Deploying Your First Lambda Container

**SPL-TF-200-CTALAM-1 - Version 1.0.13**

© 2024 Amazon Web Services, Inc. or its affiliates. All rights reserved. This work may not be reproduced or redistributed, in whole or in part, without prior written permission from Amazon Web Services, Inc. Commercial copying, lending, or selling is prohibited. All trademarks are the property of their owners.

Note: Do not include any personal, identifying, or confidential information into the lab environment. Information entered may be visible to others.

Corrections, feedback, or other questions? Contact us at [*AWS Training and Certification*](https://support.aws.amazon.com/#/contacts/aws-training).

**Lab Overview**

You can package and deploy Lambda functions as container images of up to 10 GB in size. This allows for building and deploying larger workloads that rely on sizable dependencies, such as machine learning or data intensive workloads. Just like functions packaged as ZIP archives, functions deployed as container images benefit from the same operational simplicity, automatic scaling, high availability, and native integrations with many services.

In this lab, you will deploy a sample application to AWS Lambda utilizing a container image. You will make use of AWS Cloud9 as your cloud-based integrated development environment (IDE), where you will create the sample application, test it locally, upload the container image to Amazon Elastic Container Registry (Amazon ECR), and finally deploy it to AWS Lambda using the AWS Serverless Application Model (SAM).

This lab will be completed using Javascript, and you will start with an AWS base image for Lambda to build a container image for a new Lambda function. Note that you can also create an image from an [alternative base image](https://docs.aws.amazon.com/lambda/latest/dg/images-create.html#images-create-from-alt).

To create your container image, you can use any development tool that supports one of the following container image manifest formats:

* Docker image manifest V2, schema 2 (used with Docker version 1.10 and newer)
* OCI Specifications (v1.0.0 and up)

For example, you can use the Docker CLI to build, test, and deploy your container images. In this lab, we will be using the [AWS SAM CLI](https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-getting-started.html) for this purpose.

This lab will take approximately 90 minutes to complete.

TOPICS COVERED

By the end of this lab, you will be able to:

* Create and test a Lambda function locally in your Cloud9 development environment using the Lambda Runtime Interface Emulator.
* Upload your Lambda container image to the Amazon Elastic Container Registry (Amazon ECR).
* Modify and re-upload the container used in your Lambda function.
* Invoke your Lambda function via an Amazon API Gateway endpoint.

TECHNICAL KNOWLEDGE PREREQUISITES

To successfully complete this lab, you should be familiar with basic navigation of the AWS Management Console and be comfortable editing scripts using a text editor.

**Start lab**

1. To launch the lab, at the top of the page, choose **Start lab**.

**Caution:** You must wait for the provisioned AWS services to be ready before you can continue.

1. To open the lab, choose **Open Console**.

You are automatically signed in to the AWS Management Console in a new web browser tab.

**WARNING:** **Do not change the Region unless instructed.**

COMMON SIGN-IN ERRORS

**Error: You must first sign out**



If you see the message, **You must first log out before logging into a different AWS account:**

* Choose the **click here** link.
* Close your **Amazon Web Services Sign In** web browser tab and return to your initial lab page.
* Choose **Open Console** again.

**Error: Choosing Start Lab has no effect**

In some cases, certain pop-up or script blocker web browser extensions might prevent the **Start Lab** button from working as intended. If you experience an issue starting the lab:

* Add the lab domain name to your pop-up or script blocker’s allow list or turn it off.
* Refresh the page and try again.

**Task 1: Creating and Locally Testing your Function**

TASK 1.1: GETTING STARTED WITH CLOUD9

1. Navigate to the Cloud9 Console by searching for **Cloud9** in the unified search box at the top of the screen. Alternatively, in the AWS Management Console, on the Services menu, choose **Cloud9**.
2. Launch the existing environment by navigating to **My environments** in the left sidebar. If the sidebar is minimized, you can click  at the upper-left to open it.
3. A Cloud9 environment has been created for you with a name similar to **MyC9Environment-XXXXXXXXXXXX**. Click the **Open**

 If you are not able to find any instances, ensure you are in the correct region as specified for this lab.

Give the page a few minutes to finish loading. Take a moment to familiarize yourself with the AWS Cloud9 IDE.

