Author: Madhurima Rawat

Using Cloud Functions for Serverless Computing

This experiment involves the deployment, execution, and testing of an AWS Lambda function using AWS CLI and LocalStack. The Lambda runtime image is pulled from Amazon ECR with Docker for local execution. The process simulates a serverless environment, enabling function invocation and result validation.

What is AWS Lambda and Serverless Computing?

AWS Lambda is a **serverless compute service** provided by Amazon Web Services (AWS) that allows users to run code without provisioning or managing servers. With AWS Lambda, developers can focus solely on writing the code for their applications, while AWS automatically handles the infrastructure, scaling, and maintenance.

What is Serverless Computing?

Serverless computing refers to a cloud computing model where the cloud provider manages the infrastructure and automatically allocates resources as needed. The term "serverless" does not mean that there are no servers involved; rather, it means that the responsibility of managing servers is abstracted away from the developer. The cloud provider handles provisioning, scaling, and managing the servers, allowing the developer to focus on building and deploying the application code.

AWS Lambda: Detailed Explanation

AWS Lambda is one of the most prominent examples of serverless computing. It allows users to execute code in response to events (e.g., HTTP requests, file uploads, database changes) without having to manage servers. Lambda functions can be triggered by various AWS services, such as **API Gateway**, **S3**, **DynamoDB**, and more.

Key Features and Concepts:

1. **Event-Driven**: Lambda functions are triggered by events such as HTTP requests, data uploads, or database changes.
2. **Automatic Scaling**: AWS Lambda automatically scales to accommodate the number of requests. If your application experiences a sudden increase in traffic, Lambda adjusts automatically to handle the load without manual intervention.

3. **Pay-Per-Use**: You only pay for the compute time your Lambda function consumes, rather than for idle server capacity. Charges are based on the number of requests and the duration of the execution.
4. **No Server Management**: Developers don't need to worry about provisioning, managing, or scaling servers. AWS takes care of that, making it ideal for applications with unpredictable traffic.
5. **Multiple Language Support**: Lambda supports a variety of programming languages such as Python, JavaScript (Node.js), Java, C#, Go, and Ruby.
6. **Stateless**: Lambda functions are stateless by nature, meaning that they do not retain any state between invocations. This makes them suitable for distributed architectures.

Use Cases of AWS Lambda:

1. **Data Processing**:
   - Automatically process files uploaded to S3, such as resizing images or transcribing audio files.
2. **Real-Time File Processing**:
   - Trigger Lambda to process data in real time when events occur, such as when a new file is uploaded to a bucket.
3. **Microservices**:
   - Create small, independent functions for individual parts of a service. Lambda makes it easy to break down a large monolithic application into small, reusable components.
4. **API Backends**:
   - Integrate Lambda with API Gateway to build serverless web applications and APIs.
5. **Automation**:
   - Automate administrative tasks like backup management, report generation, and cleanup activities.
6. **Event-Driven Applications**:
   - Build event-driven architectures that respond to changes in real-time, such as sending an email notification when a new user signs up.

Pros and Cons of AWS Lambda:

Pros:

- **Cost-Efficient**: You only pay for the actual compute time your function consumes. There are no charges for idle time.
- **Scalable**: Automatically scales depending on the number of requests or events, making it highly flexible.
- **No Infrastructure Management**: No need to provision or maintain servers.

