Name : **Devansh Wadhwani**

Class : **D15A**

Roll No. : **64**

EXPERIMENT - 11

Aim: To understand AWS Lambda, its workflow, various functions and create your first Lambda functions using Python / Java / Nodejs.

Theory:

AWS Lambda

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). Users of AWS Lambda create functions, self-contained applications written in one of the supported languages and runtimes, and upload them to AWS Lambda, which executes those functions in an efficient and flexible manner.The Lambda functions can perform any kind of computing task, from serving web pages and processing streams of data to calling APIs and integrating with other AWS services.

The concept of "serverless" computing refers to not needing to maintain your own servers to run these functions. AWS Lambda is a fully managed service that takes care of all the infrastructure for you. And so "serverless" doesn't mean that there are no servers involved: it just means that the servers, the operating systems, the network layer and the rest of the infrastructure have already been taken care of so that you can focus on writing application code.

Features of AWS Lambda

●   AWS Lambda easily scales the infrastructure without any additional configuration. It reduces the operational work involved.

●   It offers multiple options like AWS S3, CloudWatch, DynamoDB, API Gateway, Kinesis, CodeCommit, and many more to trigger an event.

●   You don't need to invest upfront. You pay only for the memory used by the lambda function and minimal cost on the number of requests hence cost-efficient.

●   AWS Lambda is secure. It uses AWS IAM to define all the roles and security policies.

Packaging Functions

Lambda functions need to be packaged and sent to AWS. This is usually a process of compressing the function and all its dependencies and uploading it to an S3 bucket. And letting AWS know that you want to use this package when a specific event takes place. To help us with this process we use the Serverless Stack Framework (SST). We'll go over this in detail later on in this guide.

Execution Model

The container (and the resources used by it) that runs our function is managed completely by AWS. It is brought up when an event takes place and is turned off if it is not being used. If additional requests are made while the original event is being served, a new container is brought up to serve a request. This means that if we are undergoing a usage spike, the cloud provider simply creates multiple instances of the container with our function to serve those requests.

This has some interesting implications. Firstly, our functions are effectively stateless. Secondly, each request (or event) is served by a single instance of a Lambda function. This means that you are not going to be handling concurrent requests in your code. AWS brings up a container whenever there is a new request. It does make some optimizations here. It will hang on to the container for a few minutes (5 - 15mins depending on the load) so it can respond to subsequent requests without a cold start.

Stateless Functions

The above execution model makes Lambda functions effectively stateless. This means that every time your Lambda function is triggered by an event it is invoked in a completely new environment. You don't have access to the execution context of the previous event.

However, due to the optimization noted above, the actual Lambda function is invoked only once per container instantiation. Recall that our functions are run inside containers. So when a function is first invoked, all the code in our handler function gets executed and the handler function gets invoked. If the container is still available for subsequent requests, your function will get invoked and not the code around it.

For example, the `createNewDbConnection` method below is called once per container instantiation and not every time the Lambda function is invoked. The `myHandler` function on the other hand is called on every invocation.

Common Use Cases for Lambda

Due to Lambda's architecture, it can deliver great benefits over traditional cloud computing setups for applications where:

1. Individual tasks run for a short time;
2. Each task is generally self-contained;
3. There is a large difference between the lowest and highest levels in the workload of the application.

Some of the most common use cases for AWS Lambda that fit these criteria are: Scalable APIs. When building APIs using AWS Lambda, one execution of a Lambda function can serve a single HTTP request. Different parts of the API can be routed to different Lambda functions via Amazon API Gateway. AWS Lambda automatically scales individual functions according to the demand for them, so different parts of your API can scale differently according to current usage levels. This allows for cost-effective and flexible API setups.