AWS Cloud9 allows you to run your development environment on a managed Amazon EC2 Instance and interact with the environment with just a web browser. You can write, run, debug, and collaborate on applications without needing to install any additional software.

 If a box appears saying the following: **.c9/project.settings have been changed on disk** you can disregard this message and click **Discard**

 If an Error box appears saying the following: **Unable to register Service Workers…** you can disregard this message and click **OK**

 You may close out of the **Welcome** tab in the Cloud9 window.

1. Verify that necessary packages are installed by running the following command in the terminal window near the bottom of the screen. You can open a new terminal at any time by selecting **Window** > **New Terminal**.

aws --version && sam --version && docker --version && node --version && npm --version

You should see an output similar to the following, indicating that the required packages are installed in your Cloud9 Environmnent.

~/environment $ aws --version && sam --version && docker --version && node --version && npm --version

aws-cli/2.15.36 Python/3.11.8 Linux/6.1.82-99.168.amzn2023.x86\_64 exe/x86\_64.amzn.2023 prompt/off

SAM CLI, version 1.112.0

Docker version 25.0.3, build 4debf41

v20.12.1

10.5.1

TASK 1.2: CREATING YOUR FUNCTION WITH A NEW SAM APPLICATION

The AWS Serverless Application Model (AWS SAM) is an open-source framework that you can use to build serverless applications on AWS. As seen in the previous step, the AWS SAM CLI (Command Line Interface) comes preinstalled on Cloud9 instances, allowing us to quickly author, test, and debug SAM-based serverless applications. AWS SAM also supports the ability to manage, build, and deploy Lambda functions using container images.

We will initialize our project using SAM and begin to add our core code for our function.

1. Run the following command to initialize a new SAM application.

sam init

1. For the guided SAM Application wizard, choose the following options:
   * **Choice:** AWS Quick Start Templates
   * **Package type:** Hello World Example
   * **Use the most popular runtime and package type? (Python and zip) [y/N]:** N
   * **Which runtime would you like to use?:** nodejs18.x
   * **What package type would you like to use?:** Image
   * **Would you like to enable X-Ray tracing on the function(s) in your application? [y/N]:** N
   * **Would you like to enable monitoring using CloudWatch Application Insights? [y/N]:** N
   * **Would you like to set Structured Logging in JSON format on your Lambda functions? [y/N]:** N
   * **Project name:**

getletter

The AWS Cloud9 IDE has many of the built-in features you would expect in an Integrated Development Environment, including the ability to manage directories of files, support for syntax highlighting, and many other features.

1. A new directory named **getletter** will appear in the left navigation pane. Click the down arrow next to the folder to reveal the folder’s contents:

getletter/

├─ events/

├─ hello-world/

│ ├─ tests/

│ ├─ app.mjs

│ ├─ Dockerfile

├─ README.md

├─ template.yaml

1. Open the **template.yaml** file in the root of the project.

AWS SAM templates are an extension of AWS CloudFormation templates, with some additional components that make them easier to work with. The AWS SAM template includes two values that are required when working with container images:

* **PackageType: Image** tells AWS SAM that this function is using container images for packaging.
* **Metadata** helps AWS SAM manage the container images.

1. With the **template.yaml** file open, replace everything in the file with the following code and **save** the file.

AWSTemplateFormatVersion: '2010-09-09'

Transform: AWS::Serverless-2016-10-31

Description: >

getletter

SAM Template for our letter writer

Globals:

Function:

Timeout: 5

Resources:

GetLetter:

Type: AWS::Serverless::Function

Properties:

PackageType: Image

Role: !Sub 'arn:aws:iam::${AWS::AccountId}:role/lambda-execution-role'

Metadata:

DockerTag: nodejs18.x-v1

DockerContext: ./hello-world

Dockerfile: Dockerfile

Outputs:

GetLetter:

Description: "Letter Writer Lambda Function ARN"

Value: !GetAtt GetLetter.Arn

1. Click  next to the **hello-world** folder and open the **app.mjs** file. Replace all existing code in the file with the following code and save it.

