Explanation for Each Step in Your EC2 Automation Task Document.

INTRODUCTION

This document demonstrates the complete automation of starting and stopping EC2 instances using AWS Lambda and Amazon EventBridge (CloudWatch). Each screenshot represents a specific step in the configuration process, beginning with EC2 instance setup, IAM permissions, Lambda creation, script deployment, and finally configuring CloudWatch rules to trigger the functions automatically.

The goal of this automation is to reduce cost and operational overhead by ensuring EC2 instances run only during required time intervals.

STEP-WISE EXPLANATION USING SCREENSHOT's

1. Creating EC2 Instances (node1 & node2)

The first screenshots show the creation of two EC2 instances named **node1** and **node2**. These instances are used to test the automation. Both instances are launched in the default VPC with basic configuration.

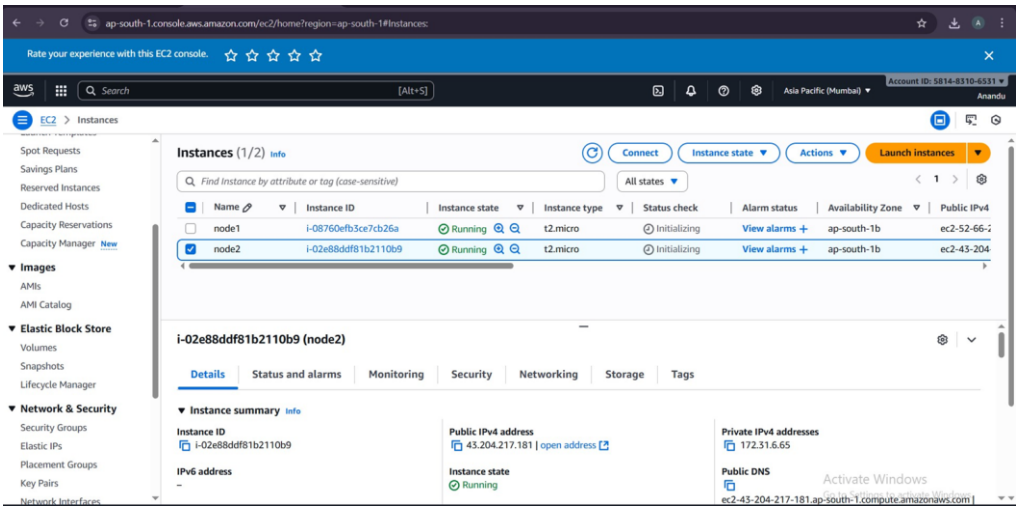

Fig 1: EC2 instances created — node1 and node2.

2. Stopping the Instances Before Automation

Before testing the Lambda start script, the instances are manually stopped.

This ensures that when the Lambda start function runs, the effect is immediately visible — the instances change from "stopped" to "running".

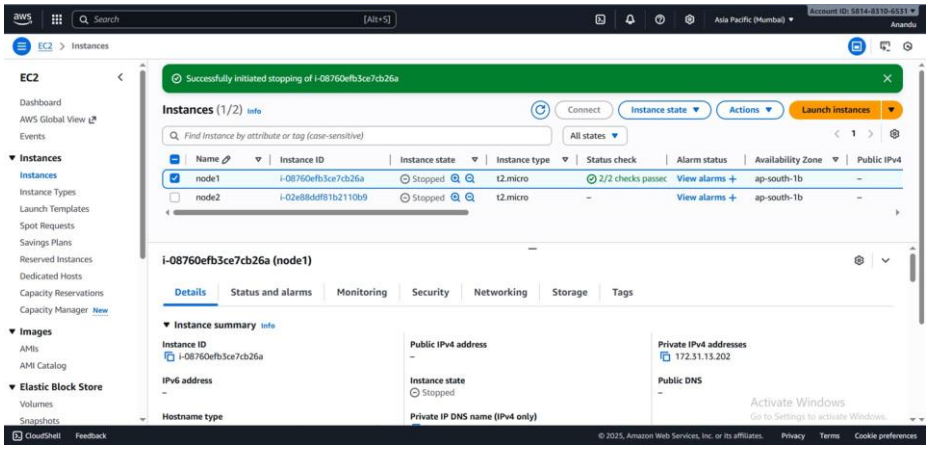

Fig 2: Both instances manually stopped before automation testing.

3. IAM Dashboard – Creating IAM Policy

The next set of screenshots shows:

- Opening the **IAM dashboard**
- Selecting **Policies → Create policy**
- Entering the JSON policy that provides permissions for:
  - Starting EC2 instances
  - Stopping EC2 instances
  - Describing EC2 instances
  - Writing logs to CloudWatch

This policy ensures Lambda has the exact permissions required to control EC2.

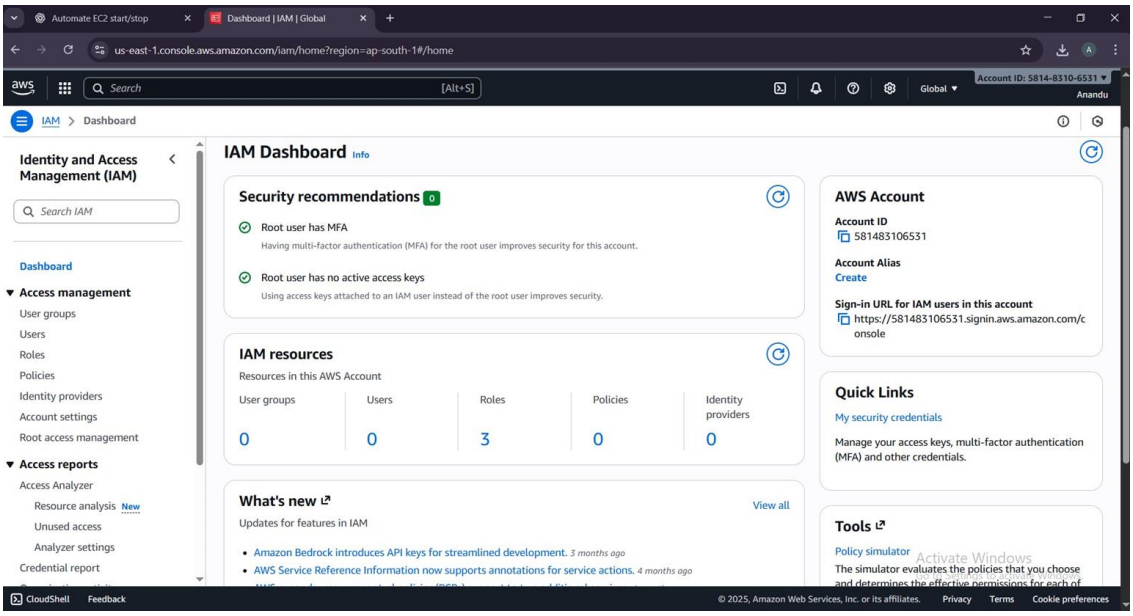

Fig 3: IAM Dashboard — Creating a new IAM Policy.

4. IAM POLICIES Creation (ec2-lambda-policy)

Here, using three screenshots, a new IAM Policies is created.
During the Policy creation :

- Trust entity: **I AM**
- Final POLICES name: **ec2-lambda-policy**

This POLICES will later be assigned to the IAM ROLE.

JSON CODE Final POLICES name: **ec2-lambda-policy**


```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowEC2StartStopDescribeSpecificInstances",
            "Effect": "Allow",
            "Action": [
                "ec2:StartInstances",
                "ec2:StopInstances",
                "ec2:DescribeInstances"
            ],
            "Resource": [
                "arn:aws:ec2:ap-south-1:581483106531:instance/i-08760efb3ce7cb26a",
                "arn:aws:ec2:ap-south-1:581483106531:instance/i-02e88ddf81b2110b9"
            ]
        },
        {
            "Sid": "AllowCloudWatchLogs",
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:ap-south-1:581483106531:*"
        }
    ]
}
```

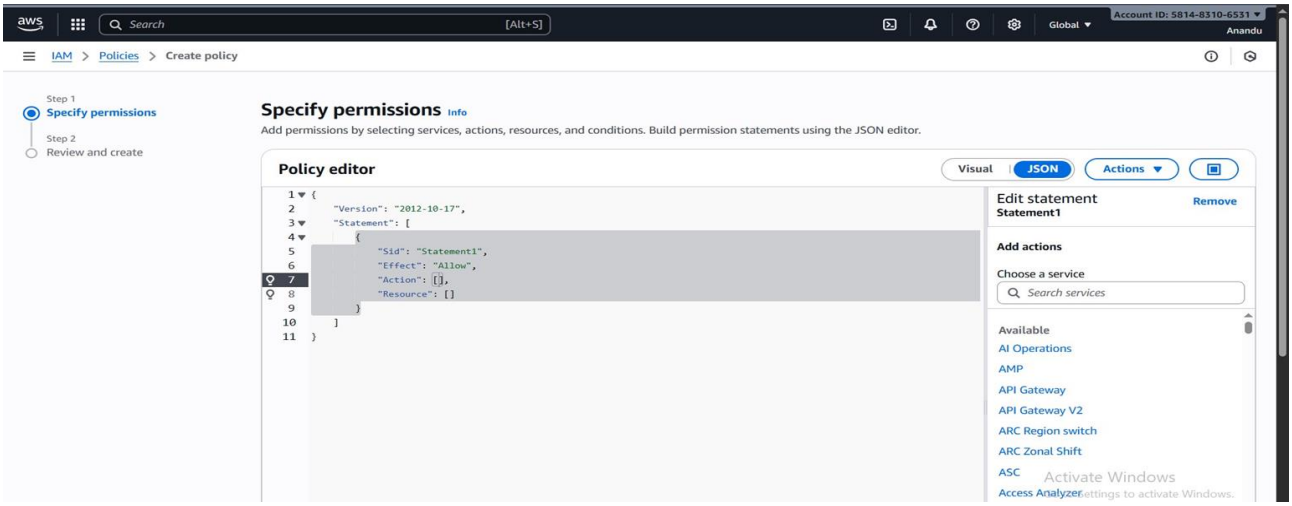

Fig 4: JSON policy added to grant EC2 and CloudWatch Logs permissions.

