What is AWS Lambda?

You can use AWS Lambda to run code without provisioning or managing servers.

Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, and logging. With Lambda, all you need to do is supply your code in one of the language runtimes that Lambda supports.

You organize your code into Lambda functions. The Lambda service runs your function only when needed and scales automatically. You only pay for the compute time that you consume—there is no charge when your code is not running. For more information, see AWS Lambda Pricing.

Tip

To learn how to build **serverless solutions**, check out the Serverless Developer Guide.

When to use Lambda

Lambda is an ideal compute service for application scenarios that need to scale up rapidly, and scale down to zero when not in demand. For example, you can use Lambda for:

- **File processing:** Use Amazon Simple Storage Service (Amazon S3) to trigger Lambda data processing in real time after an upload.
- **Stream processing:** Use Lambda and Amazon Kinesis to process real-time streaming data for application activity tracking, transaction order processing, clickstream analysis, data cleansing, log filtering, indexing, social media analysis, Internet of Things (IoT) device data telemetry, and metering.
- **Web applications:** Combine Lambda with other AWS services to build powerful web applications that automatically scale up and down and run in a highly available configuration across multiple data centers.
- **IoT backends:** Build serverless backends using Lambda to handle web, mobile, IoT, and third-party API requests.
- **Mobile backends:** Build backends using Lambda and Amazon API Gateway to authenticate and process API requests. Use AWS Amplify to easily integrate with your iOS, Android, Web, and React Native frontends.

When using Lambda, you are responsible only for your code. Lambda manages the compute fleet that offers a balance of memory, CPU, network, and other resources to run your code. Because Lambda manages these resources, you cannot log in to compute instances or customize the operating system on provided runtimes. Lambda performs operational and administrative activities on your behalf, including managing capacity, monitoring, and logging your Lambda functions.

Key features

The following key features help you develop Lambda applications that are scalable, secure, and easily extensible:

Environment variables

Use environment variables to adjust your function's behavior without updating code.

Versions

Manage the deployment of your functions with versions, so that, for example, a new function can be used for beta testing without affecting users of the stable production version.

Container images

Create a container image for a Lambda function by using an AWS provided base image or an alternative base image so that you can reuse your existing container tooling or deploy larger workloads that rely on sizable dependencies, such as machine learning.

Lambda layers

Package libraries and other dependencies to reduce the size of deployment archives and makes it faster to deploy your code.

Lambda extensions

Augment your Lambda functions with tools for monitoring, observability, security, and governance.

Function URLs

Add a dedicated HTTP(S) endpoint to your Lambda function.

Response streaming

Configure your Lambda function URLs to stream response payloads back to clients from Node.js functions, to improve time to first byte (TTFB) performance or to return larger payloads.

Concurrency and scaling controls

Apply fine-grained control over the scaling and responsiveness of your production applications.

Code signing

Verify that only approved developers publish unaltered, trusted code in your Lambda functions

Private networking

Create a private network for resources such as databases, cache instances, or internal services.

File system

Configure a function to mount an Amazon Elastic File System (Amazon EFS) to a local directory, so that your function code can access and modify shared resources safely and at high concurrency.

Lambda SnapStart

Lambda SnapStart can provide as low as sub-second startup performance, typically with no changes to your function code.

Create your first Lambda function

To get started with Lambda, use the Lambda console to create a function. In a few minutes, you can create and deploy a function and test it in the console.

As you carry out the tutorial, you'll learn some fundamental Lambda concepts, like how to pass arguments to your function using the Lambda *event object*. You'll also learn how to return log outputs from your function, and how to view your function's invocation logs in Amazon CloudWatch Logs.

To keep things simple, you create your function using either the Python or Node.js runtime. With these interpreted languages, you can edit function code directly in the console's built-in code editor. With compiled languages like Java and C#, you must create a deployment package on your local build machine and upload it to Lambda. To learn about deploying functions to Lambda using other runtimes, see the links in the the section called “Next steps” section.