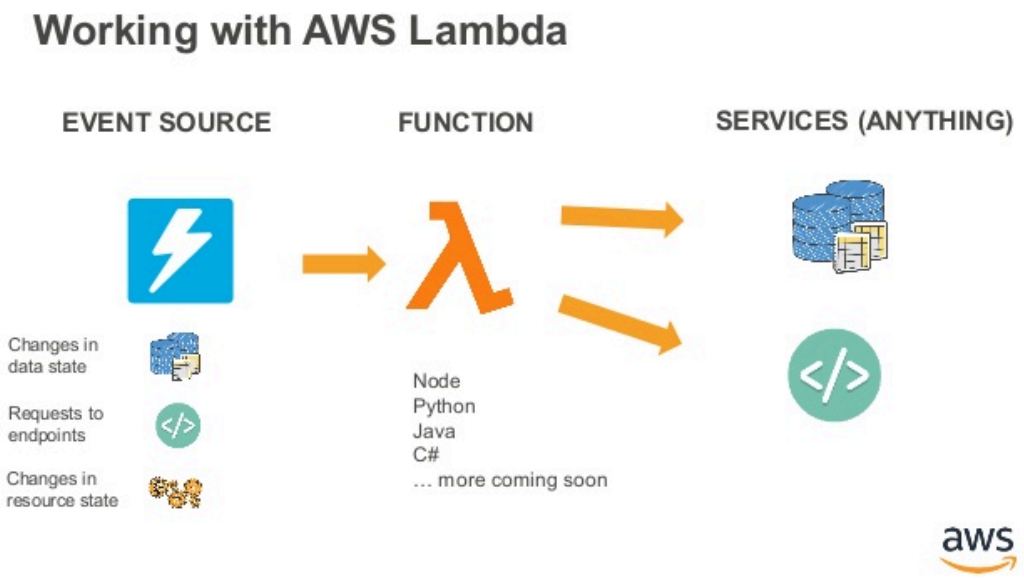
- **Quick Deployment**: With serverless computing, you can deploy applications quickly without worrying about backend management.
- **Built-In Security**: Lambda integrates with AWS IAM (Identity and Access Management), which makes managing permissions and security easier.

Cons:

- **Cold Start Latency**: When a Lambda function is called for the first time after being idle for a while, there may be a slight delay (cold start) as the environment is initialized.
- **Limited Execution Time**: Each Lambda function can run for a maximum of 15 minutes. Long-running processes need to be split into smaller tasks.
- **Stateless**: Lambda functions do not store state between invocations, so developers must handle the state externally if needed (e.g., using databases).
- **Resource Limits**: Lambda has certain resource limits, such as memory, which could be restrictive for resource-heavy tasks.

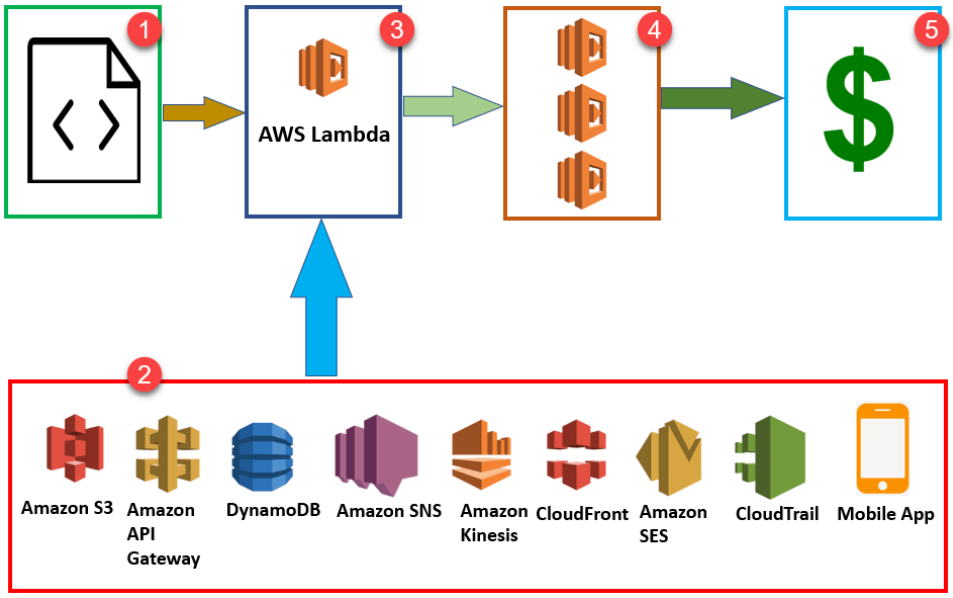
In-Depth Understanding of AWS Lambda and Serverless Computing: Key Concepts, Uses, and Real-World Examples