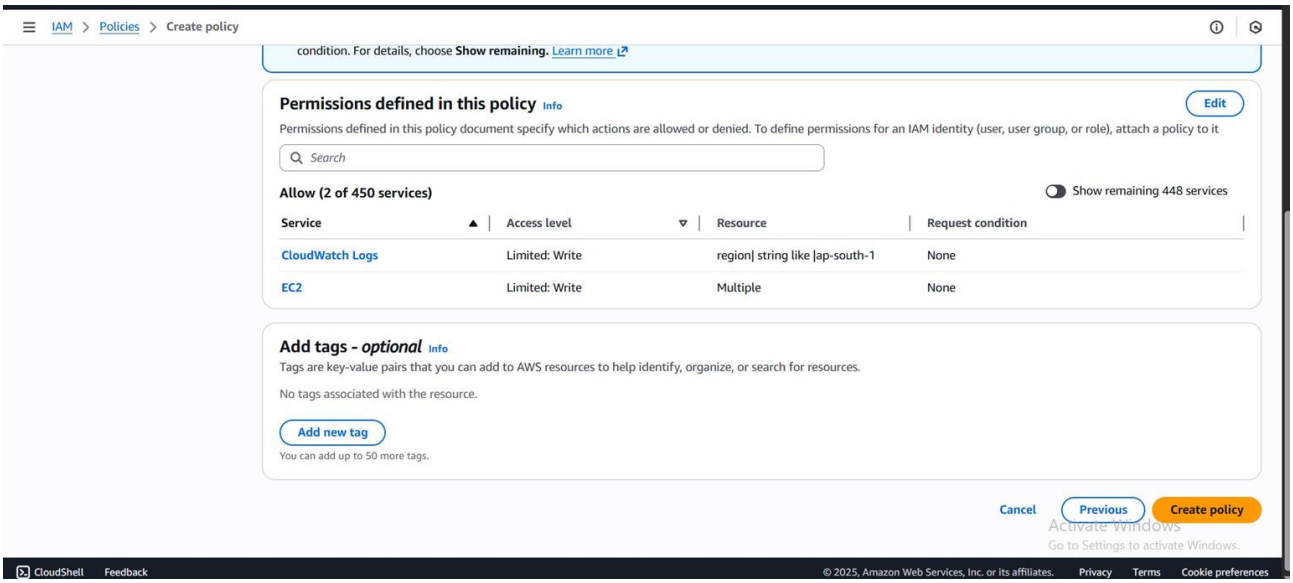

Fig 5: IAM Policy successfully created (ec2-lambda-policy)

5. IAM Role Creation (lambdarole)

Here, using three screenshots, a new IAM role is created.
During the role creation:

- Trust entity: **AWS Lambda**
- Permission attached: The previously created policy

- Final role name: **lambdarole**

This role will later be assigned to the Lambda functions.

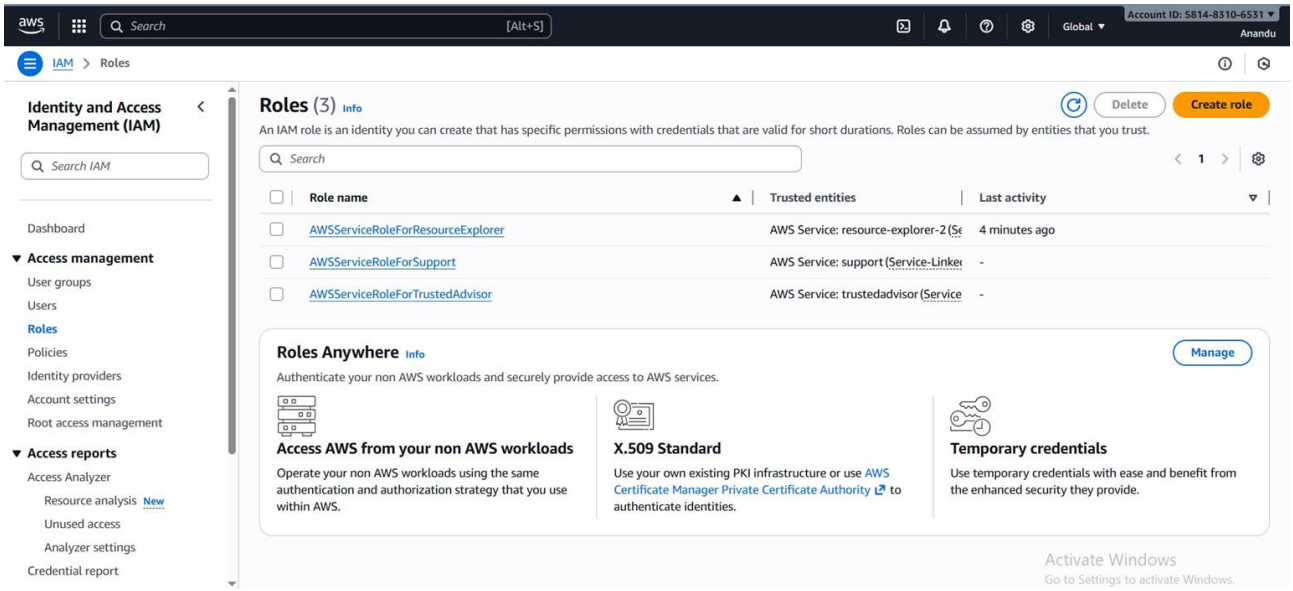

Fig 6: IAM Role creation page — Selecting AWS Lambda as trusted entity.

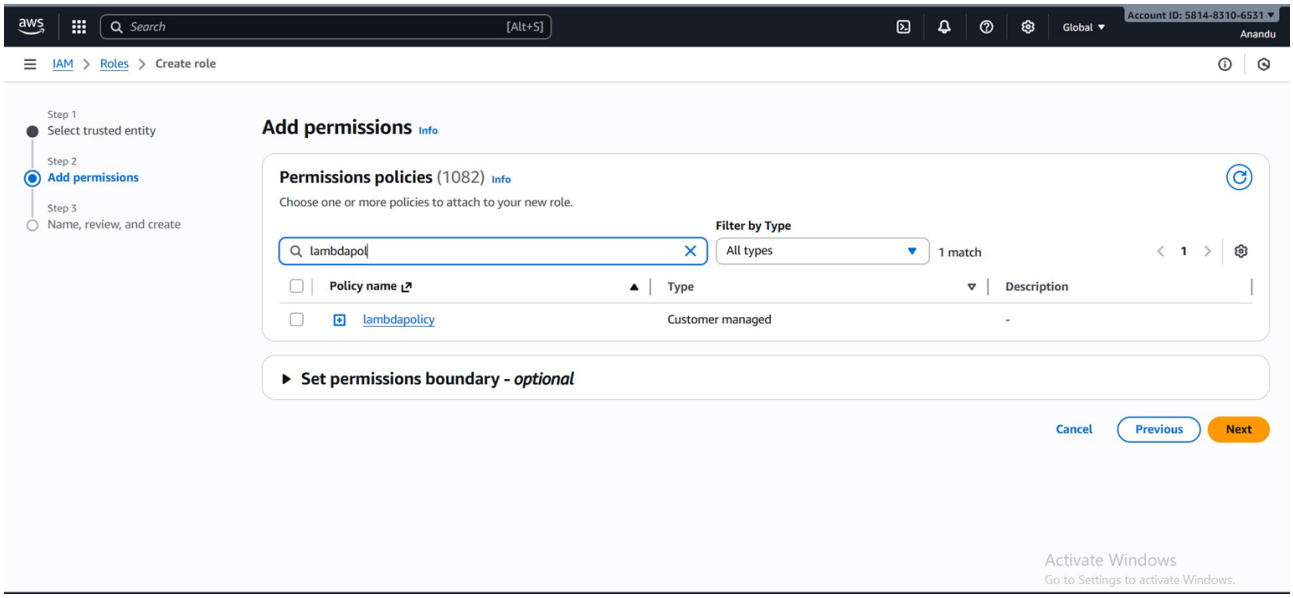

Fig 7: Attaching the custom EC2 policy to the new IAM role.

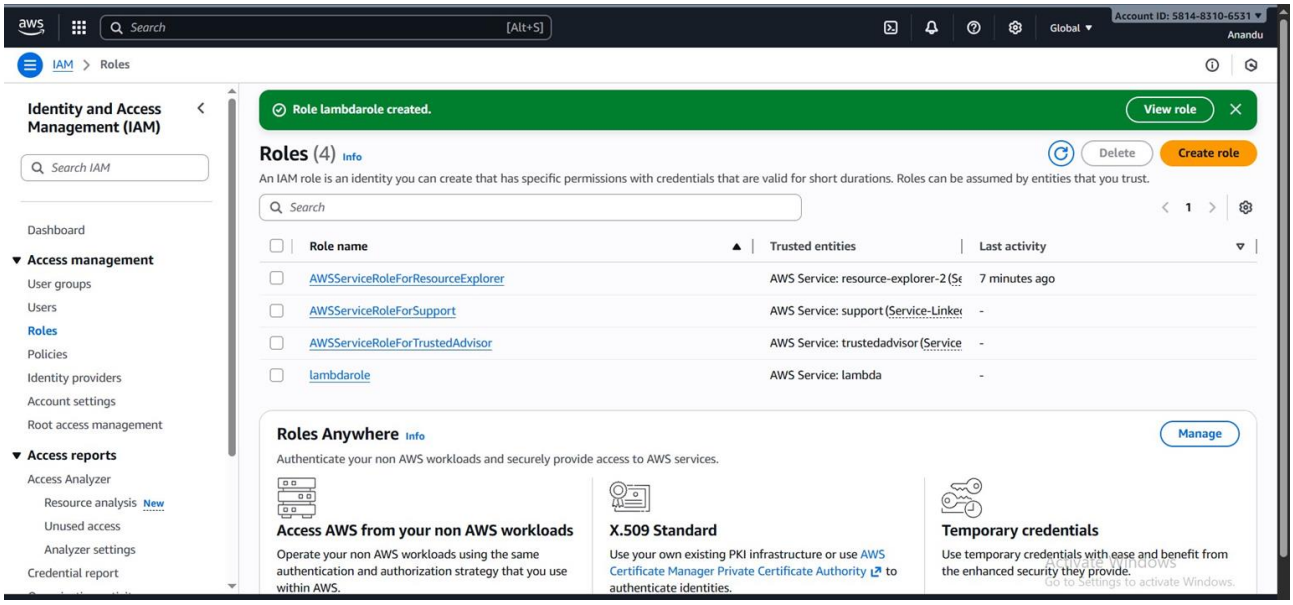

Fig 8: IAM Role successfully created (lambdarole).