Data processing. Lambda functions are optimized for event-based data processing. It is easy to integrate AWS Lambda with data sources like Amazon DynamoDB and trigger a Lambda function for specific kinds of data events. For example, you could employ Lambda to do some work every time an item in DynamoDB is created or updated, thus making it a good fit for things like notifications, counters and analytics.
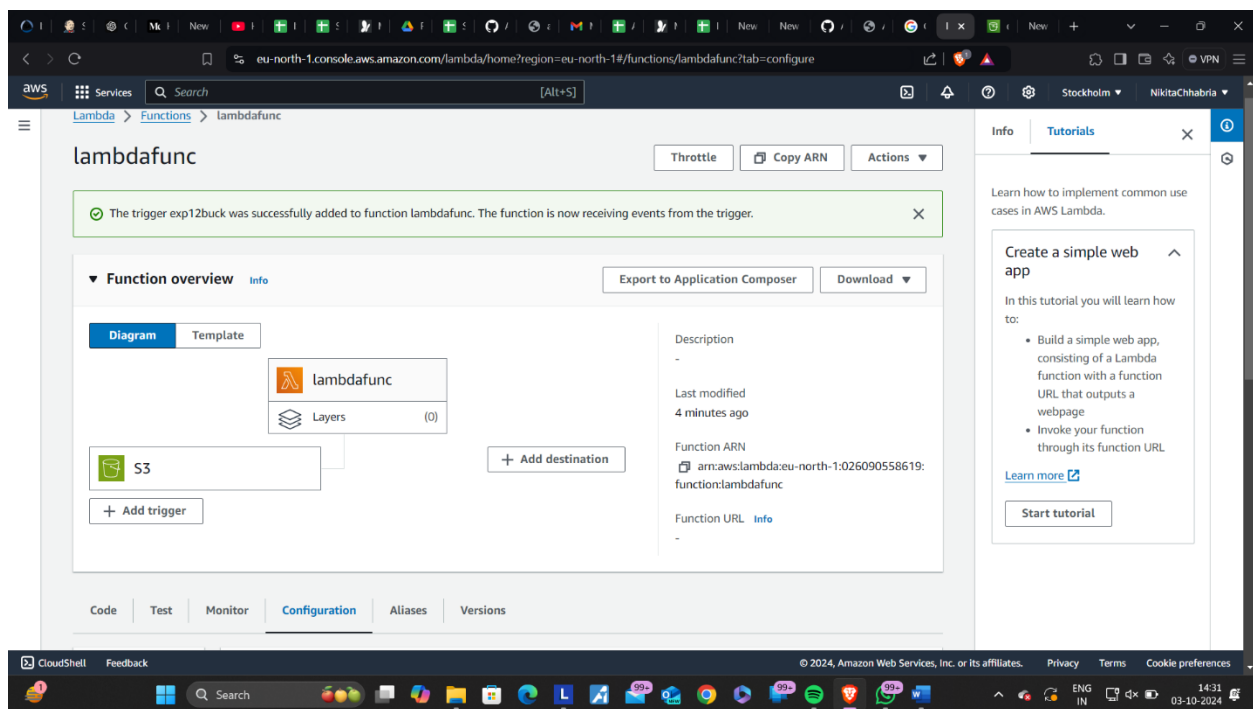
Configure test event   ✕

A test event is a JSON object that mocks the structure of requests emitted by AWS services to invoke a Lambda function. Use it to see the function's invocation result.

To invoke your function without saving an event, configure the JSON event, then choose Test.

Test event action

⦿ Create new event   ◯ Edit saved event

Event name

devanshbuck

Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

Event sharing settings

⦿ Private
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. Learn more 🔗

◯ Shareable
This event is available to IAM users within the same account who have permissions to access and use shareable events. Learn more 🔗