*Note that the file will show errors until we install required packages in the next step.*

import PDFDocument from 'pdfkit';

import faker from 'faker';

export const lambdaHandler = async (event, context) => {

const pdfBuffer = await new Promise(resolve => {

const doc = new PDFDocument();

const randomName = faker.name.findName();

//const letterdate = new Date(faker.date.past()).toLocaleDateString('en-us');

doc.text(randomName, { align: 'right' });

doc.text(faker.address.streetAddress(), { align: 'right' });

doc.text(faker.address.secondaryAddress(), { align: 'right' });

doc.text(faker.address.zipCode() + ' ' + faker.address.city(), { align: 'right' });

doc.moveDown();

//doc.text(letterdate, { align: 'left' });

//doc.moveDown();

doc.text('Dear ' + randomName + ',');

doc.moveDown();

for(let i = 0; i < 3; i++) {

doc.text(faker.lorem.paragraph());

doc.moveDown();

}

doc.text(faker.name.findName(), { align: 'right' });

doc.end();

const buffers = []

doc.on("data", buffers.push.bind(buffers))

doc.on("end", () => {

const pdfData = Buffer.concat(buffers)

resolve(pdfData)

})

})

const pdfBase64 = pdfBuffer.toString('base64');

const response = {

statusCode: 200,

headers: {

'Content-Length': Buffer.byteLength(pdfBase64),

'Content-Type': 'application/pdf',

'Content-disposition': 'attachment;filename=test.pdf'

},

isBase64Encoded: true,

body: pdfBase64

};

return response;

};

This code performs a simple task. It creates a fictitious letter in PDF form, and we will deploy it to our Lambda function later on.

1. Change into your **hello-world** folder by running the following command in the terminal:

cd getletter/hello-world/

1. Install the packages required by your code by running the following commands in the terminal window:

npm init -y && npm install pdfkit faker

npm audit fix --force

npm fund

You should see an output similar to the following, indicating that the required packages are installed:

Wrote to /home/ec2-user/environment/getletter/hello-world/package.json:

{

"name": "hello\_world",

"version": "1.0.0",

"description": "hello world sample for NodeJS",

"main": "app.mjs",

"repository": {

"type": "git",

"url": "git+https://github.com/awslabs/aws-sam-cli.git#develop"

},

"author": "SAM CLI",

"license": "MIT",

"dependencies": {

"axios": "^0.21.1"

},

"scripts": {

"test": "mocha tests/unit/"

},

"devDependencies": {

"chai": "^4.2.0",

"mocha": "^8.2.1"

},

"bugs": {

"url": "https://github.com/awslabs/aws-sam-cli/issues"

},

"homepage": "https://github.com/awslabs/aws-sam-cli/tree/develop#readme",

"directories": {

"test": "tests"

},

"keywords": []

}

Your function is now created! Next, we will utilize Docker to containerize our function and get it ready to deploy to AWS Lambda. Docker requires a template file called a **Dockerfile** which is a text document that contains all the commands a user could call on the command line to assemble an image.

TASK 1.3: CREATING YOUR DOCKER IMAGE

1. Earlier, we utilized AWS SAM CLI to initialize our SAM application with a Package Type of **Image**. This configured our template and also created a Dockerfile which is necessary for creating our container.

*Note that a new Dockerfile will have been created in the left sidebar and the file will have no extension.*

1. Open the **Dockerfile** and review the code that has been added for you. These four simple lines are all that are required in our Dockerfile template for Docker to create a container image.

FROM public.ecr.aws/lambda/nodejs:18

COPY app.mjs package.json ./

RUN npm install

CMD [ "app.lambdaHandler" ]

In this file, the first line (FROM) is specifying that Docker should use a base image from Amazon ECR. If we wanted to source our container image from Docker Hub, we could change the first line to

FROM amazon/aws-lambda-nodejs:18

 to specifiy that image location. You can also create an image from an [alternative base image](https://docs.aws.amazon.com/lambda/latest/dg/images-create.html#images-create-from-alt).

When built, Docker will execute the COPY, RUN, and CMD instructions to pull in the rest of your application files (copying the **app.mjs** code and **package.json** files to the container image), install listed dependencies into your container image, and specify the invocation handler for your Lambda function.

1. Build the Dockerfile into a Docker container image by running the following command in the terminal window:

docker build -t getletter .

This step can take a minute or two to run through each of the 4 steps, be patient!

 Please ignore any warnings about **OPTIONAL DEPENDENCY**.

Once you see the message **Successfully tagged getletter:latest**, you have successfully built your local app into a Docker Image.