6. Navigating to AWS Lambda

The screenshot shows how to access the Lambda service:

- In AWS Console → open **Services**
- Under **Compute** → select **Lambda**

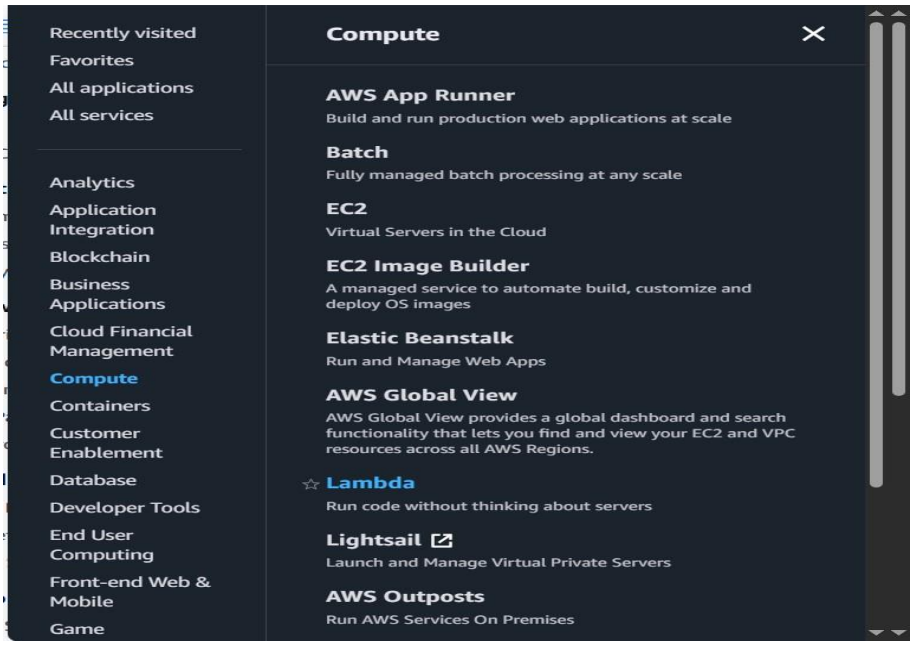

Fig 9: Navigating to AWS Lambda from the Compute section.

7. Lambda Homepage – Selecting "Create Function"

This screenshot shows the Lambda dashboard and the **Create function** button.
This is where new automation functions are defined.

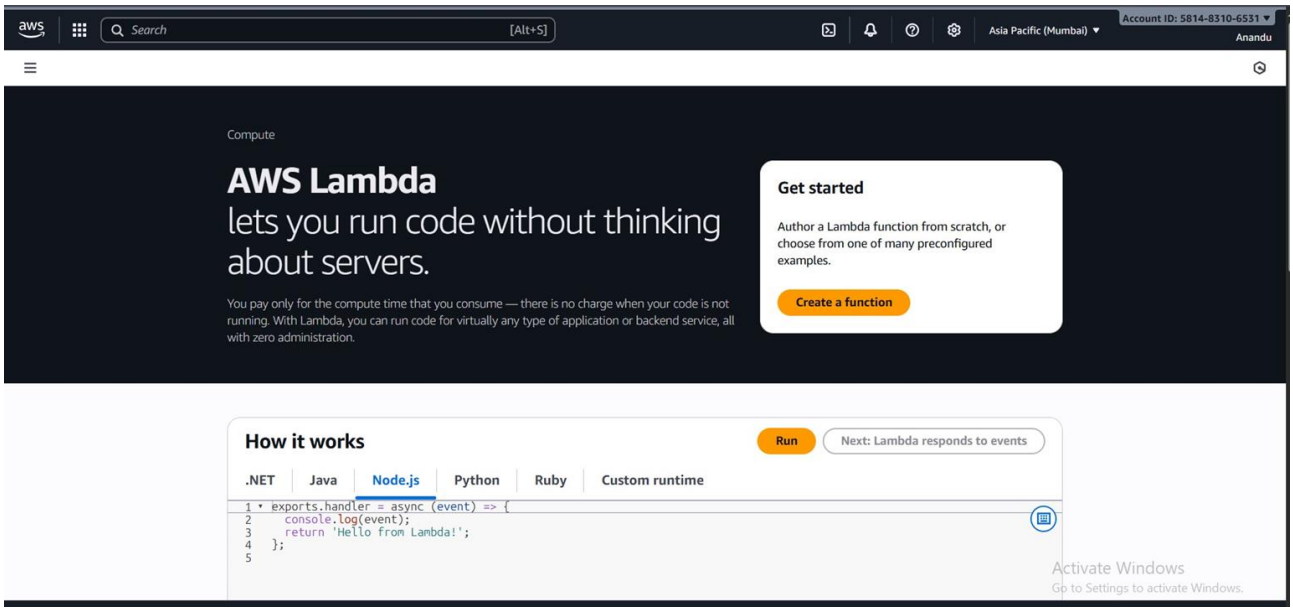

Fig 10: Lambda homepage — Selecting "Create Function

8. Choosing "RUNTIME LANGUAGE AS PYTHON"

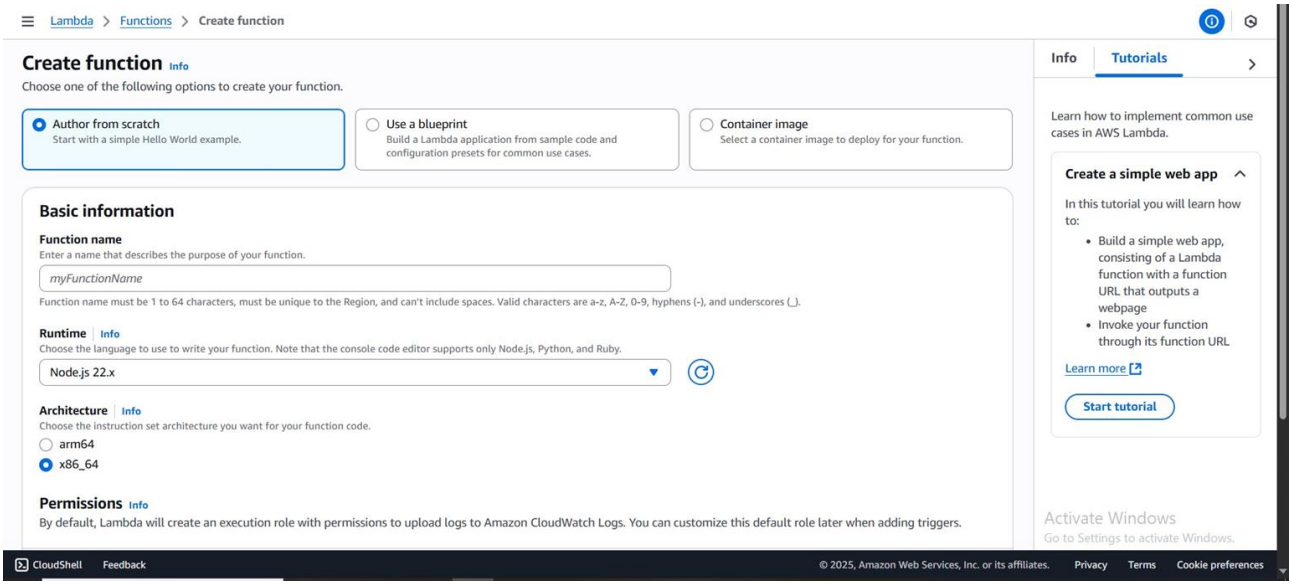

Fig 11: Choosing Python as the runtime language for the Lambda function.

9. Choosing "Use an Existing Role"

While creating the function, the screenshot highlights:

- "Change default execution role"
- Selecting **Use an existing role**

This ensures the Lambda function uses the IAM role **lambdarole**, which already contains the required EC2 permissions.

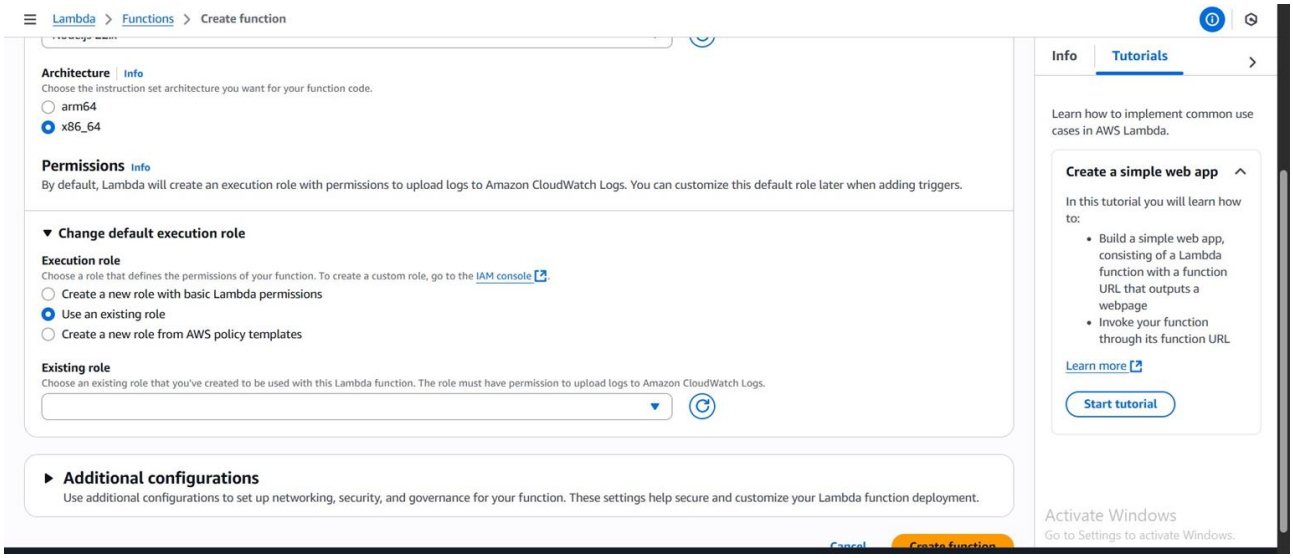

Fig 12: Selecting "Use an existing role" for execution permissions.

10. Selecting the Role ‘lambdarole’

The screenshot here confirms that the execution role chosen for the Lambda function is **lambdarole**.