First Image: AWS Lambda Overview



This image provides a visual overview of **AWS Lambda**, emphasizing that it is a serverless compute service where code is executed in response to events. AWS Lambda handles infrastructure management, such as provisioning, scaling, and monitoring, automatically. This means developers can focus solely on writing the function code, leaving AWS to take care of the rest. Lambda integrates seamlessly with various AWS services, making it a powerful tool for building event-driven applications.

Second Image: Deep Dive into AWS Lambda's Workflow and Integration



The second image takes a **deeper dive** into AWS Lambda, focusing on how it interacts with other AWS services like **API Gateway**, **S3**, and **DynamoDB** to trigger functions. This visual highlights Lambda's flexibility in real-time data processing, automating backend processes, and building scalable microservices. It illustrates how AWS Lambda fits into event-driven architectures, enabling seamless integration and communication between different AWS components.

Practical Real-World Example:

Imagine you're building a **photo-sharing application**. When a user uploads an image to an S3 bucket, an **AWS Lambda function** is automatically triggered to resize the image to multiple resolutions and store them back in the S3 bucket for further processing or display. Here's how it works in practice:

1. **Triggering Event**: A user uploads an image to an S3 bucket.
2. **Lambda Function**: AWS Lambda listens for this event and invokes a function to resize the image.
3. **Real-Time Processing**: Lambda processes the image in real-time and saves the resized versions back to S3.
4. **Automation**: This happens automatically without the need for manual intervention or server management.

This illustrates the power of AWS Lambda in handling real-time file processing and integrating with other AWS services like S3 for seamless operations in a serverless environment.

Summary:

- **AWS Lambda** is a serverless compute service that simplifies running code in response to events, without requiring server management. It enables event-driven architectures, automatic scaling, and pay-per-use pricing, where you only pay for the compute time consumed.

- **Serverless computing** abstracts infrastructure management, allowing applications to scale automatically based on demand. With Lambda, developers can focus on writing functional code while AWS handles the scaling, provisioning, and maintenance of servers.
- **Real-world use cases** for Lambda include real-time file processing, API backends, building microservices, and automating tasks. Lambda integrates effortlessly with other AWS services like **API Gateway**, **S3**, and **DynamoDB**, making it a highly versatile solution for modern cloud applications.
- **Advantages of AWS Lambda** include cost savings, scalability, and ease of deployment without needing to manage infrastructure. However, it has some limitations, such as cold start latency and execution time limits.

By leveraging AWS Lambda, developers can quickly build scalable and efficient applications while focusing on writing business logic and application code, rather than managing infrastructure. AWS handles the complexities of provisioning, scaling, and maintenance, making Lambda a powerful tool for modern cloud application development.

Simple Lambda Function with LocalStack

Step-by-Step Creation of a Lambda Function with LocalStack

1. Creating the Lambda Function:

To create a Lambda function using LocalStack, the first step involves writing the Lambda function code. The function will be triggered by an event and should return a simple response.

- **Example Lambda Function Code** (`lambda_function.py`):

```
def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': 'Hello from LocalStack Lambda!'
    }
```

This function will output a basic response with a `200 OK` status code when invoked.

2. Deploying the Lambda Function:

Before deploying the Lambda function, the Python code needs to be packaged into a ZIP file. This ZIP file will be used to deploy the function to LocalStack.

- **Steps for Packaging the Lambda Function (on Windows)**:
  i. **Navigate to the directory** where your `lambda_function.py` file is located.

ii. **Right-click** the `lambda_function.py` file and select **Send to > Compressed (zipped) folder** from the context menu.

iii. Rename the resulting `.zip` file to `function.zip`.

