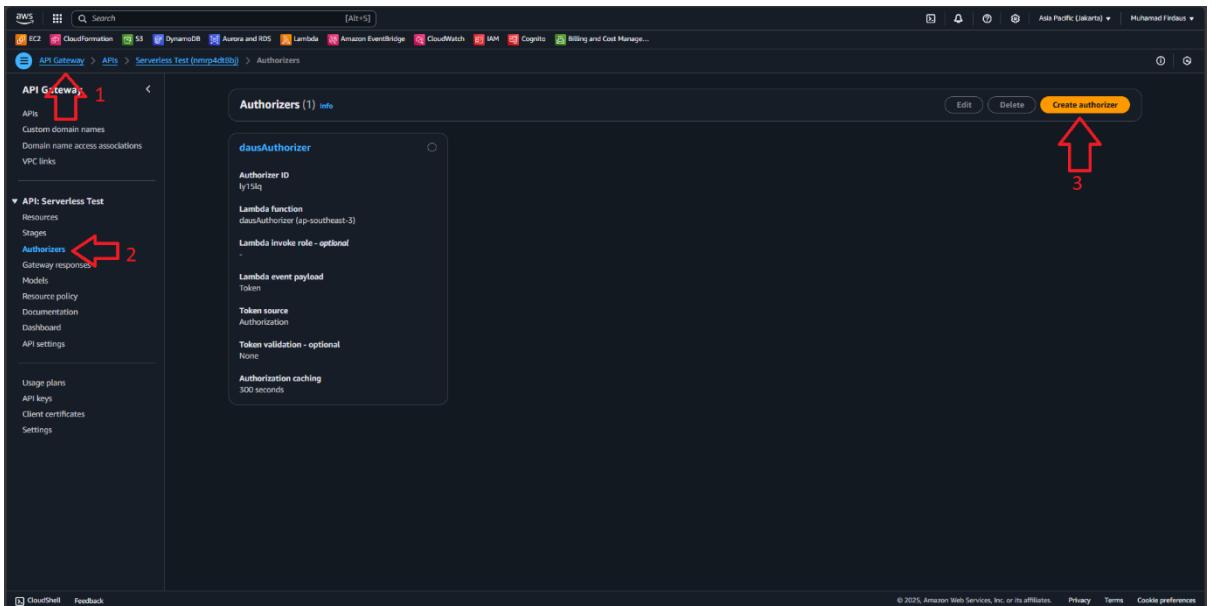
11. Scroll down and choose edit button on the runtime settings.



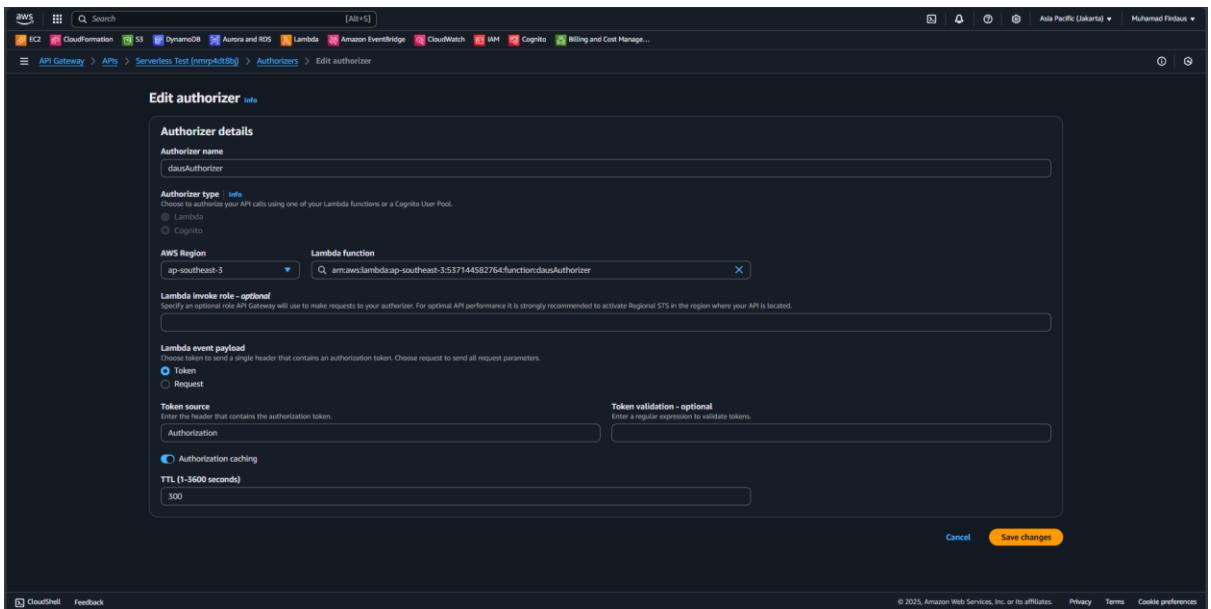
12. Choose the header as your authorizer file name, make sure it is the same to avoid connection error (in this case it is dausAuthorizer).