1. List the Docker images available by running the following command in the terminal window. This will show you details on the latest images (including our getletter image) such as the image sizes and when they were created.

docker images

1. Since the Docker image has been built, we can now run a new Docker container from the image. Run the following command to test our PDF-generating function:

docker run -p 9000:8080 getletter:latest

This command will cause the terminal to wait, as the container is now running. Earlier, we specified the **Lambda base image** to be used for our Docker container image.

The **AWS base images for Lambda** include a component called the **AWS Lambda Runtime Interface Emulator (RIE)**. If you use an alternate base image, you can test your image without adding RIE to the image. You can also build the RIE component into your base image. AWS provides an open-sourced RIE component on the AWS GitHub repository.

The AWS Lambda RIE is a proxy for the Lambda Runtime API that allows you to locally test your Lambda function packaged as a container image. The emulator is a lightweight web server that converts HTTP requests into JSON events to pass to the Lambda function in the container image.

You can read more about the [AWS Lambda Runtime Interface Emulator here](https://docs.aws.amazon.com/lambda/latest/dg/images-test.html).

Now that our Docker container is running, it is also running the Lambda RIE included in the base image, which allows you to test your function locally without needing to test on the AWS Lambda service.

1. Open a new terminal window and run the following code to run a test invocation, using the Lambda RIE:

curl -XPOST "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'

You will see an encoded response starting with a **{“statusCode”:200}**, which is a Base64 representation of the PDF generated by your function. The previous terminal window will show the request information from this invocation.

Earlier, we ran

docker build

 to build our container image. Conveniently, AWS SAM CLI has the ability to build our container image as well. While it is not necessary to run **docker build** when using AWS SAM CLI, it did give us the ability to locally run and test our image.

1. Close the terminal window. Return to the previous terminal window titled **docker -** and press **CTRL + C** to interrupt the current Docker process.
2. Use the following command to navigate to getletter directory and build your container image again using AWS SAM:

cd /home/ec2-user/environment/getletter

sam build

1. You can also test your function using AWS SAM CLI with the following command:

sam local invoke GetLetter

Again, you will see an encoded response starting with a **{“statusCode”:200}**.

**Task 2: Deploying and Testing Your App**

TASK 2.1: CREATING YOUR ECR REPOSITORY AND PUSHING YOUR IMAGE

Now that you’ve built your conainer image using AWS SAM and tested your function locally running on a container, let’s get it ready to deploy to a Lambda function which can be invoked using the Lambda service. First we’ll create a repository to host a copy of our local container image so Lambda can refer to that copy.

1. Create an Amazon Elastic Container Registry (ECR) repository by running the following code in your terminal window. If a process is still running in your terminal window, you can press **CTRL + C** to interrupt it.

aws ecr create-repository \

--repository-name getletter \

--image-scanning-configuration scanOnPush=true

The output from this command will be similar to the below. Be sure to copy your **repositoryUri** between the quotes for use in the following steps.

{

"repository": {

"repositoryArn": "arn:aws:ecr:us-east-1:111111111111:repository/getletter",

"registryId": "111111111111",

"repositoryName": "getletter",

"repositoryUri": "111111111111.dkr.ecr.us-east-1.amazonaws.com/getletter",

"createdAt": 1633448589.0,

"imageTagMutability": "MUTABLE",

"imageScanningConfiguration": {

"scanOnPush": true

},

"encryptionConfiguration": {

"encryptionType": "AES256"

}

}

}

1. Run the following command to tag the local Docker container image to indicate which container you would like to push. Replace

<repositoryUri>

 with the value from the previous step. There will be no output.

docker tag getletter:latest <repositoryUri>

Next, we’ll get authenticate the Docker CLI to our default registry. This will allow us to use Docker commands to push and pull images with Amazon ECR. The AWS CLI provides a

get-login-password

 command to simplify the authentication process.

*To simplify the login process, we are using a bash pipe operator to supply the output of the*

*get-login-password*

*command as stdin to the*

*docker login*

*command.*

1. Run the following command, replacing

<repositoryUri>

 with the **repositoryUri** you copied earlier.

aws ecr get-login-password \

| docker login \

--username AWS \

--password-stdin <repositoryUri>

You should see a message saying **Login Succeeded**.

Once you have authenticated the Docker CLI to the ECR registry, you can now push the local image into the registry.