Template - *optional*

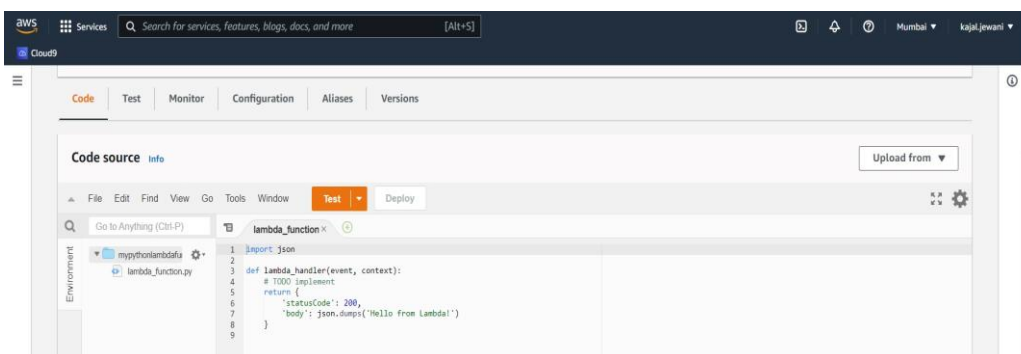hello-world   ▼

Template - *optional*

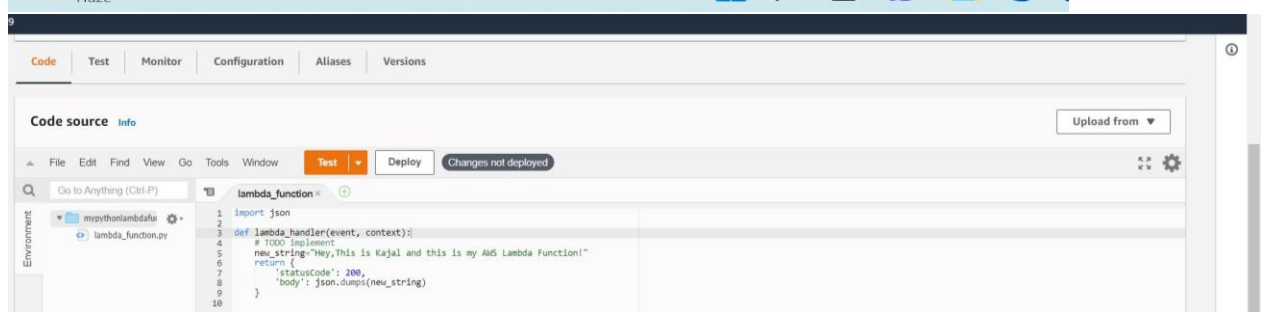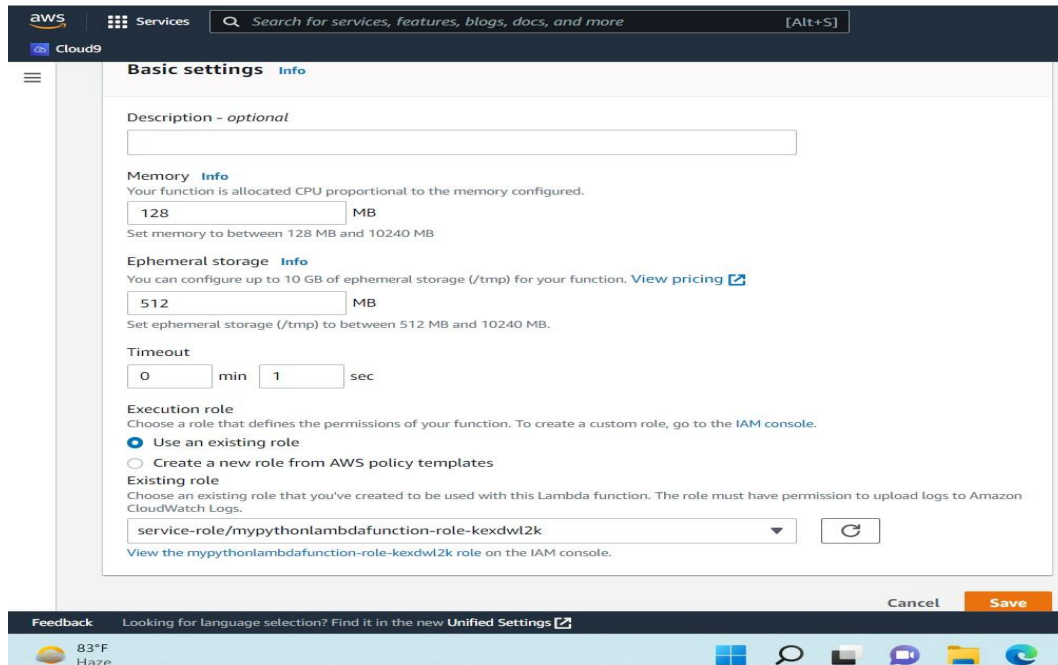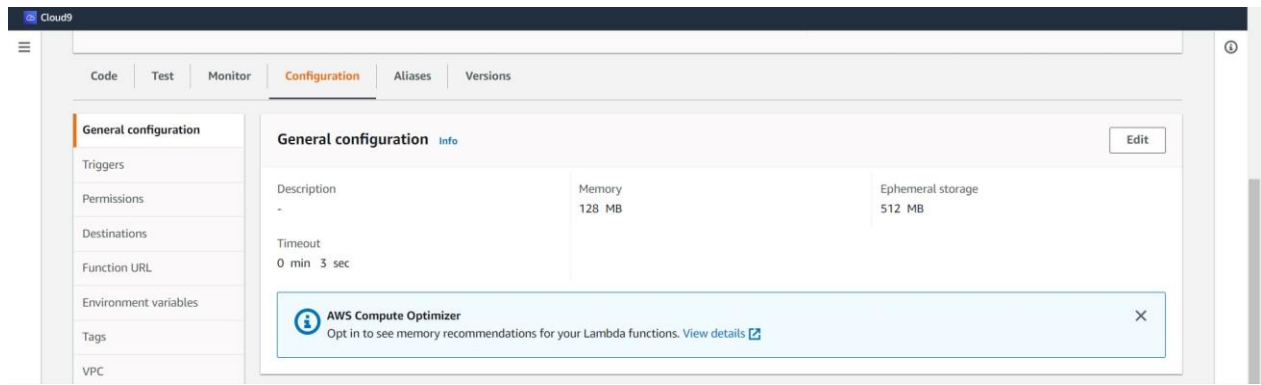hello-world   ▼