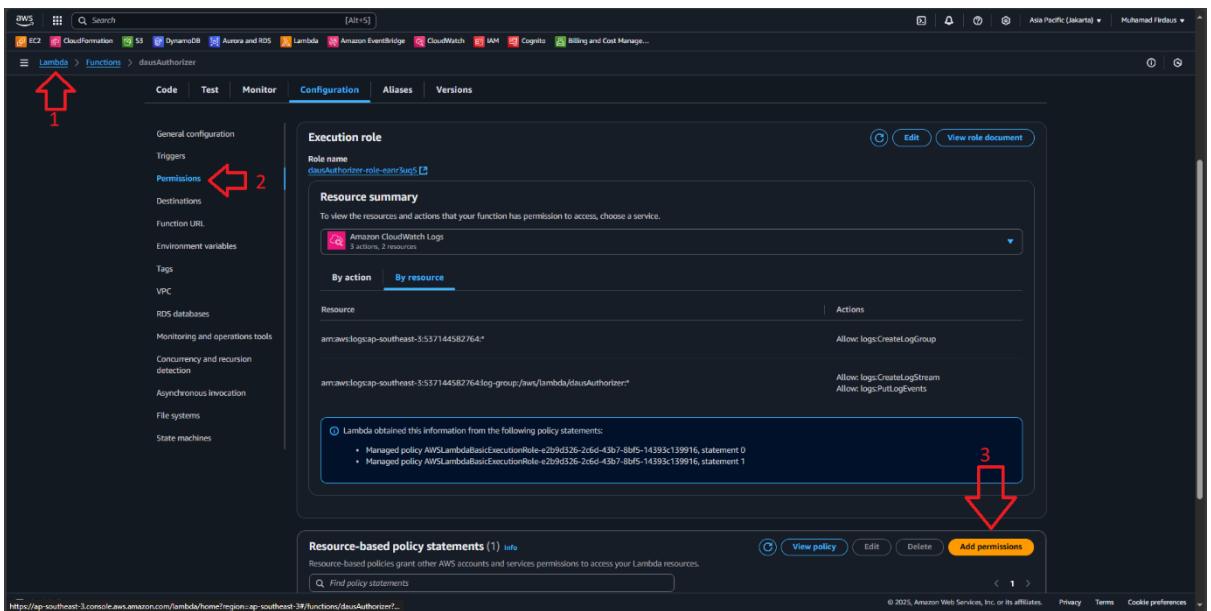
13. Go to the API Gateway and choose the authorize menu, click on the “Create authorizer” button to create a new authorizer.