After the Lambda function is packaged, it is ready for deployment.

3. Checking the Function Creation Status:

After the Lambda function is deployed, its status can be checked to ensure it has been successfully created. Initially, the status might be in a **Pending** state, indicating that LocalStack is still processing the request to create the function.

- **Command to Check the Function's Status**:

```
aws --endpoint-url=http://localhost:4566 lambda get-function
--function-name myLambdaFunction
```

If the function is still in the **Pending** state, wait for a few moments before checking again. Once the function's status is **Active** or **Available**, it can be invoked.

4. Invoking the Lambda Function:

After the function is in the **Active** state, it can be invoked using the AWS CLI. The invocation will trigger the Lambda function and return the response in a file (in this case, `output.txt`).

- **Command to Invoke the Function**:

```
aws --endpoint-url=http://localhost:4566 lambda invoke
--function-name myLambdaFunction output.txt
```

This will execute the Lambda function and store the output in the `output.txt` file.

Here is the error message:

```
-----------------------------------------------------
|                  GetFunction                      |
-----------------------------------------------------

|                     Code                          |
-----------------------------------------------------
| Location        | http://s3.localhost.localstack |
|                 | .cloud:4566/awslambda-us-east-1 |
|                 | -tasks/snapshots/000000000000/ |
|                 | myLambdaFunction-88a260f7-7427 |
|                 | -46dc-8226-b7f079a11527?AWS    |
|                 | AccessKeyId=949334387222&      |
|                 | Signature=DksEUmpzW5tO%2FXj49v |
```

```
|                | KMaIqPNwM%3D&Expires=1739893833|
-------------------------------------------------
| RepositoryType  | S3                           |
-------------------------------------------------

|                 Configuration                 |
-------------------------------------------------
| CodeSha256      | G16xwAmelqgPBfzQVMlj7SR/dZS5l |
|                | TV2WSY8mGqvvRw=                |
-------------------------------------------------
| CodeSize        | 312 KB                       |
-------------------------------------------------
| Description     |                              |
-------------------------------------------------
| FunctionArn     | arn:aws:lambda:us-east-1:0000  |
|                | 00000000:function:myLambdaFunc |
|                | tion                           |
-------------------------------------------------
| FunctionName    | myLambdaFunction             |
-------------------------------------------------
| Handler         | lambda_function.lambda_handle |
|                | r                              |
-------------------------------------------------
| LastModified    | 2025-02-18T13:15:50.795443+00 |
|                | 00                             |
-------------------------------------------------
| LastUpdateStatus | Failed                       |
-------------------------------------------------
| MemorySize      | 128 MB                       |
-------------------------------------------------
| PackageType     | Zip                          |
-------------------------------------------------
| RevisionId      | b27b563d-ede9-4878-bb36-013a3 |
|                | 5678f00                        |
-------------------------------------------------
| Role            | arn:aws:iam::000000000000:rol |
|                | e/execution_role               |
-------------------------------------------------
| Runtime         | python3.8                    |
-------------------------------------------------

|                 State Details                 |
-------------------------------------------------
| State           | Failed                       |
-------------------------------------------------
| StateReason     | Error while creating lambda:  |
|                | public.ecr.aws/lambda/python:3.8 |
-------------------------------------------------
| StateReasonCode | InternalError                |
-------------------------------------------------
```

Troubleshooting Lambda Creation Issues:

It’s common to face issues when setting up Lambda functions with LocalStack. For example, if LocalStack is unable to fetch the required Lambda runtime (in this case, Python 3.8), the creation will fail. In the example provided, the Lambda function was in a **Failed** state with the error message:

- **State:** `Failed`
- **StateReason:** `Error while creating lambda: public.ecr.aws/lambda/python:3.8`
- **StateReasonCode:** `InternalError`

This indicates that LocalStack could not retrieve the Python 3.8 runtime image from the repository (`public.ecr.aws/lambda/python:3.8`). The error is internal, likely caused by an issue with LocalStack’s ability to access or fetch the necessary container image from the registry.