Event JSON   Format JSON

```
1 ▾ {
2       "key1": "value1",
3       "key2": "value2",
4       "key3": "value3"
5   }
```

Code | Test | Monitor | Configuration | Aliases | Versions

General configuration
Triggers
Permissions
Destinations
Function URL
Environment variables
Tags
VPC

General configuration Info

Edit

Description
-

Memory
128 MB

Ephemeral storage
512 MB

Timeout
0 min 3 sec

(i) **AWS Compute Optimizer**
Opt in to see memory recommendations for your Lambda functions. View details [↗]   ✕

---

aws | ::: Services | 🔍 Search for services, features, blogs, docs, and more   [Alt+S]

Cloud9

Basic settings Info

Description - *optional*

Memory Info
Your function is allocated CPU proportional to the memory configured.

128   MB

Set memory to between 128 MB and 10240 MB

Ephemeral storage Info
You can configure up to 10 GB of ephemeral storage (/tmp) for your function. View pricing [↗]

512   MB

Set ephemeral storage (/tmp) to between 512 MB and 10240 MB.

Timeout

0   min   1   sec

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.

⦿ Use an existing role
◯ Create a new role from AWS policy templates

Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

service-role/mypythonlambdafunction-role-kexdwl2k   ▼   ⟳

View the mypythonlambdafunction-role-kexdwl2k role on the IAM console.

Cancel   Save

Feedback   Looking for language selection? Find it in the new **Unified Settings** [↗]

83°F
Haze

Code | Test | Monitor | Configuration | Aliases | Versions

Code source Info   Upload from ▼

▲ File Edit Find View Go Tools Window   Test | ▼   Deploy   Changes not deployed   :: ⚙

🔍 Go to Anything (Ctrl-P)   lambda_function × ⊕

▼ mypythonlambdafu ⚙ ▾
  ⊕ lambda_function.py

```
1  import json
2
3  def lambda_handler(event, context):
4      # TODO implement
5      new_string="Hey,This is Kajal and this is my AWS Lambda Function!"
6      return {
7          'statusCode': 200,
8          'body': json.dumps(new_string)
9      }
10
```

A test event is a JSON object that mocks the structure of requests emitted by AWS services to invoke a Lambda function. Use it to see the function's invocation result.

To invoke your function without saving an event, configure the JSON event, then choose Test.

Test event action

◉ Create new event   ○ Edit saved event

Event name

mytestevent

Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

Event sharing settings

◉ Private
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. Learn more ↗

○ Shareable
This event is available to IAM users within the same account who have permissions to access and use shareable events. Learn more ↗

Template - optional

hello-world ▼

Event JSON   Format JSON

```
1 ▾ {
2    "key1": "value1",
3    "key2": "value2",
4    "key3": "value3"
5  }
```

Code source Info   Upload from ▼

File Edit Find View Go Tools Window   Test ▼   Deploy   Changes not deployed

Go to Anything (Ctrl-P)

lambda_function ×   Environment Var ×   Execution result × ⊕

▾ lambdakomal
  lambda_function.py

▾ Execution results   Status: Succeeded | Max memory used: 33 MB | Time: 1.90 ms

Test Event Name
komalevent

Response
```
{
  "statusCode": 200,
  "body": "\"Hello from Lambda!\""
}
```

Function Logs
START RequestId: 3c07521e-c0c3-4431-afca-991cd27e7724 Version: $LATEST
END RequestId: 3c07521e-c0c3-4431-afca-991cd27e7724
REPORT RequestId: 3c07521e-c0c3-4431-afca-991cd27e7724 Duration: 1.90 ms   Billed Duration: 2 ms   Memory Size: 128 MB Max Memory Used: 33 MB

Request ID
3c07521e-c0c3-4431-afca-991cd27e7724

Conclusion:

AWS Lambda is a serverless computing service that allows you to run code without managing servers, making it highly scalable, cost-effective, and easy to use. It automatically manages the compute resources, executes your code in response to specific events such as API calls, file uploads, or database updates, and scales based on the demand.