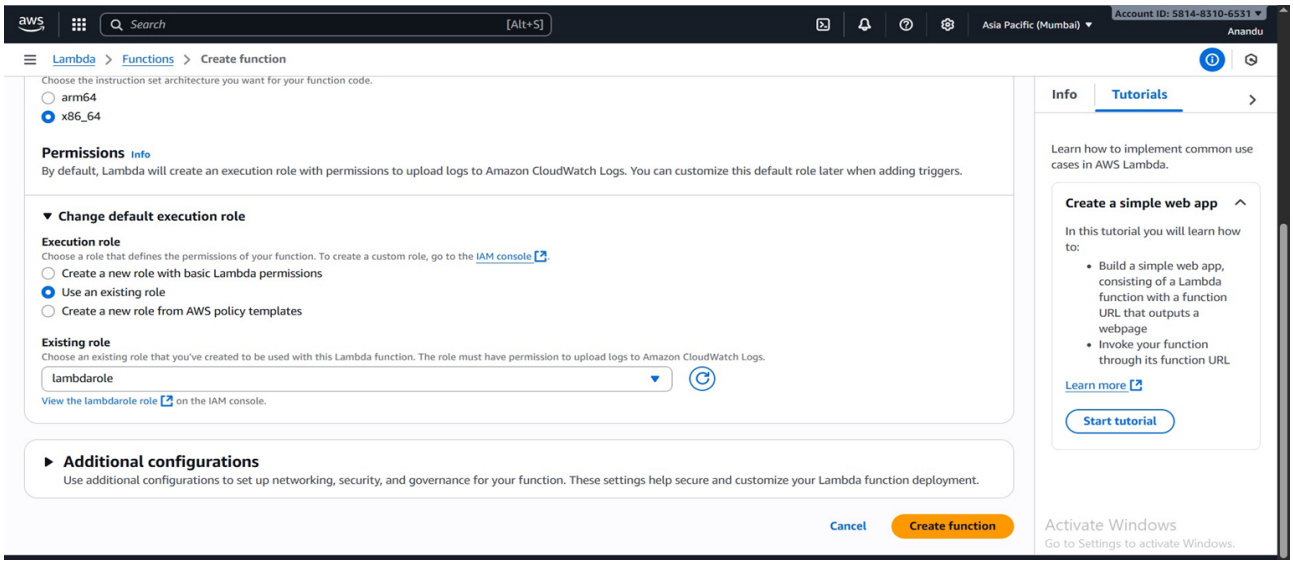

Fig 13: Assigning the IAM role *lambdarole* to the Lambda function.

11. Lambda Function Successfully Created

This image shows the successfully created Lambda function named **lambdafunction**, along with available options to edit its code and configuration.

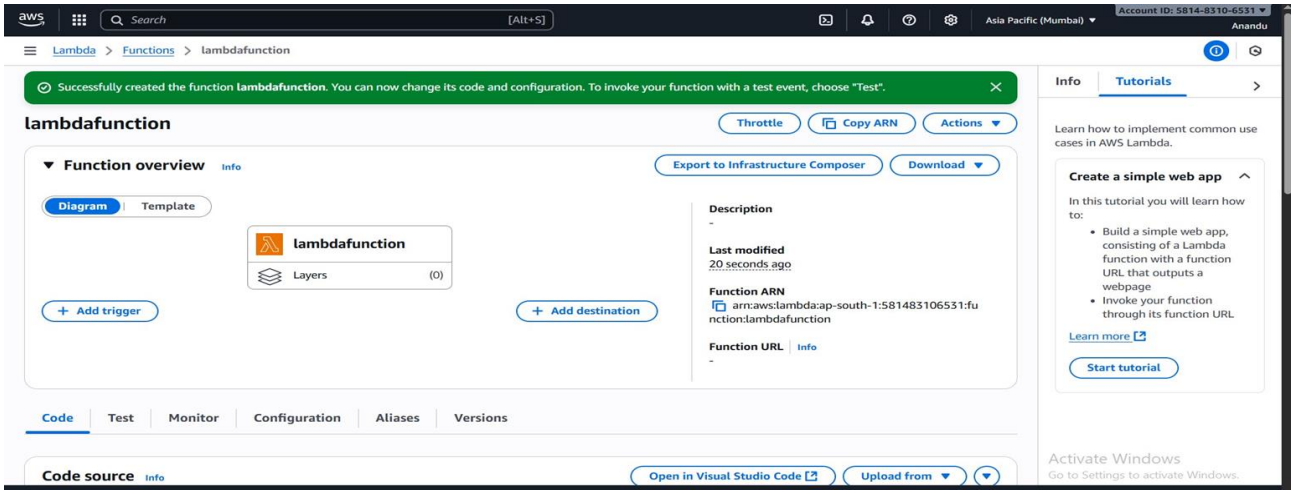

Fig 14: Lambda function created successfully with default configuration.

12. Uploading & Deploying lambdafunction.py

This screenshot shows the Lambda function updated with the Python script (**lambdafunction.py**) responsible for starting the EC2 instances.
After upload, the code is deployed and tested successfully.

```python
import boto3

# EC2 Start Script for Mumbai Region
region = 'ap-south-1'
instances = ['i-08760efb3ce7cb26a', 'i-02e88ddf81b2110b9']
ec2 = boto3.client('ec2', region_name=region)

def lambda_handler(event, context):
    ec2.start_instances(InstanceIds=instances)
    print('✅ Started your instances: ' + str(instances))
```

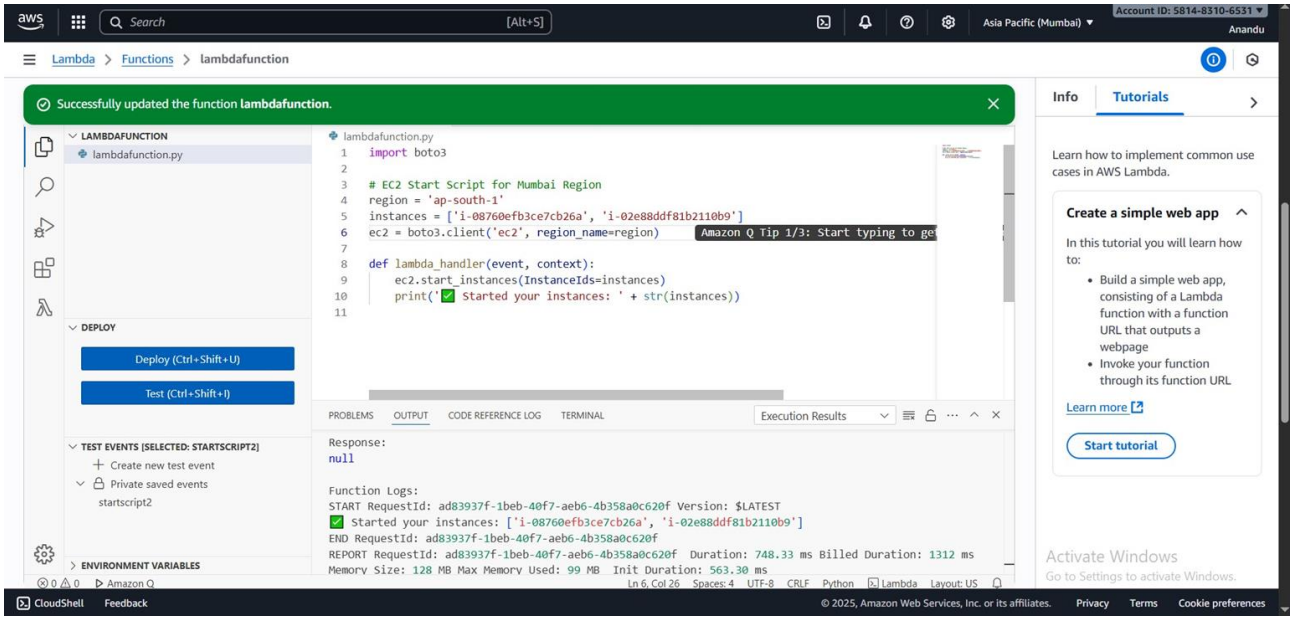

Fig 15: Lambda function updated with lambdafunction.py and deployed successfully.

13. Runtime Settings

The screenshot shows:

- Runtime: **Python 3.13**

- Handler: **lambda_function.lambda_handler**

This verifies that Lambda will execute the correct entry function.

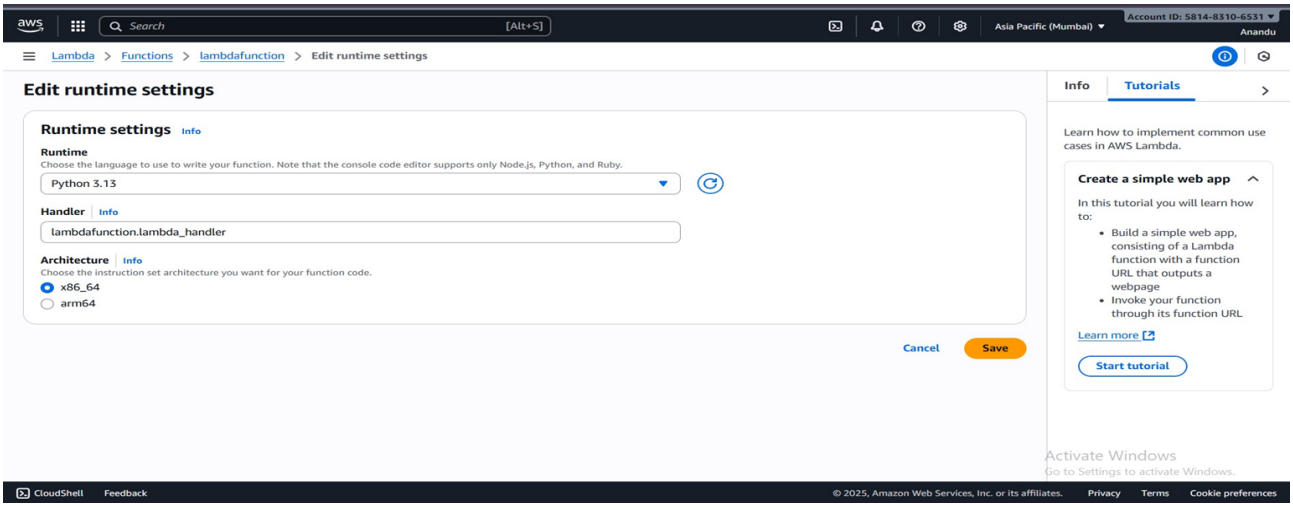

Fig 16: Runtime settings — Python 3.13 and correct Lambda handler configured.

14. Test Result – EC2 Instance Started

This screenshot shows that the previously stopped instance (node1/node2) is now running. This confirms the Lambda start script works correctly.