Tip

To learn how to build **serverless solutions**, check out the Serverless Developer Guide.

Prerequisites

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

   For instructions, see Create a permission set in the *AWS IAM Identity Center User Guide*.
2. Assign users to a group, and then assign single sign-on access to the group.

   For instructions, see Add groups in the *AWS IAM Identity Center User Guide*.

Create a Lambda function with the console

In this example, your function takes a JSON object containing two integer values labeled `"length"` and `"width"`. The function multiplies these values to calculate an area and returns this as a JSON string.

Your function also prints the calculated area, along with the name of its CloudWatch log group. Later in the tutorial, you'll learn to use CloudWatch Logs to view records of your functions' invocation.

To create a Hello world Lambda function with the console

1. Open the Functions page of the Lambda console.
2. Choose **Create function**.
3. Select **Author from scratch**.
4. In the **Basic information** pane, for **Function name**, enter **`myLambdaFunction`**.
5. For **Runtime**, choose either **Node.js 22.x** or **Python 3.13**.
6. Leave **architecture** set to **x86_64**, and then choose **Create function**.

In addition to a simple function that returns the message `Hello from Lambda!`, Lambda also creates an execution role for your function. An execution role is an AWS Identity and Access Management (IAM) role that grants a Lambda function permission to access AWS services and

resources. For your function, the role that Lambda creates grants basic permissions to write to CloudWatch Logs.

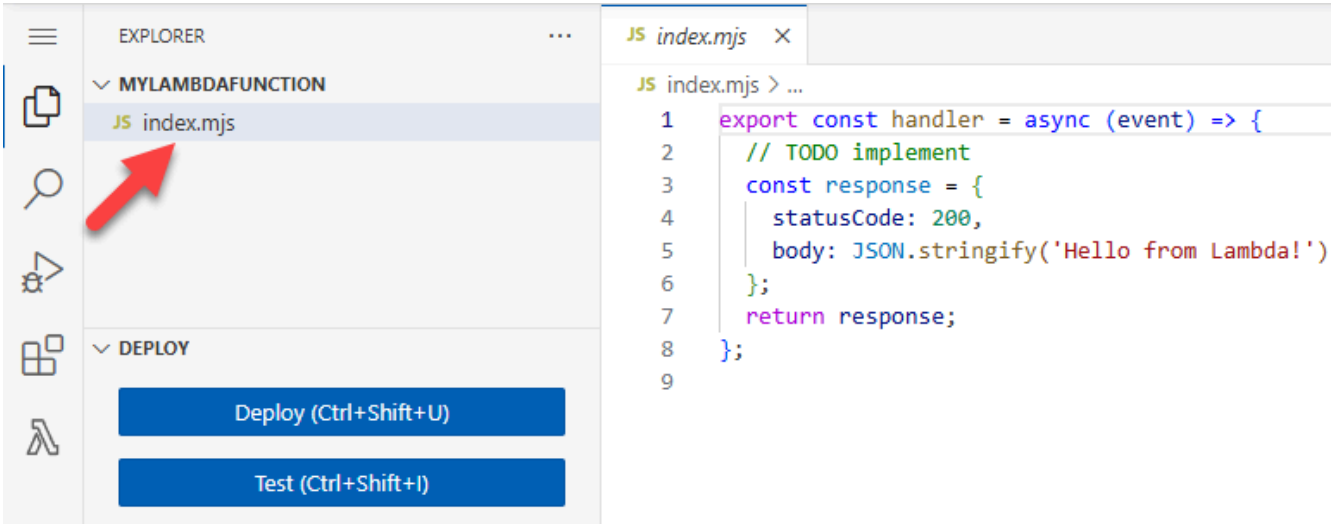
Use the console's built-in code editor to replace the Hello world code that Lambda created with your own function code.

Node.js

To modify the code in the console

1. Choose the **Code** tab.

   In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the **index.mjs** tab in the code editor, select **index.mjs** in the file explorer as shown on the following diagram.