1. Run the following command, replacing your

<repositoryUri>

 copied earlier. Be sure to replace the angle brackets as well.

docker push <repositoryUri>:latest

This step will take another minute or two. If your ouput is similar to the following, you have successfully pushed your container to your ECR repository and you can now use the container for your Lambda function:

The push refers to repository [111111111111.dkr.ecr.us-east-1.amazonaws.com/getletter]

c1e7990681a4: Pushed

dcaeec1ca227: Pushed

d93a8fae1256: Pushed

59c6fcf779e8: Pushed

33d73f3b21b2: Pushed

07979f5f253a: Pushed

b83f65c414d0: Pushed

fdab84953703: Pushed

latest: digest: sha256:c09b4179afa737a9559f559dafda62198b9ea2f6f2bb96cfb59a33064472dc4a size: 2000

TASK 2.2: USING SAM TO UPLOAD YOUR ECR IMAGE AND DEPLOY YOUR LAMBDA FUNCTION

While it is possible to manually push our container to an ECR repository (as we have done in the previous steps), this is also not necessary with AWS SAM. The AWS SAM CLI provides an easy method of building and deploying our application, which includes tagging and pushing our container to the specified repository.

After building the application locally and creating a repository for the container image, we’ll utilize AWS SAM deploy the application.

1. Open the **samconfig.toml** file in the root of the project and update the values for **[default.deploy.parameters]** as below and **save** the file.

* **confirm\_changeset**:

false

* **resolve\_s3**:

false

* **resolve\_image\_repos**:

false

1. Run the following command to deploy your SAM application. Be sure to replace

<S3Bucket>

 with the **S3Bucket** name shown to the left of the instructions, as well as

<repositoryUri>

 with the **repositoryUri** copied earlier.

sam deploy \

--stack-name getletter \

--s3-bucket <S3Bucket> \

--capabilities CAPABILITY\_NAMED\_IAM CAPABILITY\_AUTO\_EXPAND \

--image-repository <repositoryUri>

This may take around five minutes to complete, please be patient!

During the

sam deploy

 process, SAM is deploying a new CloudFormation Stack, using the template you initialized earlier. You will see an output of the CloudFormation stack changeset, along with the status of resources being created for you.

Once you see the message **Successfully created/updated stack - getletter …**, your SAM application has been deployed, along with the corresponding Lambda function and container image from ECR.

TASK 2.3: ADDING AN API GATEWAY TRIGGER TO YOUR LAMBDA FUNCTION

1. Navigate back to the AWS Management Console either by clicking **Open Console** at the upper right of these instructions, or by navigating back to the open tab in your browser.
2. Navigate to the Lambda Console by searching for **Lambda** in the unified search box at the top of the screen.
3. Once in the Lambda console, click on **Applications** in the left sidebar.
4. Click on the new application **getletter** created by SAM. Scroll to the bottom of the Overview page and click on **GetLetter** under **Resources**.

On the main function page for **GetLetter** notice that there is an option to **+ Add Trigger** to our Lambda function. No trigger is currently defined, so we will add one. You will also notice an **Image URI** is already specified under the **Image** section of the page.

1. Click **+ Add Trigger**
2. On the **Trigger configuration** page, select **API Gateway** and configure with the following settings:
   * **API:** Create a new API
   * **API type:** HTTP API
   * **Security:** Open
3. Click **Add**

Although we have manually created our API Gateway trigger in the Lambda console, note that it is also possible to set this up in our SAM template.

Give Lambda a minute to set up the API Gateway trigger for your function. When you see a success message at the top of the page, you can proceed to the next step.

1. Under the **Configuration** tab, click **Triggers**.

Here you will see the API Gateway endpoint provided by configuring the trigger, which should look similar to the following:

https://abc0123def.execute-api.us-east-1.amazonaws.com/default/getletter-GetLetter-abc012DEF345

1. Copy the entire API Endpoint and paste it into the URL bar of a new browser tab.

In about 10-20 seconds, your browser should begin downloading the PDF generated by your function or display the PDF in a new window.

1. Open the PDF to see the output file of your function. The output PDF file will include generated placeholder text and resemble a letter.

 If your PDF will not open or does not display properly, you may have selected **API type: REST** for your API Gateway configuration. In this case, you can redo the previous step. Click **Add Trigger** select API Gateway, **Create an API - HTTP API** and set **Security** to **Open**. Once you add this new trigger, you will have a new URL to use as your API Gateway endpoint.