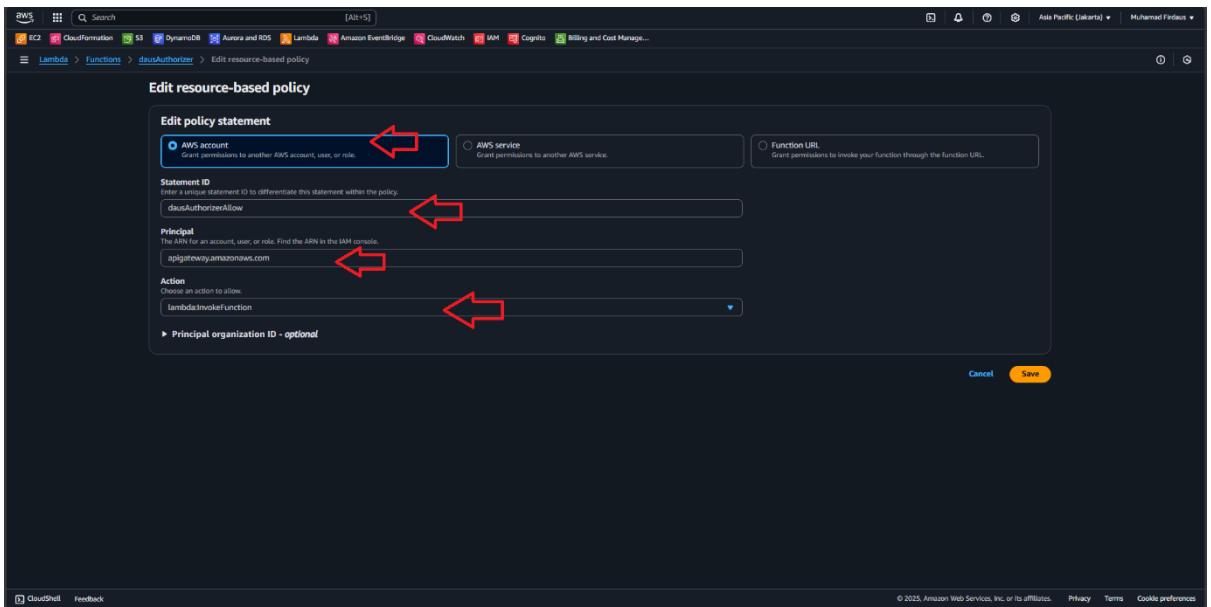
14. At the Edit authorizer menu, fill the details with the same file you created earlier, in my case the name is “dausAuthorizer”, choose the lambda function as the same “dausAuthorizer” and then fill the token source as “Authorization”.



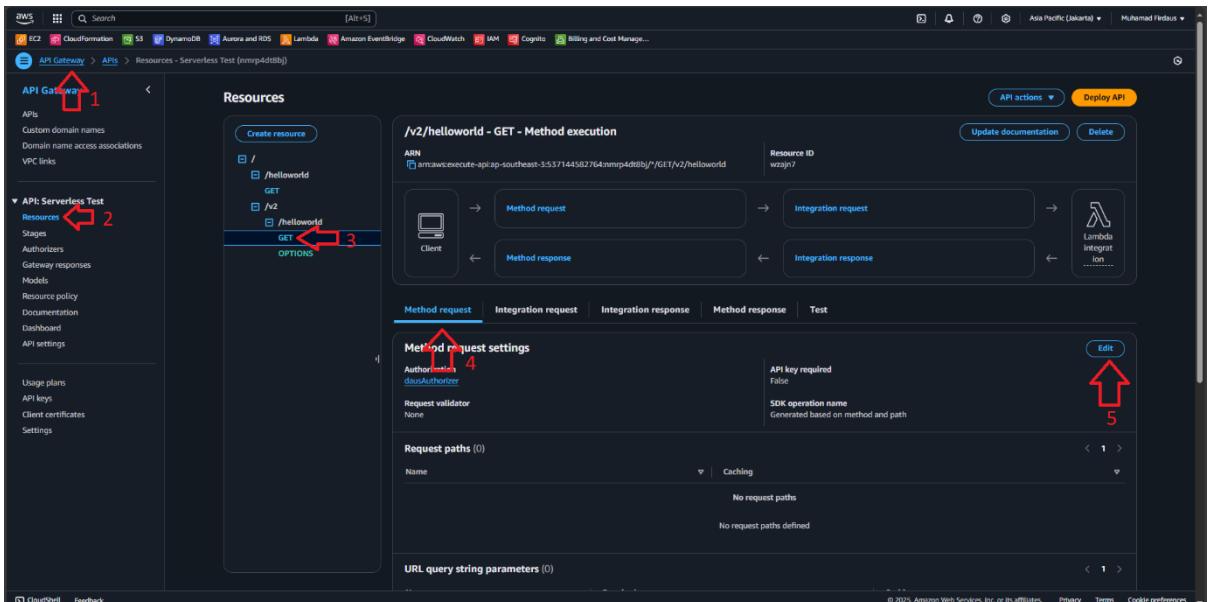
15. Return back to lambda and proceed to the permission section, then click on the “Add Permission” button so we can add a permission for the lambda authorizer.