Why Did the Lambda Function Fail?

The Lambda function failed due to an issue where **LocalStack was unable to fetch the Python 3.8 runtime** from the expected image repository (`public.ecr.aws/lambda/python:3.8`). This led to an **InternalError**.

- **State:** `Failed`
- **StateReason:** `Error while creating lambda: public.ecr.aws/lambda/python:3.8`
- **StateReasonCode:** `InternalError`

This error indicates that LocalStack could not access or retrieve the Lambda runtime, which caused the function creation process to fail.

Troubleshooting and Resolving the issue using Docker

How to Resolve This Issue?

1. Ensure the Correct Lambda Runtime is Available

LocalStack uses **Docker-based Lambda runtimes**. To ensure that LocalStack can properly run the Lambda function, it is necessary to manually pull the required Docker image. In this case, LocalStack requires the Python 3.8 Lambda runtime.

To resolve this, follow these steps to pull the necessary Docker image:

```
docker pull public.ecr.aws/lambda/python:3.8
```

Once this command is executed, Docker will download the required Python 3.8 Lambda image. Below is the expected output when the image is successfully pulled:

Expected Output After Pulling the Image:

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud
Computing\Lab\Experiment 5\Codes>docker pull
public.ecr.aws/lambda/python:3.8
3.8: Pulling from lambda/python
bc2b3a540f9b: Pull complete
7e7a8ab075f3: Pull complete
eb63e7acde8e: Pull complete
35e8cd243a50: Pull complete
d4160264f321: Pull complete
32e94a394cab: Pull complete
Digest: sha256:93e78742873d3ad0c28582366b217
ce5169889f4d63d61179598c2a3dc6142ff
Status: Downloaded newer image for public.ecr.aws/lambda/python:3.8
public.ecr.aws/lambda/python:3.8
```

2. Verify the Image Availability Locally

To confirm that the image has been successfully downloaded, use the following command to list all the Docker images present on the system:

```
docker images
```

The output should list the `public.ecr.aws/lambda/python:3.8` image among other available images. Below is an example of the output you should expect:

```
C:\Users\rawat\Documents\8 SEMESTER\Cloud
Computing\Lab\Experiment 5\Codes>docker images
REPOSITORY                      TAG       IMAGE ID
CREATED       SIZE
flask-app                       latest    269bf42596ed
12 days ago   126MB
<none>                          <none>    44b808030263
2 weeks ago   126MB
<none>                          <none>    063caad47b0a
2 weeks ago   126MB
localstack/localstack           latest    b686f3948f42
3 weeks ago   1.18GB
hello-world                     latest    74cc54e27dc4
3 weeks ago   10.1kB
public.ecr.aws/lambda/python    3.8       348b357f1c82
4 weeks ago   575MB
```

This verifies that the image is available locally.

3. Retry the Lambda Function Creation

Once the Python 3.8 Lambda runtime image is available locally, LocalStack should be able to retrieve the runtime successfully. Retry the Lambda function creation process, and it should deploy without issues.

By following these steps, the error related to the missing Lambda runtime will be resolved, ensuring that LocalStack can properly run the Lambda function without any further issues.

Running a Simple Lambda Function with LocalStack After Resolving Issues

Recreating and Invoking the Lambda Function with LocalStack

Step 1: Recreate the Lambda Function

The first step is to create a new Lambda function in LocalStack. To do so, the AWS CLI `create-function` command is used. This command tells LocalStack to create a Lambda function with the specified configurations.

Command to Create the Lambda Function:

```
aws --endpoint-url=http://localhost:4566 lambda create-function \
    --function-name myLambdaFunction \
    --runtime python3.8 \
    --role arn:aws:iam::000000000000:role/execution_role \
    --handler lambda_function.lambda_handler \
    --zip-file fileb://lambda_function.zip
```

Here's a breakdown of what each part of the command does:

1. **`aws --endpoint-url=http://localhost:4566`**:
    - This specifies that the AWS CLI should interact with LocalStack rather than the actual AWS cloud. `localhost:4566` is the default endpoint for LocalStack's services.
2. **`lambda create-function`**:
    - This is the AWS CLI command used to create a new Lambda function.
3. **`--function-name myLambdaFunction`**:
    - This specifies the name of the Lambda function being created. In this case, the function is named `myLambdaFunction`.
4. **`--runtime python3.8`**:

- This sets the runtime for the Lambda function to Python 3.8. LocalStack uses Docker to run Lambda functions with specific runtimes, and Python 3.8 is chosen here.

5. **`--role arn:aws:iam::000000000000:role/execution_role`**:
   - This assigns an IAM role to the Lambda function. The `execution_role` is required for Lambda functions to execute. In LocalStack, this is a placeholder, and a specific role ARN is not needed for local execution.
6. **`--handler lambda_function.lambda_handler`**:
   - This specifies the function within the code to be executed when the Lambda function is invoked. The handler format is `file_name.function_name`, which tells LocalStack to look in the `lambda_function.py` file for the `lambda_handler` function to run.
7. **`--zip-file fileb://lambda_function.zip`**:
   - This points to the zip file containing the Lambda function code. The `fileb://` prefix tells AWS CLI that the file is on the local file system. The `lambda_function.zip` file must contain the Python script (`lambda_function.py`).

Step 2: Invoke the Lambda Function

Once the function is created successfully, it can be invoked using the AWS CLI. This triggers the Lambda function and stores the output in a file.

Command to Invoke the Lambda Function:

```
aws --endpoint-url=http://localhost:4566 lambda invoke
--function-name myLambdaFunction output.txt
```

Here's a breakdown of what each part of the invoke command does:

1. **`aws --endpoint-url=http://localhost:4566`**:
   - Just like before, this specifies that the command should interact with LocalStack on the specified endpoint.
2. **`lambda invoke`**:
   - This AWS CLI command triggers the Lambda function and executes it.
3. **`--function-name myLambdaFunction`**:
   - This specifies the name of the Lambda function that should be invoked. In this case, it's `myLambdaFunction`, which was created in the previous step.

4. **`output.txt`**：
    - This is the file where the output of the Lambda function invocation will be stored. After the function runs, its response will be written to `output.txt`.

Step 3: View the Output

After invoking the Lambda function, the results are stored in the `output.txt` file. The `type` command in Windows can be used to display the contents of this file.

Command to View the Output:

```
type output.txt
```

This command displays the content of the `output.txt` file. The expected output looks like this:

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

Explanation of the Output:

The content of the `output.txt` file is a JSON response returned by the Lambda function:

1. **`statusCode: 200`**：
    - This indicates that the Lambda function executed successfully and returned an HTTP status code of `200`, which signifies a successful response.
2. **`body: "Hello from LocalStack Lambda!"`**：
    - This is the body of the response. It contains the message `"Hello from LocalStack Lambda!"`, which was defined in the Lambda function. This message is returned as part of the response when the function is triggered.

Summary of Commands and Outputs:

1. **Recreate the Lambda Function**: The `create-function` command defines the function, its runtime (Python 3.8), execution role, and handler function. It also links the Lambda function to the `lambda_function.zip` file containing the code.
2. **Invoke the Lambda Function**: The `invoke` command runs the Lambda function and stores the response in `output.txt`.

3. **View the Output**: The `type` command shows the contents of `output.txt`, which contains a JSON response indicating the Lambda function executed successfully.

By following these steps, the Lambda function is successfully created, invoked, and the output is captured and displayed.

Complex Lambda Function with Payload with LocalStack

Detailed Commands, Redeployment, and Output for Lambda Function with Payload Processing

Updated Lambda Function Code:

Here is the updated Lambda function code that processes a payload with an input event and performs some computations:

```
import json
import logging

# Set up logging for debugging and monitoring
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger()

# Function versioning for tracking updates
VERSION = "1.0.0"


def lambda_handler(event, context):
    """
    Handles AWS Lambda execution by processing event data
    and performing basic computations.