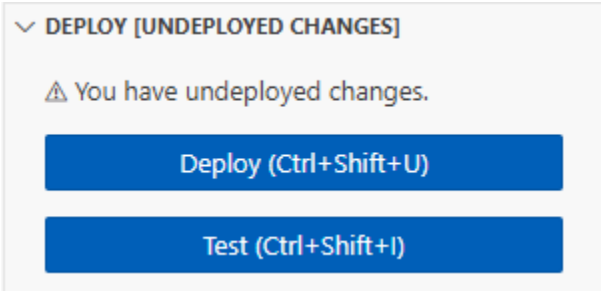


2. Paste the following code into the **index.mjs** tab, replacing the code that Lambda created.

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
```

```
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Understanding your function code

Before you move to the next step, let's take a moment to look at the function code and understand some key Lambda concepts.

- The Lambda handler:

  Your Lambda function contains a Node.js function named `handler`. A Lambda function in Node.js can contain more than one Node.js function, but the *handler* function is always the entry point to your code. When your function is invoked, Lambda runs this method.

  When you created your Hello world function using the console, Lambda automatically set the name of the handler method for your function to `handler`. Be sure not to edit the name of this Node.js function. If you do, Lambda won't be able to run your code when you invoke your function.

  To learn more about the Lambda handler in Node.js, see the section called "Handler".

- The Lambda event object:

  The function `handler` takes two arguments, `event` and `context`. An *event* in Lambda is a JSON formatted document that contains data for your function to process.

  If your function is invoked by another AWS service, the event object contains information about the event that caused the invocation. For example, if your function is invoked when

an object is uploaded to an Amazon Simple Storage Service (Amazon S3) bucket, the event contains the name of the bucket and the object key.

In this example, you'll create an event in the console by entering a JSON formatted document with two key-value pairs.

- The Lambda context object:

  The second argument that your function takes is `context`. Lambda passes the *context object* to your function automatically. The context object contains information about the function invocation and execution environment.

  You can use the context object to output information about your function's invocation for monitoring purposes. In this example, your function uses the `logGroupName` parameter to output the name of its CloudWatch log group.

  To learn more about the Lambda context object in Node.js, see the section called "Context".

- Logging in Lambda:

  With Node.js, you can use console methods like `console.log` and `console.error` to send information to your function's log. The example code uses `console.log` statements to output the calculated area and the name of the function's CloudWatch Logs group. You can also use any logging library that writes to `stdout` or `stderr`.

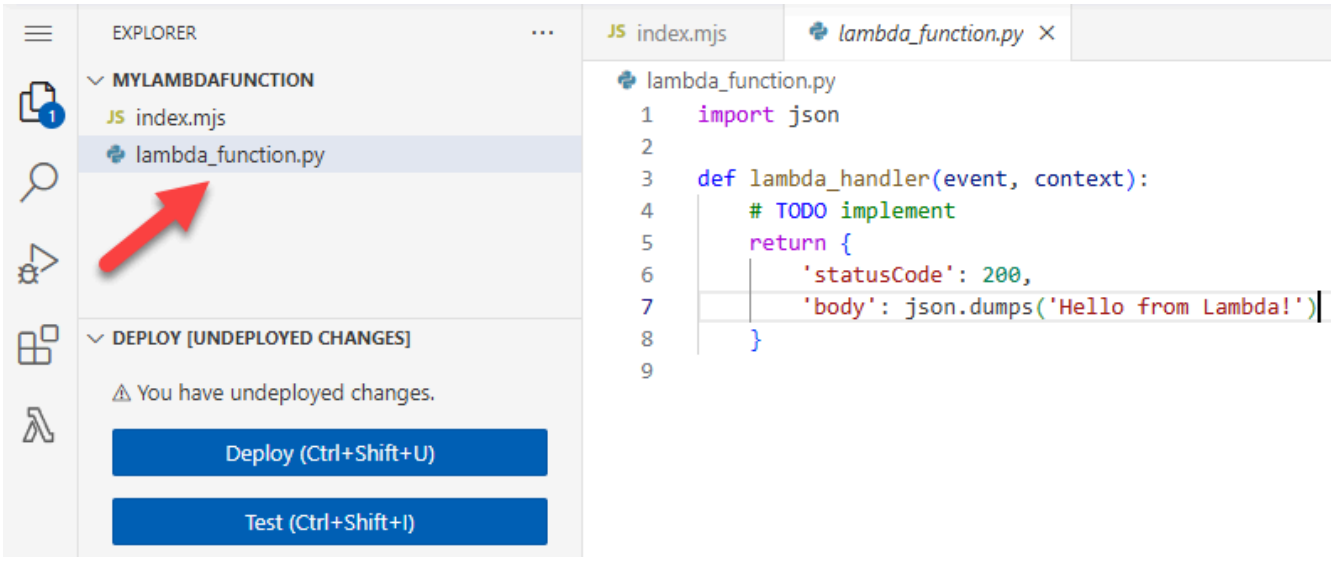
  To learn more, see the section called "Logging". To learn about logging in other runtimes, see the 'Building with' pages for the runtimes you're interested in.