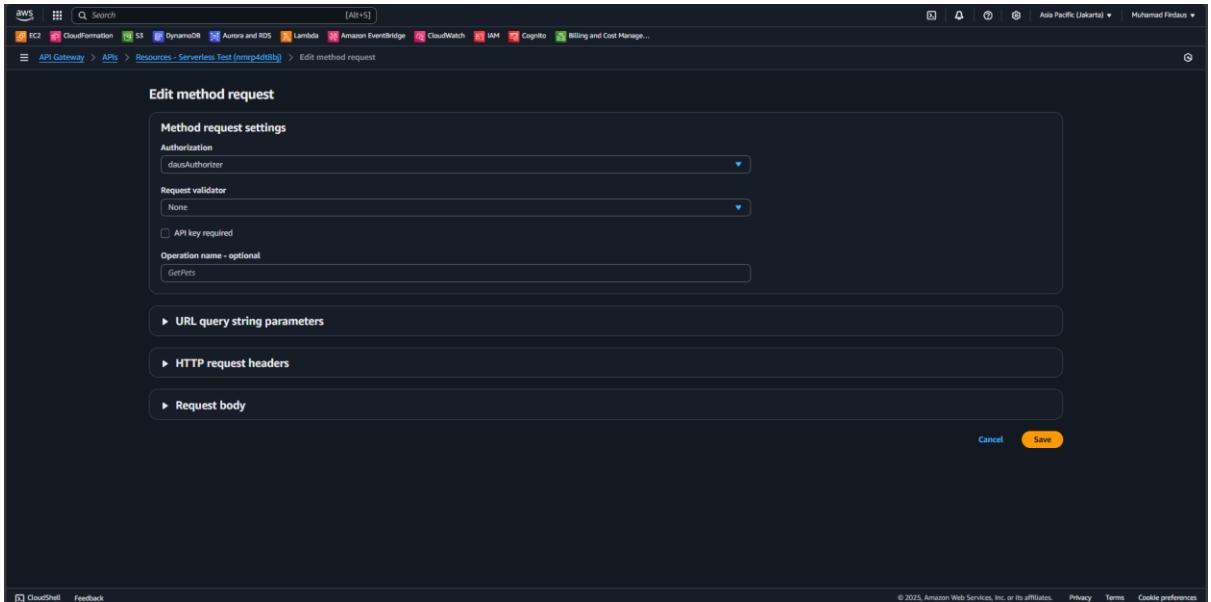
16. On the policy statement configuration, choose 1. AWS account, 2. Statement ID = dausAuthorizerAllow or anything you wish to name it, 3. Set the Principal as “apigateway.amazonaws.com” (Because API Gateway needs permission to invoke the Lambda function on our behalf. We grant API Gateway the permission to invoke the Lambda Authorizer function, since it needs to validate tokens before forwarding requests to the backend.) and the action is lambdaInvokeFunction and click save.