    **Parameters:**
    - event (dict): The input event data, expected to contain:
        - `"name"` (str, optional): The name of the user.
          Defaults to `"Guest"` if not provided.
        - `"number"` (int, optional): A number to be processed.
          Defaults to `0` if not provided.
    - context (object): AWS Lambda context object
      (not used in this function).

    **Returns:**
    - dict: JSON-formatted response with:
        - `"message"` (str): Greeting message.
        - `"processedNumber"` (int): The input number multiplied by 2.
        - `"version"` (str): Function version identifier.
    """
```

```
    # Log event details for debugging
    logger.info(f"Function invoked with event: {json.dumps(event)}")
    logger.info(f"Lambda function version: {VERSION}")

    # Extract 'name' from the event, defaulting to "Guest"
    name = event.get("name", "Guest")
    greeting = f"Hello, {name}!"

    # Extract 'number' from the event and process it
    number = event.get("number", 0)  # Default to 0 if not provided
    result = number * 2  # Double the input number

    # Construct the response payload
    response = {
        "statusCode": 200,  # HTTP status code indicating success
        "body": json.dumps(
            {
                "message": greeting,  # Greeting message
                "processedNumber": result,  # Computed number result
                "version": VERSION,  # Function version for tracking
            }
        ),
    }

    # Log response details before returning
    logger.info(f"Returning response: {json.dumps(response)}")

    return response
```

Explanation of the Code:

1. **Logging**:
    - The `logging` module is used to track and debug the Lambda function. It logs the event data received, the function version, and the response returned. The `INFO` level ensures that relevant details are logged for debugging.
    - The `logger.info` statements are used to print the function invocation details and the response.
2. **Complex Logic**:
    - The function checks if the `name` parameter exists in the event data. If not, it defaults to `"Guest"`.
    - It retrieves the `number` parameter and doubles the number as an example of computation (i.e., `number * 2`).
3. **Versioning**:

- The `VERSION` variable tracks the version of the Lambda function (e.g., `"1.0.0"`). This can be updated whenever changes are made to the code.

Steps to Fix and Redeploy the Lambda Function:

1. Delete the Existing Lambda Function:

Before updating the function with the new code, it's essential to delete the old version of the function in LocalStack to avoid conflicts.

Command to Delete the Old Lambda Function:

```
aws --endpoint-url=http://localhost:4566 lambda delete-function
--function-name myLambdaFunction
```

This command deletes the function named `myLambdaFunction` from LocalStack, preparing for the redeployment with updated code.

2. Recreate the Lambda Function with Updated Code:

Once the old function is deleted, zip the updated code and recreate the function. Make sure the updated `lambda_function.py` is in your current directory.

Step 1: Zip the Updated Code:

```
zip lambda_function.zip lambda_function.py
```

This command creates a `.zip` file containing the updated `lambda_function.py` file. This zip file will be used to deploy the new version of the Lambda function.

Step 2: Recreate the Lambda Function:

```
aws --endpoint-url=http://localhost:4566 lambda create-function \
    --function-name myLambdaFunction \
    --runtime python3.8 \
    --role arn:aws:iam::000000000000:role/execution_role \
    --handler lambda_function.lambda_handler \
    --zip-file fileb://lambda_function.zip
```

This command creates a new Lambda function in LocalStack with the updated code:

- **`--function-name myLambdaFunction`** : Specifies the function name (`myLambdaFunction`).
- **`--runtime python3.8`** : Sets the runtime to Python 3.8.

- **`--role arn:aws:iam::000000000000:role/execution_role`** : Assigns the execution role (this role can be any placeholder role in LocalStack).
- **`--handler lambda_function.lambda_handler`** : Defines the entry point for the Lambda function (`lambda_handler` in the `lambda_function.py` file).
- **`--zip-file fileb://lambda_function.zip`** : Points to the `.zip` file containing the updated code.

3. Invoke the Lambda Function with Payload:

Now that the function is updated, you can invoke it by passing a JSON payload. The payload should be contained in a file (e.g., `event.json`), which includes input data like `"name"` and `"number"` .

Example Payload (`event.json`):