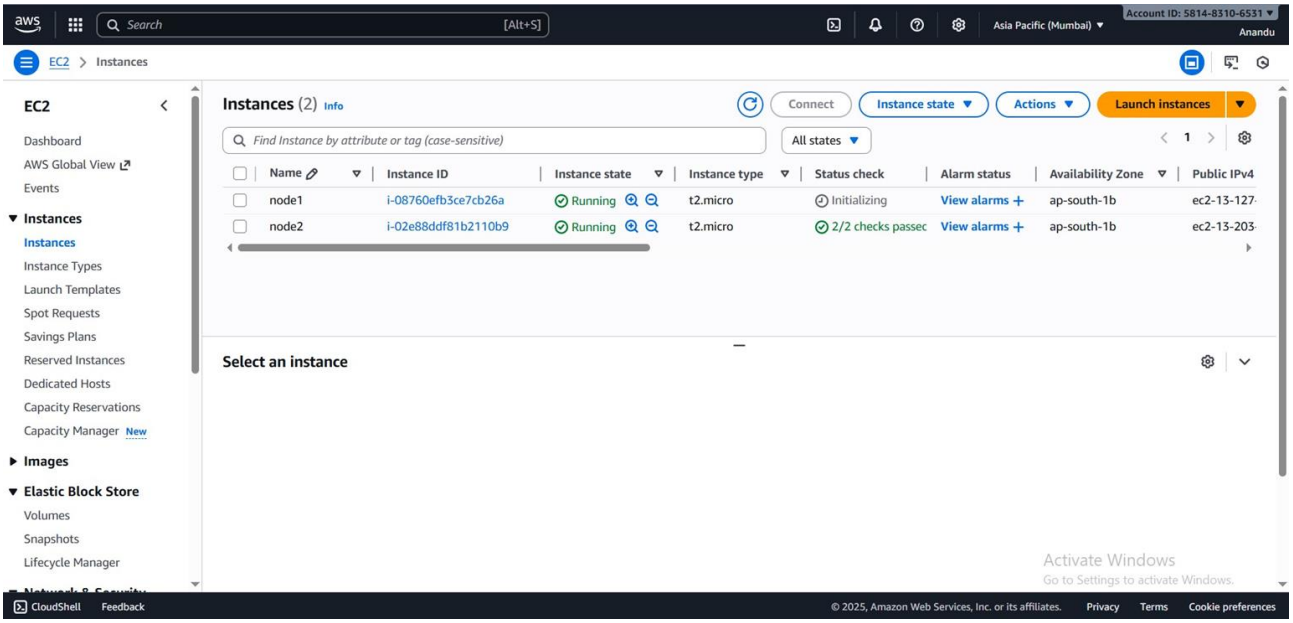

Fig 17: Test result — EC2 instance started successfully using start script.

15. Deploying EC2StopScript

Next screenshot shows uploading and deploying **EC2StopScript.py**, which stops the EC2 instances.

```
# EC2 Stop Script

import boto3

region = 'ap-south-1'
instances = ['i-08760efb3ce7cb26a', 'i-02e88ddf81b2110b9']
ec2 = boto3.client('ec2', region_name=region)

def lambda_handler(event, context):
    ec2.stop_instances(InstanceIds=instances)
    print("Stopped your instances: " + str(instances))
```

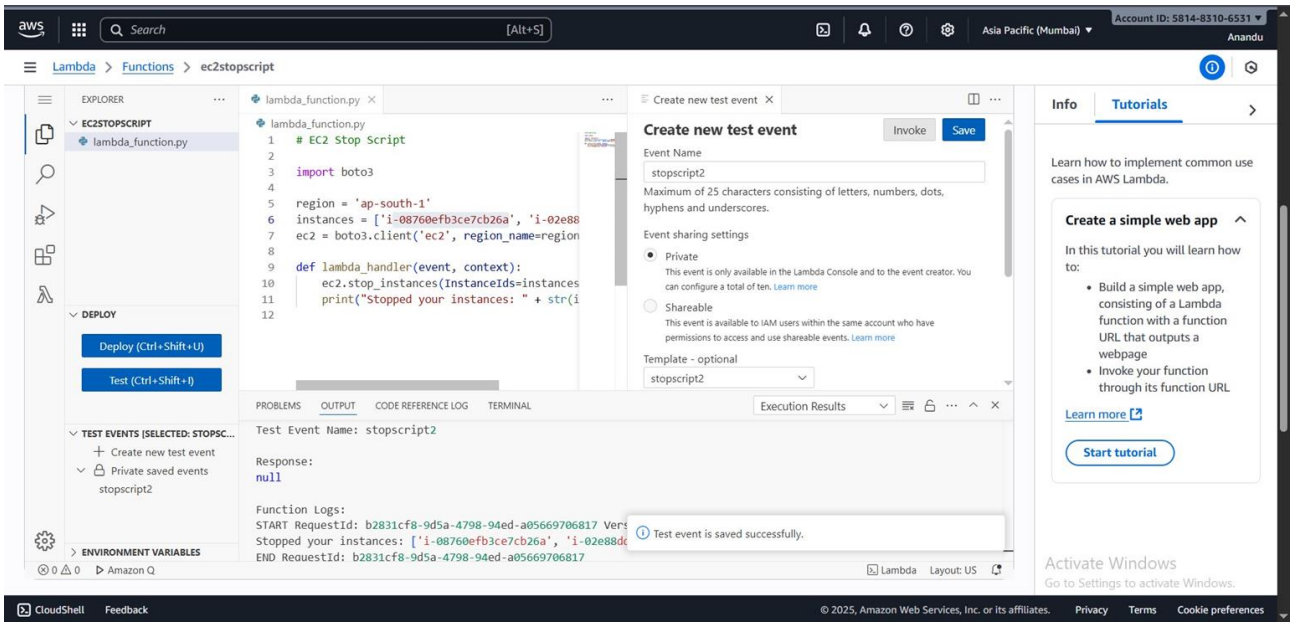

Fig 18: Deploying the EC2StopScript.py into the second Lambda function.

16. Test Result – EC2 Instance Stopped

Here, the EC2 instances transition from “running” to “stopped” due to the stop script deployment.

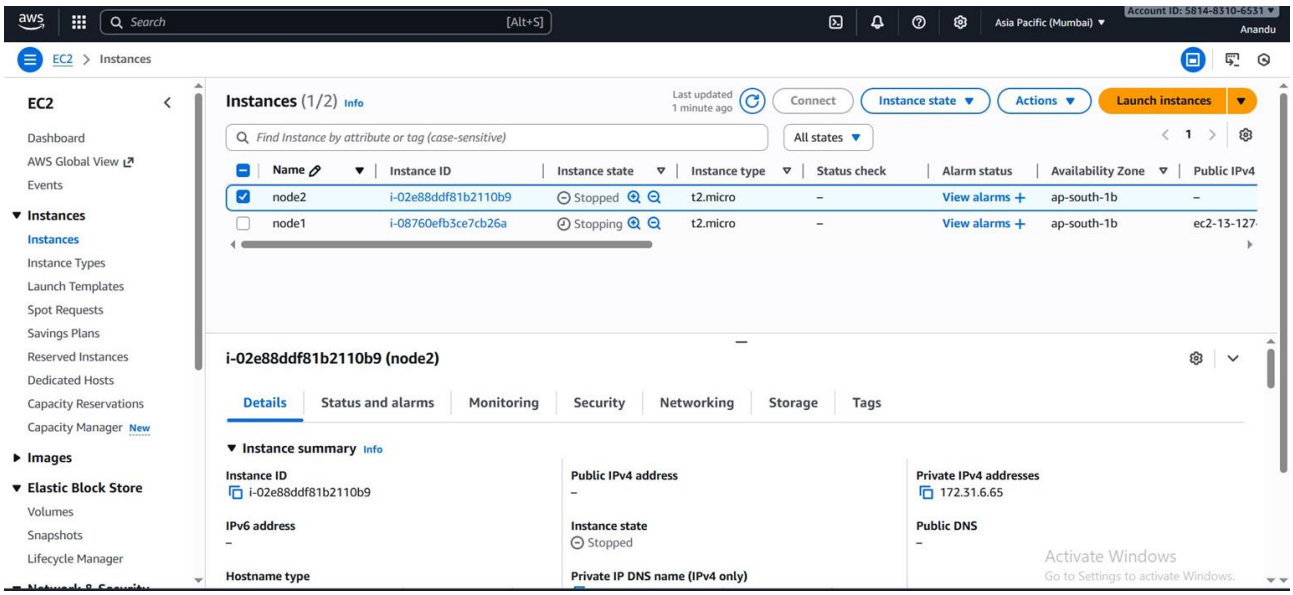

Fig 19: Test result — EC2 instance stopped successfully using stop script.

17. Searching for CloudWatch

This screenshot shows how to open CloudWatch by searching in AWS Console.

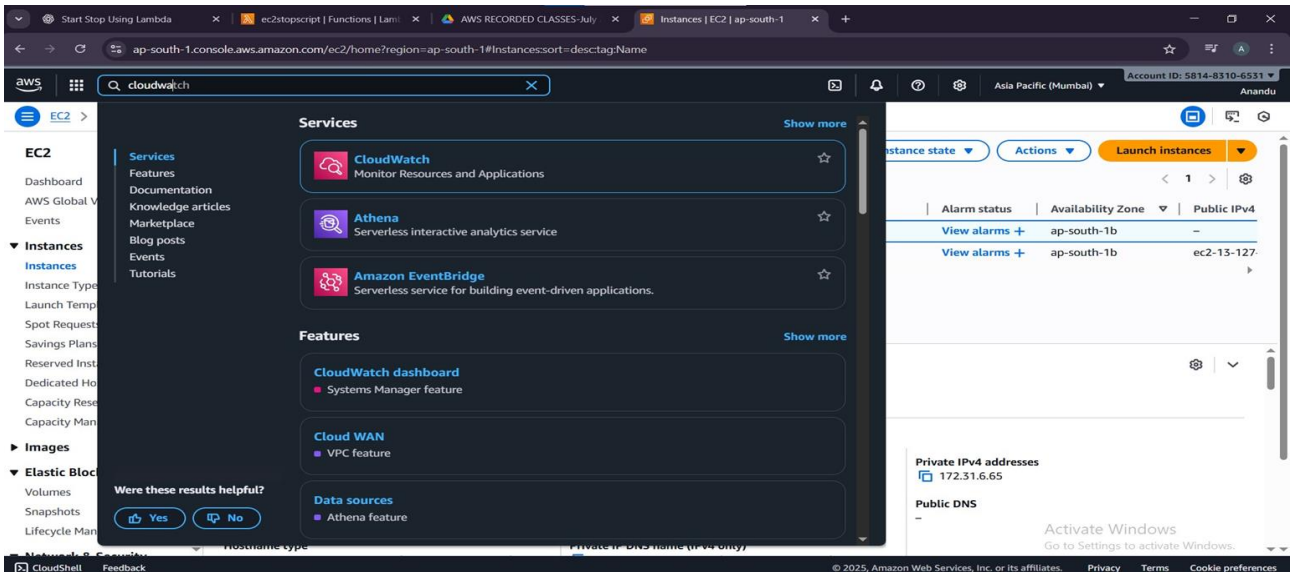

Fig 20: Searching for CloudWatch in the AWS console.