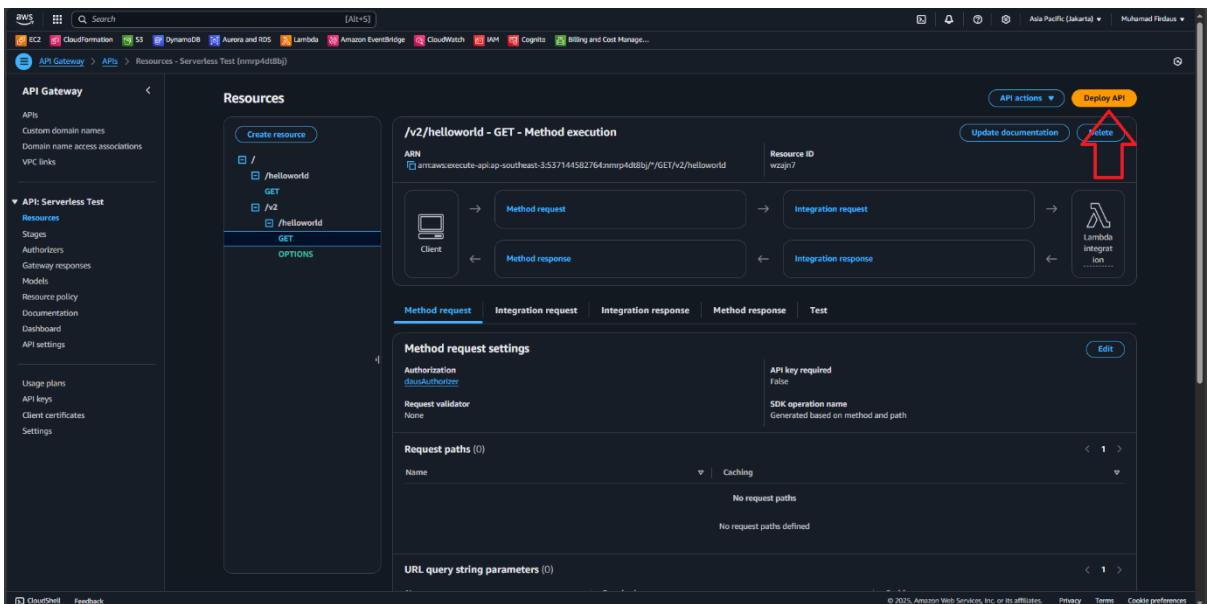
17. Then return to the API Gateway, click on resource and choose the v2 GET menu, choose method request and edit this section.



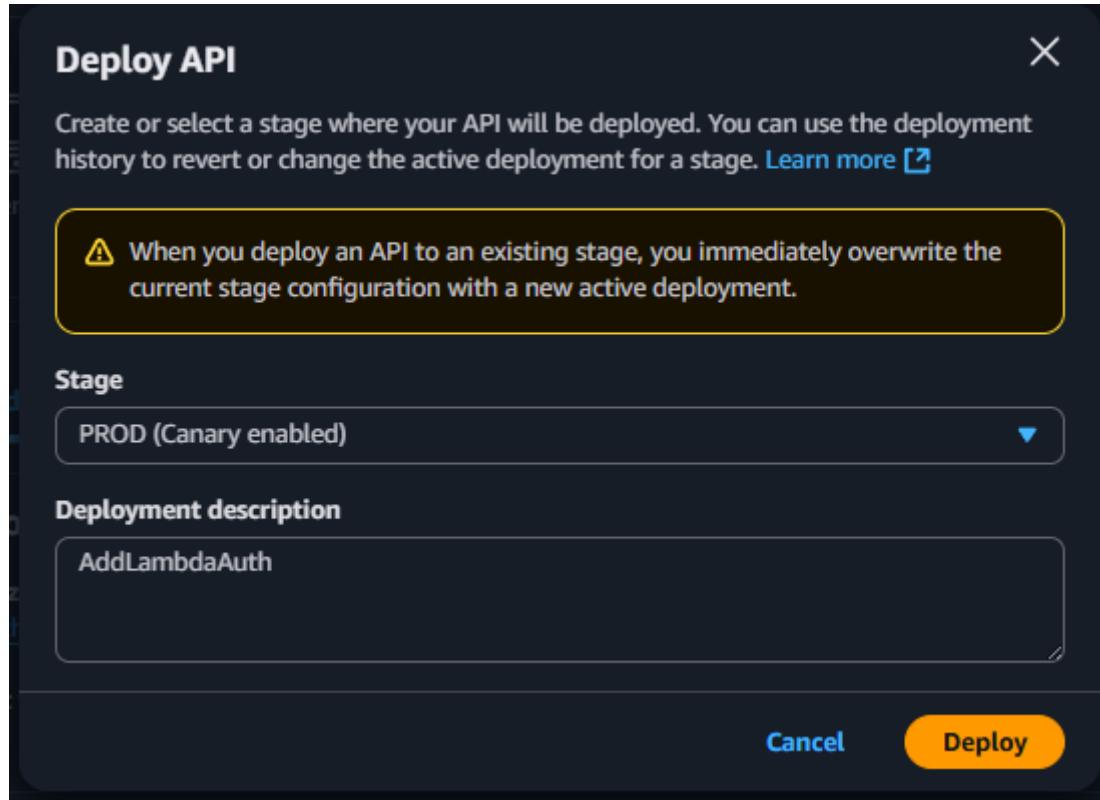
18. On the edit method request, fill out the details with the set up we created earlier making sure its matching. In my case it is Authorization = dausAuthorizer and click save.