Python

To modify the code in the console

1. Choose the **Code** tab.

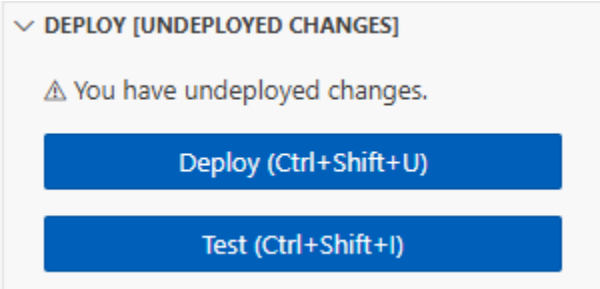
   In the console's built-in code editor, you should see the function code that Lambda created. If you don't see the **lambda_function.py** tab in the code editor, select **lambda_function.py** in the file explorer as shown on the following diagram.



2. Paste the following code into the **lambda_function.py** tab, replacing the code that Lambda created.

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
    width = event['width']

    area = calculate_area(length, width)
    print(f"The area is {area}")

    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. In the **DEPLOY** section, choose **Deploy** to update your function's code:



Understanding your function code

Before you move to the next step, let's take a moment to look at the function code and understand some key Lambda concepts.

- The Lambda handler:

  Your Lambda function contains a Python function named `lambda_handler`. A Lambda function in Python can contain more than one Python function, but the *handler* function is always the entry point to your code. When your function is invoked, Lambda runs this method.

  When you created your Hello world function using the console, Lambda automatically set the name of the handler method for your function to `lambda_handler`. Be sure not to edit the name of this Python function. If you do, Lambda won't be able to run your code when you invoke your function.

  To learn more about the Lambda handler in Python, see the section called "Handler".

- The Lambda event object:

  The function `lambda_handler` takes two arguments, `event` and `context`. An *event* in Lambda is a JSON formatted document that contains data for your function to process.

  If your function is invoked by another AWS service, the event object contains information about the event that caused the invocation. For example, if your function is invoked when an object is uploaded to an Amazon Simple Storage Service (Amazon S3) bucket, the event contains the name of the bucket and the object key.

  In this example, you'll create an event in the console by entering a JSON formatted document with two key-value pairs.

- The Lambda context object:

The second argument that your function takes is `context`. Lambda passes the *context object* to your function automatically. The context object contains information about the function invocation and execution environment.

You can use the context object to output information about your function's invocation for monitoring purposes. In this example, your function uses the `log_group_name` parameter to output the name of its CloudWatch log group.

To learn more about the Lambda context object in Python, see the section called "Context".

- Logging in Lambda:

  With Python, you can use either a `print` statement or a Python logging library to send information to your function's log. To illustrate the difference in what's captured, the example code uses both methods. In a production application, we recommend that you use a logging library.