18. Opening Amazon EventBridge (CloudWatch)

This screenshot shows EventBridge, where automation rules will be created.

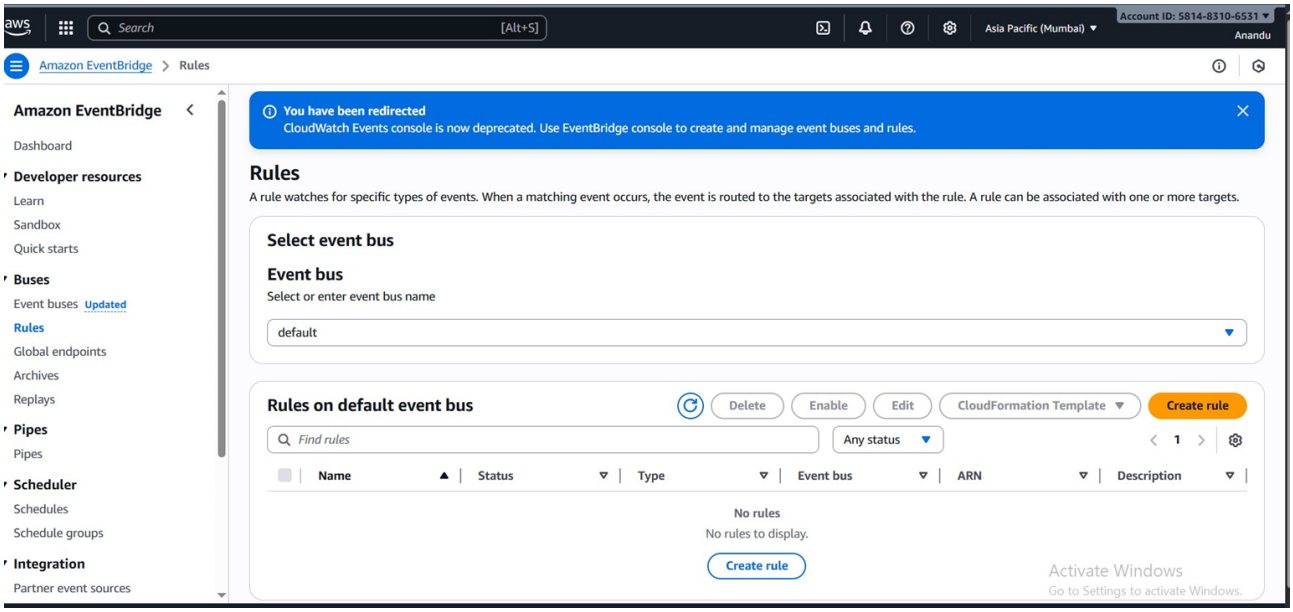

Fig 21: Amazon EventBridge dashboard — Starting the rule creation process.

19. Time Zone (GMT) Reference

A screenshot explains that cron expressions use **GMT**.
Users must adjust schedules accordingly.

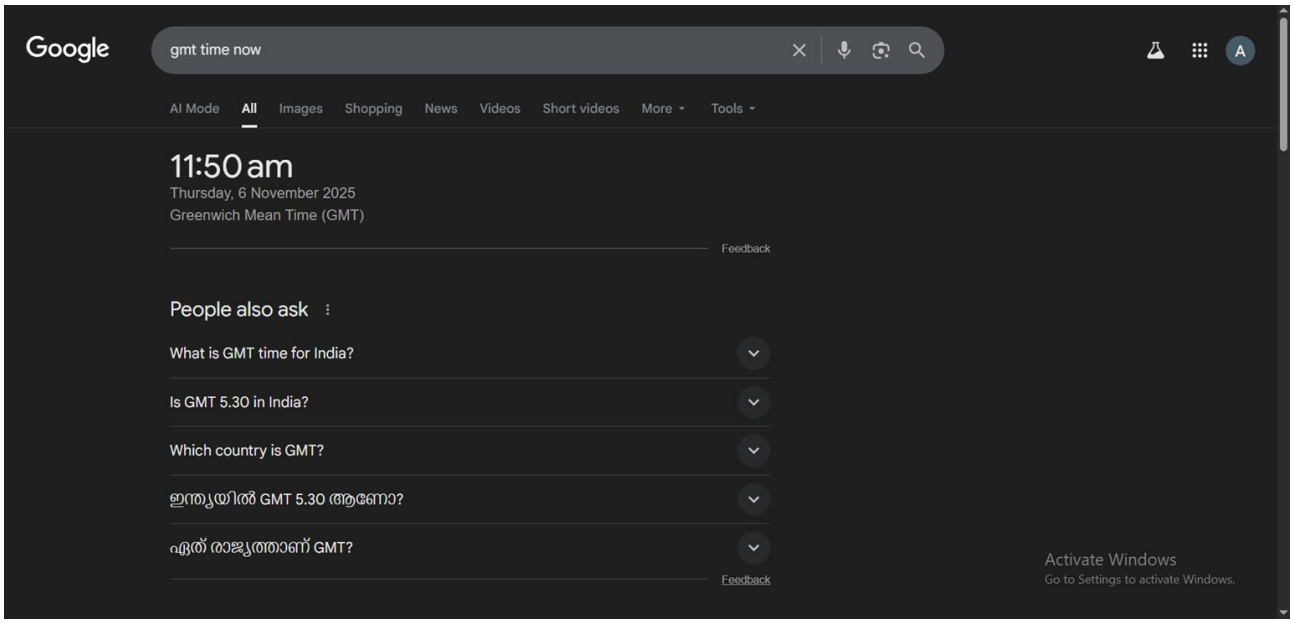

Fig 22: Time zone reference showing cron expressions use GMT/UTC.

20. Creating a Cron Job Rule

This screenshot shows defining a cron expression to automatically trigger the Lambda start function at a scheduled time.

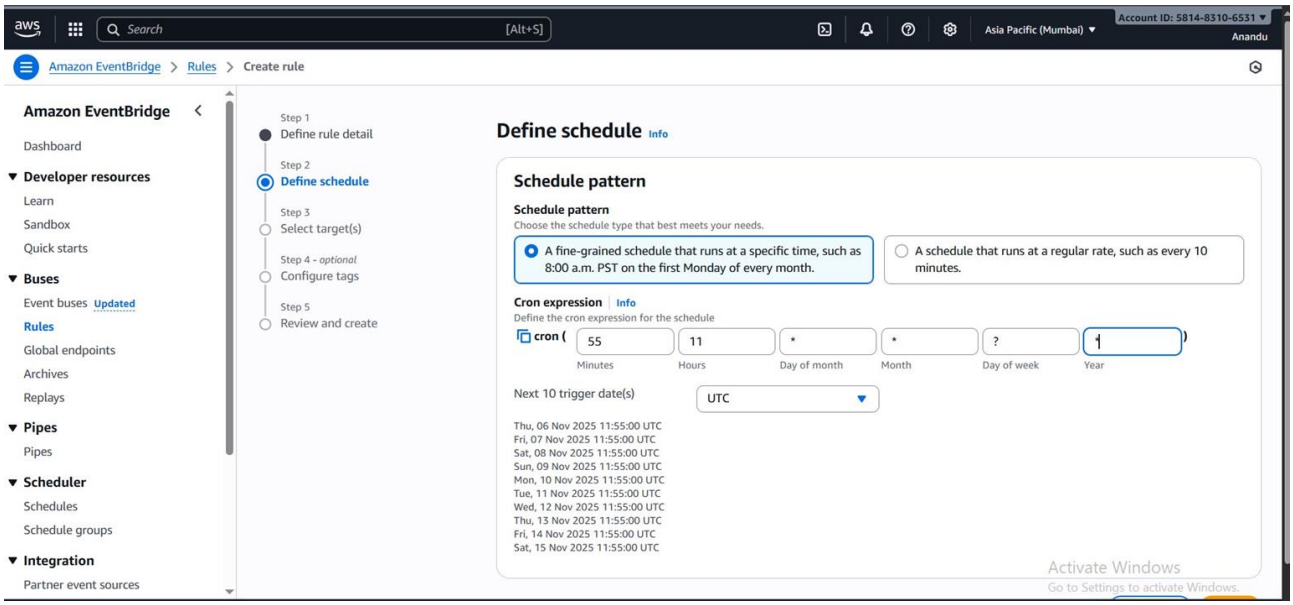

Fig 23: Creating a scheduled rule with a cron expression for EC2 start.

21. Selecting Lambda as the Target

Here you configure:

- **Target type:** AWS Lambda
- **Function name:** lambdafunction

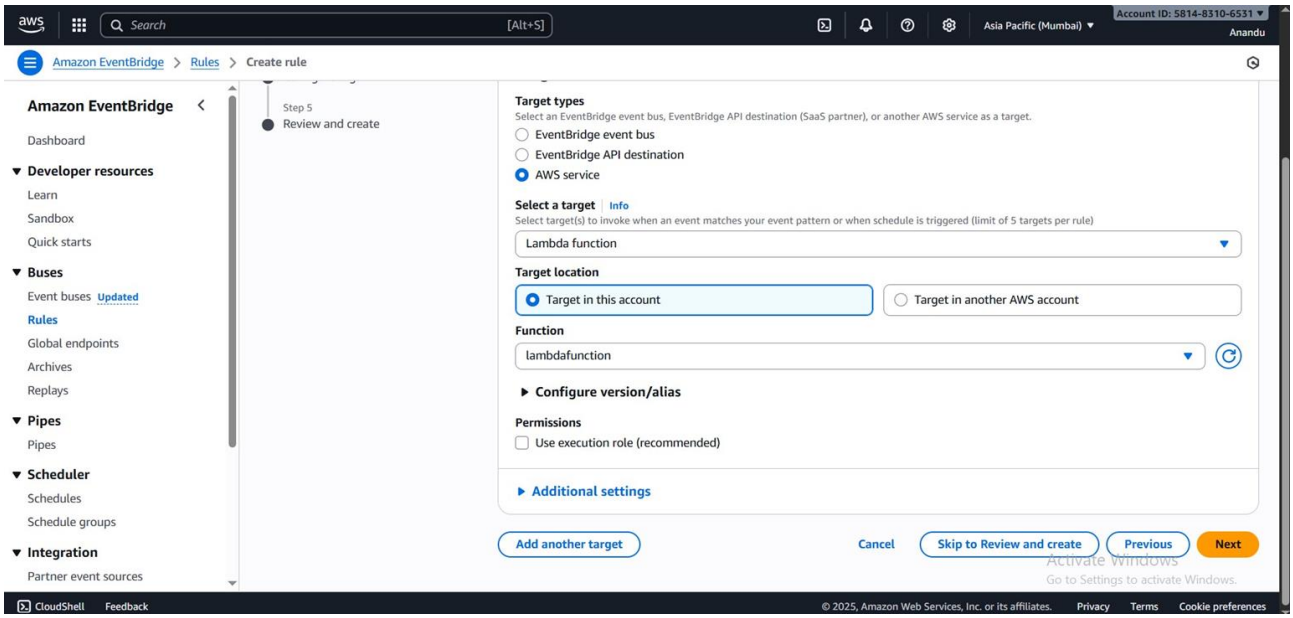

Fig 24: Selecting Lambda function as the target for the EventBridge rule.