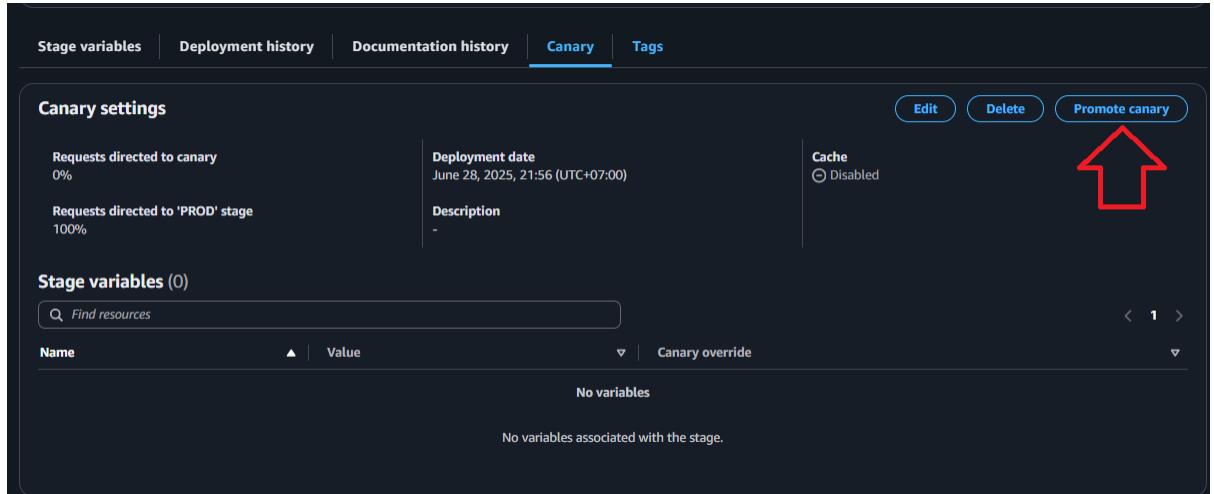
19. Click on Deploy API to deploy our changes.



20. Choose the stage, in my case it is still on the PROD with canary enabled, and add the description to make sure you know which version you just deployed.



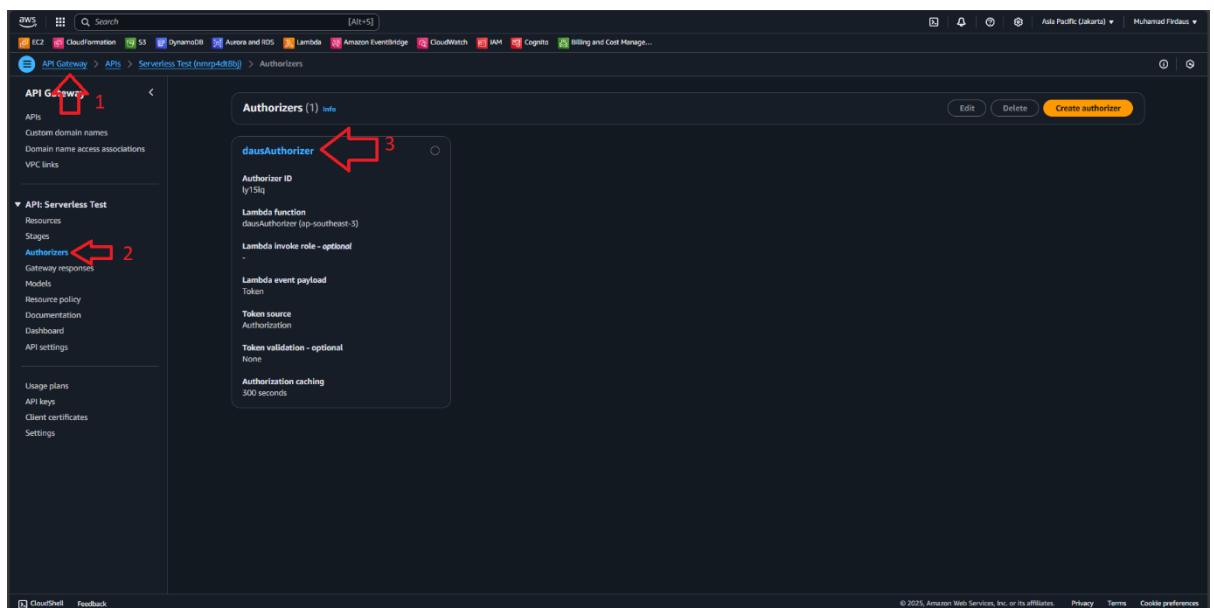
21. After successful deployment, as usual since we use canary we promote this canary to the PROD before it can be invoked.