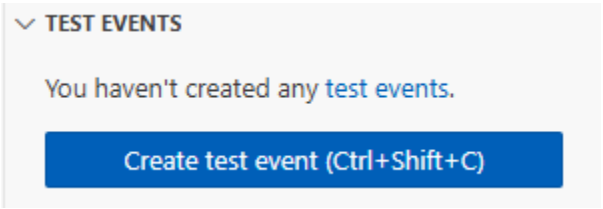
  To learn more, see the section called "Logging". To learn about logging in other runtimes, see the 'Building with' pages for the runtimes you're interested in.

Invoke the Lambda function using the console code editor

To invoke your function using the Lambda console code editor, create a test event to send to your function. The event is a JSON formatted document containing two key-value pairs with the keys `"length"` and `"width"`.

To create the test event

1. In the **TEST EVENTS** section of the console code editor, choose **Create test event**.



2. For **Event Name**, enter **`myTestEvent`**.
3. In the **Event JSON** section, replace the default JSON with the following:

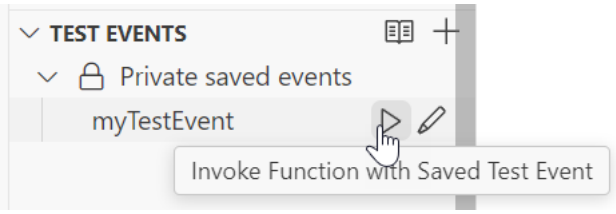
```
{
```

```
  "length": 6,
  "width": 7
}
```

4. Choose **Save**.

To test your function and view invocation records

In the **TEST EVENTS** section of the console code editor, choose the run icon next to your test event:



When your function finishes running, the response and function logs are displayed in the **OUTPUT** tab. You should see results similar to the following:

Node.js

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\":42}"

Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2024-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is 42
2024-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch log
 group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

Python

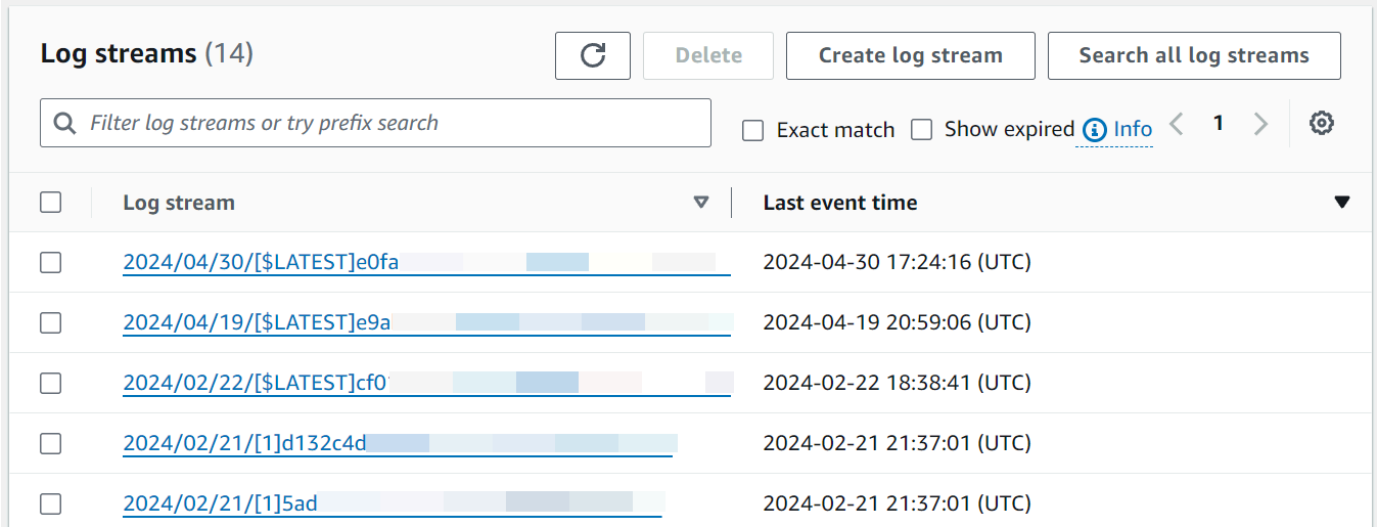
```
Status: Succeeded
Test Event Name: myTestEvent
```

```
Response
"{\"area\": 42}"

Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2024-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch logs
 group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

When you invoke your function outside of the Lambda console, you must use CloudWatch Logs to view your function's execution results.

To view your function's invocation records in CloudWatch Logs

1. Open the Log groups page of the CloudWatch console.
2. Choose the log group for your function (`/aws/lambda/myLambdaFunction`). This is the log group name that your function printed to the console.
3. Scroll down and choose the **Log stream** for the function invocations you want to look at.



You should see output similar to the following:

Node.js