**Task 3: Modifying Your App**

Let’s make one final revision to our app so that our generated letters include a date.

1. Navigate back to the Cloud9 IDE tab and open your **app.mjs** once again.
2. Replace all of the content of the file with the following code. We have now added a date to the letter so our recipient will know when it was written.

import PDFDocument from 'pdfkit';

import faker from 'faker';

export const lambdaHandler = async (event, context) => {

const pdfBuffer = await new Promise(resolve => {

const doc = new PDFDocument();

const randomName = faker.name.findName();

const letterdate = new Date(faker.date.past()).toLocaleDateString('en-us');

doc.text(randomName, { align: 'right' });

doc.text(faker.address.streetAddress(), { align: 'right' });

doc.text(faker.address.secondaryAddress(), { align: 'right' });

doc.text(faker.address.zipCode() + ' ' + faker.address.city(), { align: 'right' });

doc.moveDown();

doc.text(letterdate, { align: 'left' });

doc.moveDown();

doc.text('Dear ' + randomName + ',');

doc.moveDown();

for(let i = 0; i < 3; i++) {

doc.text(faker.lorem.paragraph());

doc.moveDown();

}

doc.text(faker.name.findName(), { align: 'right' });

doc.end();

const buffers = []

doc.on("data", buffers.push.bind(buffers))

doc.on("end", () => {

const pdfData = Buffer.concat(buffers)

resolve(pdfData)

})

})

const pdfBase64 = pdfBuffer.toString('base64');

const response = {

statusCode: 200,

headers: {

'Content-Length': Buffer.byteLength(pdfBase64),

'Content-Type': 'application/pdf',

'Content-disposition': 'attachment;filename=test.pdf'

},

isBase64Encoded: true,

body: pdfBase64

};

return response;

};

Once your function has been changed, you will need to re-build, re-tag, and re-deploy your updated container image. Remember that we can use AWS SAM to take care of this for us.

1. Run the following command to build your modified application and container image:

sam build

1. Run the below command (the same as used earlier) to deploy your modified application using SAM again. Be sure to replace

<S3Bucket>

 with the **S3Bucket** name shown to the left of the instructions, as well as

<repositoryUri>

 with the **repositoryUri** copied earlier.

sam deploy \

--stack-name getletter \

--s3-bucket <S3Bucket> \

--capabilities CAPABILITY\_NAMED\_IAM CAPABILITY\_AUTO\_EXPAND \

--image-repository <repositoryUri>

SAM will detect the changes made, removing, adding, or updating resources as necessary. Within a few minutes, your function will be updated. Once you see the message **Successfully created/updated stack - getletter in < region >**, your SAM application has been re-deployed, along with the corresponding Lambda function and container image from ECR.

1. Test the API Gateway endpoint URL again by copying it from the Lambda Triggers page and pasting it into a new browser tab URL. Your PDF should now display a date, reflecting changes have been successfully made to the container.

**Conclusion**

 Congratulations! You now have successfully:

* Familiarized yourself with Lambda Container Image Support
* Utilized the Lambda Runtime Interface Emulator (RIE) for testing Lambda Functions locally
* Created, deployed, and modified a SAM Application using the AWS SAM CLI
* Utilized AWS Cloud9, Amazon API Gateway, AWS Lambda, Amazon Elastic Container Registry (ECR), and AWS CloudFormation to build and deploy a containerized function

**End lab**

Follow these steps to close the console and end your lab.

1. Return to the **AWS Management Console**.
2. At the upper-right corner of the page, choose **AWSLabsUser**, and then choose **Sign out**.
3. Choose **End lab** and then confirm that you want to end your lab.

**Additional Resources**

* For more information about how to use AWS Lambda, see [AWS Lambda Documentation](https://docs.aws.amazon.com/lambda/index.html).
* For more information about AWS Lambda Container Image Support, see [the AWS Blog](https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support/).

For more information about AWS Training and Certification, see [*https://aws.amazon.com/training/*](https://aws.amazon.com/training/).

*Your feedback is welcome and appreciated.*  
If you would like to share any feedback, suggestions, or corrections, please provide the details in our [*AWS Training and Certification Contact Form*](https://support.aws.amazon.com/#/contacts/aws-training).