- Permissions: Uncheck “Use execution role” (not needed because the role is already attached)

Then click **Next**.

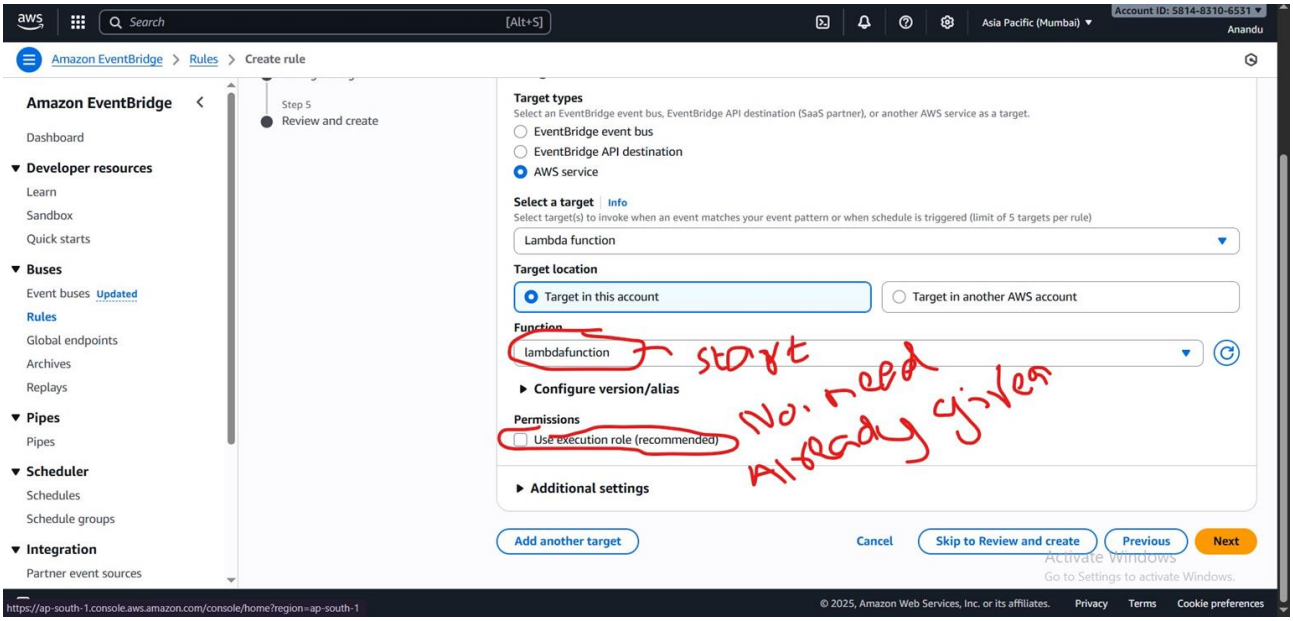

Fig 25: Permissions settings — Leaving execution role option unchecked.

22. Configure Tags (Optional)

This screenshot represents the tagging step. If tags are not required, you proceed without adding any.

Then click **Next**.

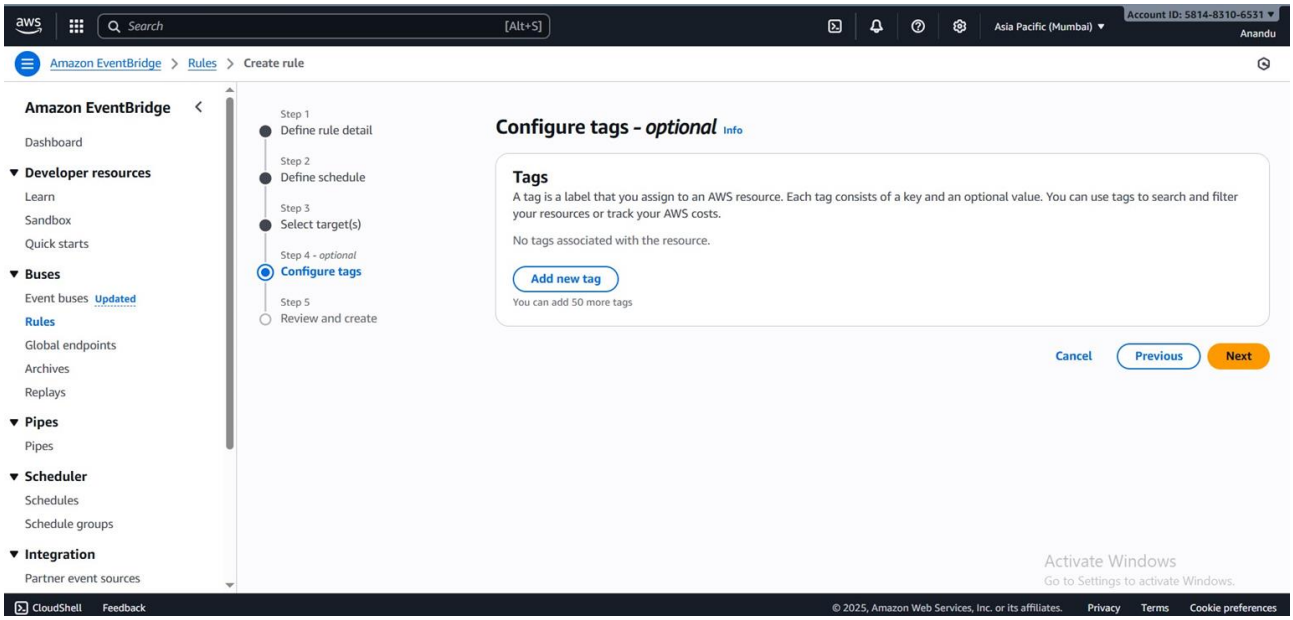

Fig 26: Optional tagging page — Proceeding without tags.

23. Review & Create Rule

This screenshot shows review of:

- Rule name,Target details,Schedule

Click **Create Rule**.

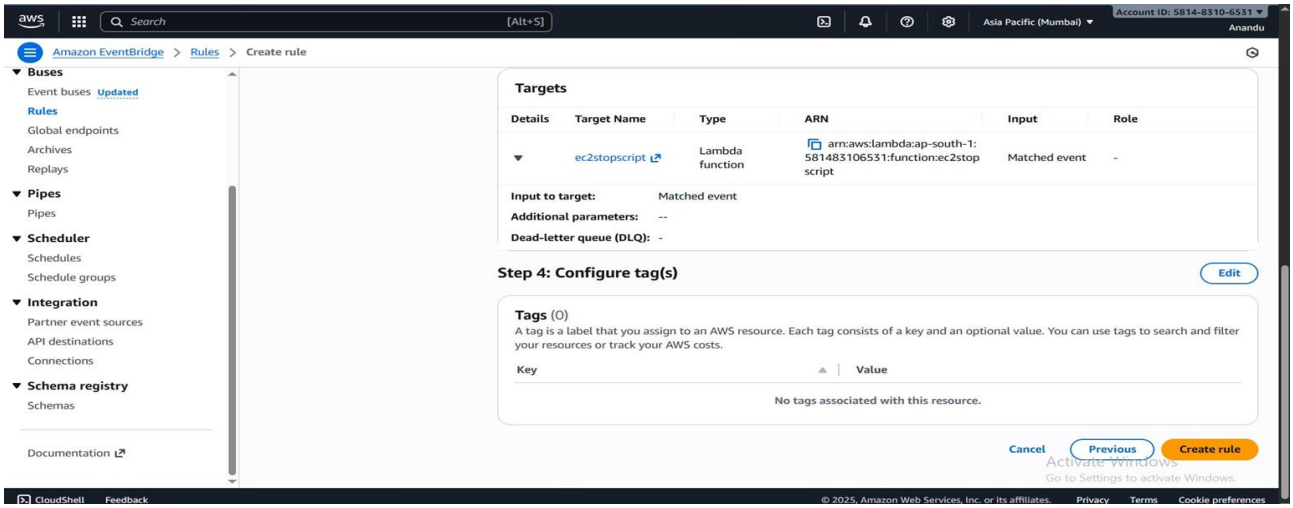

Fig 27: Review page showing rule details before final creation.

24. Success Message – EC2-start Rule Created

This screenshot shows confirmation that the EventBridge rule **Ec2-start** was created successfully.