```
INIT_START Runtime Version: nodejs:22.v13    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
2024-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
 is 42
2024-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
 CloudWatch log group:  /aws/lambda/myLambdaFunction
END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
 Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
 Duration: 178.85 ms
```

Python

```
INIT_START Runtime Version: python:3.13.v16    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
The area is 42
[INFO] 2024-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

Clean up

When you're finished working with the example function, delete it. You can also delete the log group that stores the function's logs, and the execution role that the console created.

To delete the Lambda function

1. Open the Functions page of the Lambda console.
2. Select the function that you created.
3. Choose **Actions**, **Delete**.
4. Type **`confirm`** in the text input field and choose **Delete**.

To delete the log group

1. Open the Log groups page of the CloudWatch console.
2. Select the function's log group (`/aws/lambda/myLambdaFunction`).
3. Choose **Actions**, **Delete log group(s)**.
4. In the **Delete log group(s)** dialog box, choose **Delete**.

To delete the execution role

1. Open the Roles page of the AWS Identity and Access Management (IAM) console.
2. Select the function's execution role (for example, `myLambdaFunction-role-31exxmpl`).
3. Choose **Delete**.
4. In the **Delete role** dialog box, enter the role name, and then choose **Delete**.

Additional resources and next steps

Now that you've created and tested a simple Lambda function using the console, take these next steps:

- Learn to add dependencies to your function and deploy it using a .zip deployment package. Choose your preferred language from the following links.

  Node.js

  the section called "Deploy .zip file archives"

  Typescript

  the section called "Deploy .zip file archives"

  Python

  the section called "Deploy .zip file archives"

  Ruby

  the section called "Deploy .zip file archives"

  Java

  the section called "Deploy .zip file archives"

Go

the section called “Deploy .zip file archives”

C#

the section called “Deployment package”

- To learn how to invoke a Lambda function using another AWS service, see Tutorial: Using an Amazon S3 trigger to invoke a Lambda function.
- Choose one of the following tutorials for more complex examples of using Lambda with other AWS services.
  - Tutorial: Using Lambda with API Gateway: Create an Amazon API Gateway REST API that invokes a Lambda function.
  - Using a Lambda function to access an Amazon RDS database: Use a Lambda function to write data to an Amazon Relational Database Service (Amazon RDS) database through RDS Proxy.
  - Using an Amazon S3 trigger to create thumbnail images: Use a Lambda function to create a thumbnail every time an image file is uploaded to an Amazon S3 bucket.

Key Lambda concepts

This chapter describes key concepts in Lambda:

- Basic Lambda concepts explains basics such as functions, triggers, events, runtimes, and deployment packages.
- Programming model explains how Lambda interacts with your code.
- Execution environment explains the environment Lambda uses to run your code.
- Event-driven architectures describes the most commonly used design paradigm for serverless applications built using Lambda functions.
- Application design explains various design best practices for Lambda-based applications.
- Frequently asked questions is a curated list of common FAQs about Lambda.