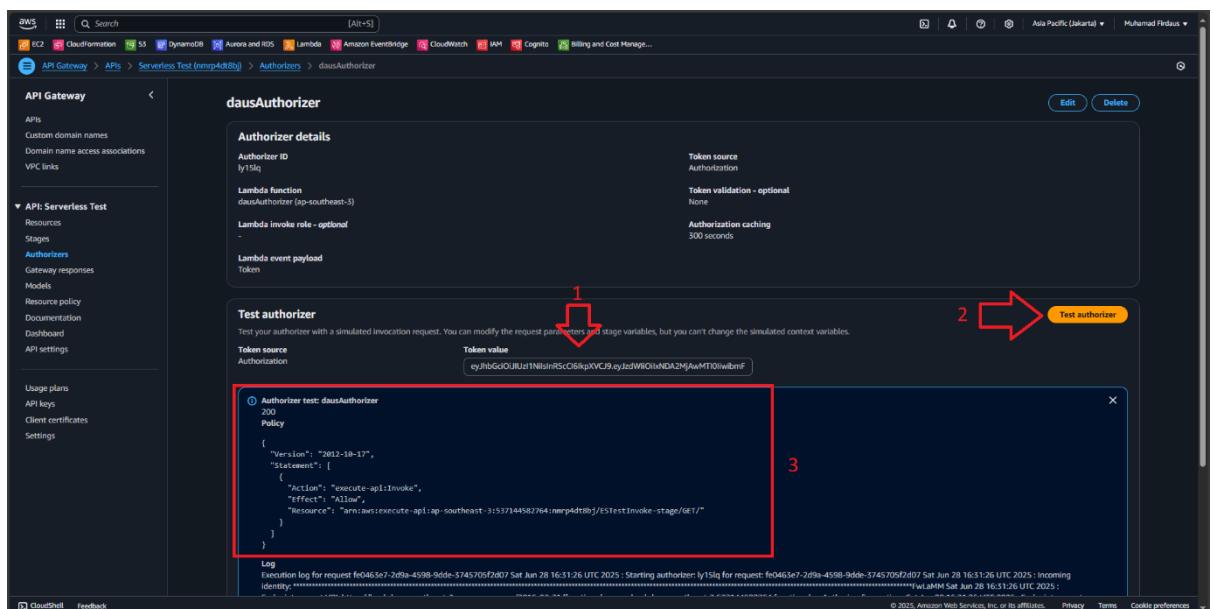
22. Promote the canary to proceed.



23. Then return to the API Gateway to test our authorization by going to the authorizer and click on the name of the authorizer we created earlier.



24. Put the JWT token you get from the secret you encoded on the JWT.io website and test.



25. Or we can use a python code that used the JWT token to request to the API Gateway, if successful you will see the 200 message with the greetings.

```
python client_test.py
```

The screenshot shows a terminal window in the VS Code interface. The terminal tab is selected at the bottom. The command entered is:

```
C:\Users\firdaus\OneDrive\Documents\Projects\lambda-api-and-monitoring\Part5\lambda-authorizer-package>python client_test.py
```

The output shows the command being run and the response from the API:

```
200
{"statusCode": 200, "body": "Greetings from Firdaus!"}
```

A red arrow points from the text "Greetings from Firdaus!" in the terminal output towards the right side of the image, indicating the successful response.