```
{
  "name": "Madhurima",
  "number": 5
}
```

Command to Invoke the Lambda Function:

```
aws --endpoint-url=http://localhost:4566 lambda invoke
--function-name myLambdaFunction --payload file://event.json output.txt
```

This command invokes the Lambda function with the provided payload and stores the result in the `output.txt` file.

- **`--function-name myLambdaFunction`** : Specifies the function to invoke.
- **`--payload file://event.json`** : Points to the input data (the `event.json` file) that contains the payload for the function.
- **`output.txt`** : This file will store the output from the Lambda function.

4. Check the Output:

Finally, to view the result of the Lambda invocation, use the `type` command to display the content of `output.txt` .

Command to View Output:

```
type output.txt
```

The output will look like this:

```
{
  "statusCode": 200,
  "body": "{\"message\": \"Hello, Madhurima!\",
  \"processedNumber\": 10, \"version\": \"1.0.0\"}"
}
```

Explanation of the Output:

- **`statusCode: 200`**:
  - The Lambda function executed successfully and returned an HTTP status code of `200`, which signifies success.
- **`body`**:
  - This contains the actual response from the Lambda function in JSON format. The `body` field includes:
    - **`"message"`**: A greeting message, `"Hello, Madhurima!"`, generated by the function based on the `name` input from the payload.
    - **`"processedNumber"`**: The result of the computation (doubling the `number` value from the payload). For the input `5`, the result is `10`.
    - **`"version"`**: The version of the Lambda function, which is `"1.0.0"` in this case.

Summary:

1. **Delete Old Function**: Use the `delete-function` command to remove the existing Lambda function.
2. **Recreate Lambda Function**: Package the updated code into a zip file and use the `create-function` command to deploy it.
3. **Invoke the Function**: Use the `invoke` command with a payload to test the updated Lambda function.
4. **Check the Output**: View the output in `output.txt` to ensure the Lambda function processed the payload correctly and returned the expected response.

By following these steps, the updated Lambda function processes the payload, performs the computation, and returns the desired results.

Useful Resources for Learning about AWS Lambda

1. **VTI Cloud - AWS Lambda Introduction and Configuration for Beginners**
   This article offers a beginner-friendly introduction to AWS Lambda, explaining the service and its

configuration for those just starting with serverless computing.

2. **AWS Lambda Documentation - Welcome to AWS Lambda**
   The official AWS documentation provides a comprehensive overview of AWS Lambda, including how to get started, manage functions, and integrate with other AWS services.

3. **Dev.to - A Beginner's Guide to AWS Lambda**
   This guide provides an easy-to-follow introduction to AWS Lambda, ideal for newcomers who want to understand the basics of Lambda and its use cases.

4. **Medium - The Complete Beginner's Guide to Creating an AWS Lambda Function from Scratch**
   This comprehensive guide takes readers through the process of creating AWS Lambda functions, starting from scratch and exploring best practices.

5. **Medium - AWS Lambda Functions: A Comprehensive Guide**
   A detailed guide that explains AWS Lambda functions, providing insights into their structure, functionality, and use cases in serverless computing.

6. **Medium - Learn AWS Lambda in 10 Minutes**
   A quick tutorial for learning AWS Lambda in 10 minutes. This resource is perfect for those who want to quickly understand the basics of Lambda and its use in cloud applications.