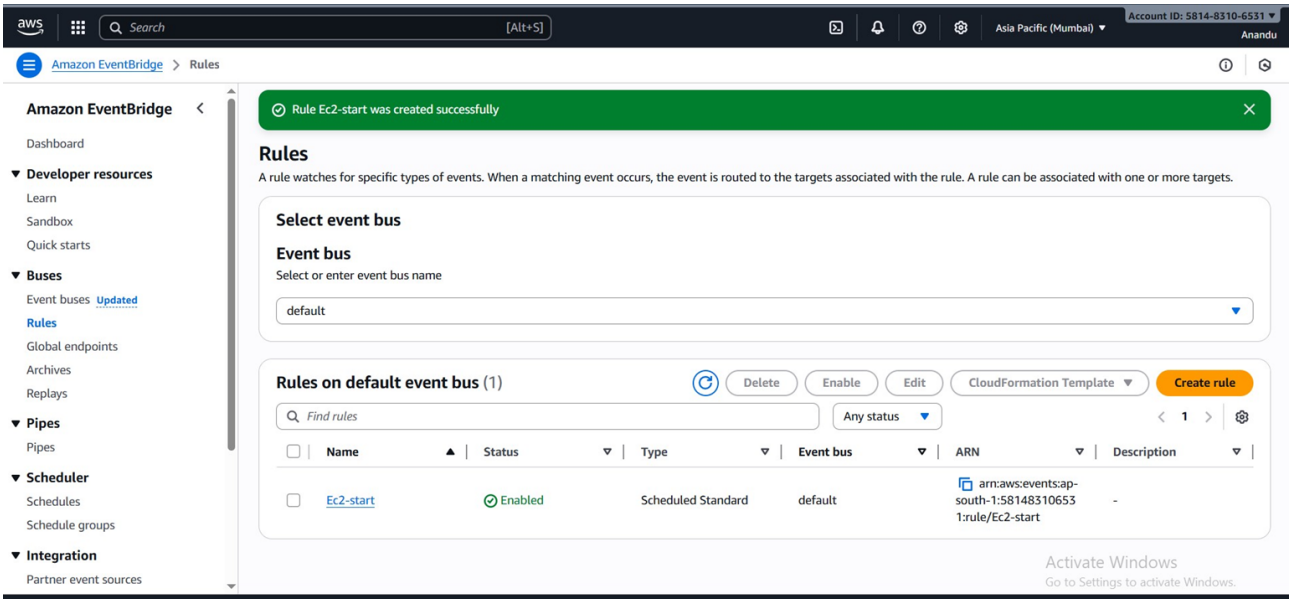

Fig 28: Success message — EC2-start rule created successfully.

25. Instance Automatically Started

This screenshot shows the EC2 instance starting automatically due to the Ec2-start rule.
This confirms the entire start automation pipeline works correctly.

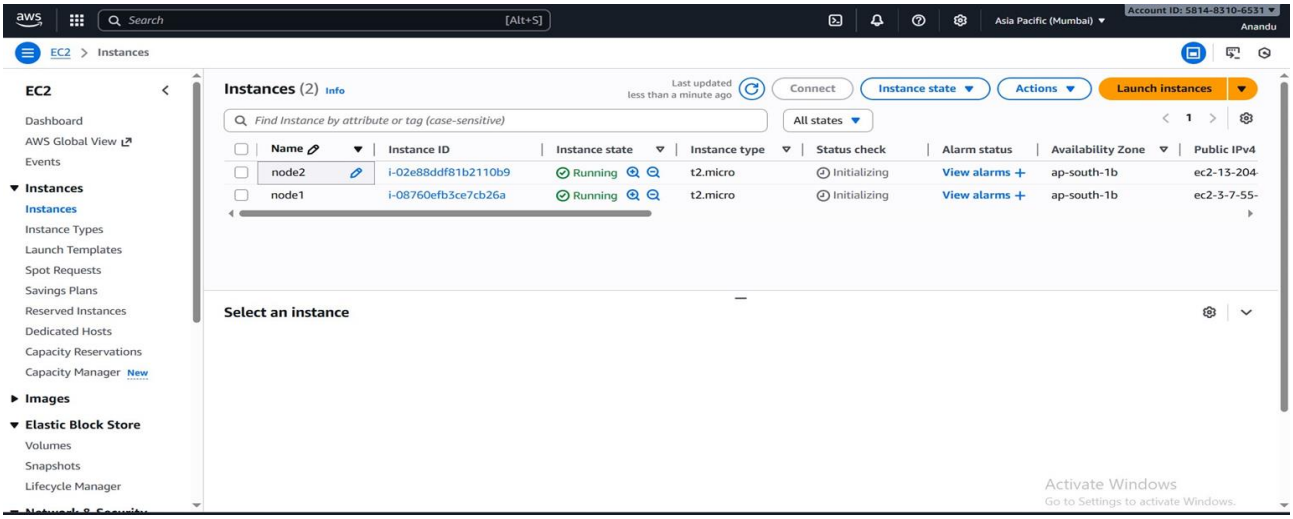

Fig 29: Instance automatically started by the EC2-start rule.

26. Creating the EC2-stop Rule

Similar to the start rule, a screenshot shows creation of an **Ec2-stop** scheduled rule.

Success Message – EC2-stop Rule Created

This screenshot shows confirmation that **Ec2-stop** rule was created successfully.

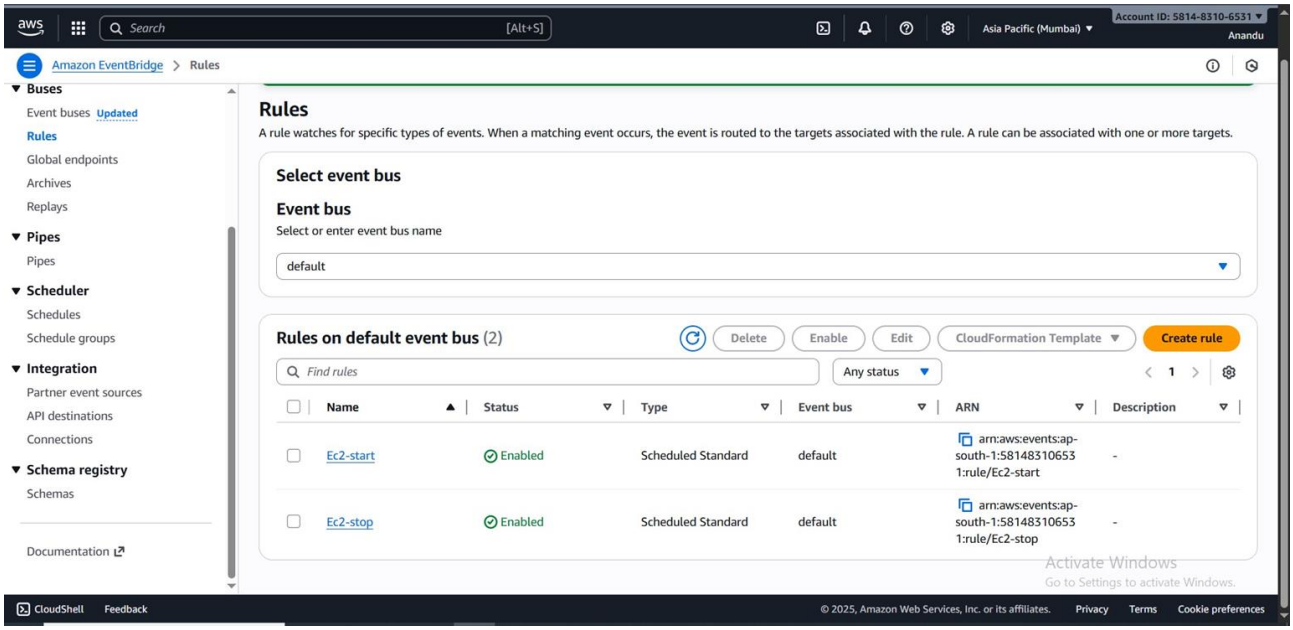

Fig 30: Success message — EC2-stop rule created successfully.

27. Instance Automatically Stopped

The last screenshot shows the EC2 instances transitioning to “stopped” automatically as a result of the Ec2-stop rule.

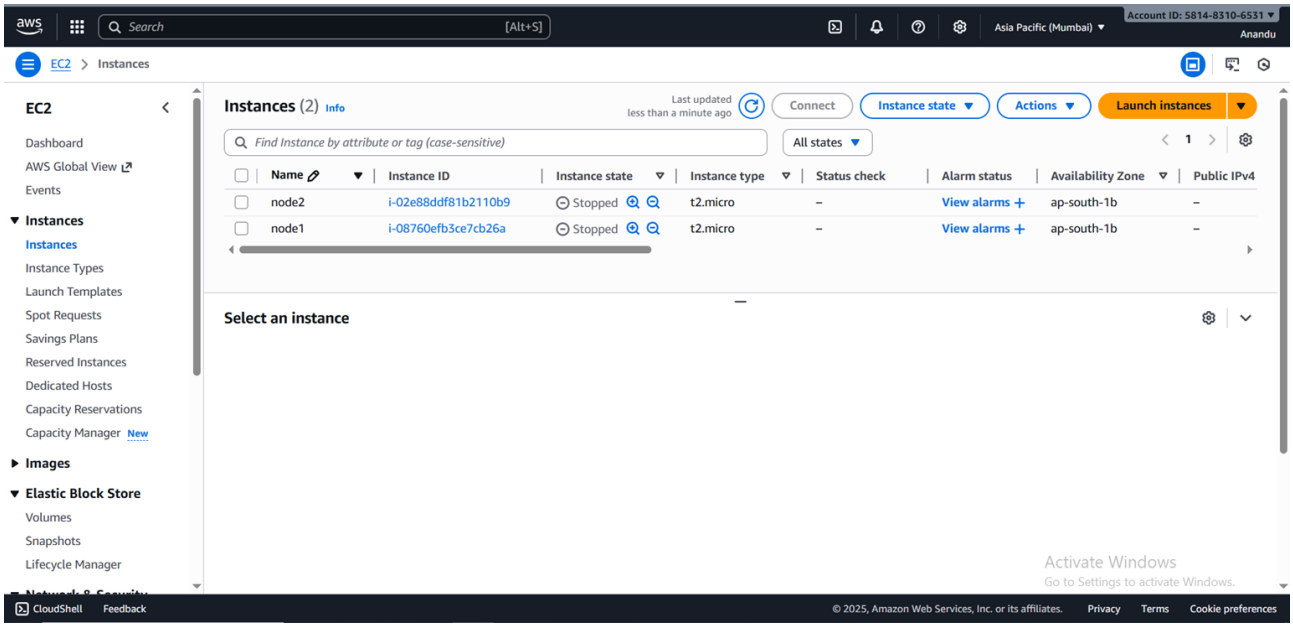

Fig 31: Instance automatically stopped by the EC2-stop rule.

✓CONCLUSION

In this project, AWS Lambda and Amazon EventBridge (CloudWatch) were successfully used to automate the lifecycle of EC2 instances. By creating IAM policies, roles, Lambda functions, and scheduled rules, the EC2 instances now start and stop automatically based on predefined cron expressions.

This automation helps reduce operational overhead and ensures cost efficiency by running compute resources only when required.

The workflow demonstrates real-world DevOps practices involving serverless functions, event scheduling, access control, and cloud resource management.

The screenshots and explanations together provide a complete understanding of how EC2 automation is implemented end-to-end in AWS.