Tutorial 4 (Part 1) - AWS Lambda and Elastic Container Service

---

Objectives

In this tutorial, you are expected to learn to

- Create a HelloWorld serverless function with AWS Lambda Console using Python 3.7.
- Create Lambda functions in Python 3.8 using AWS CLI
- Deploy a Docker-enabled application to Amazon ECS

Task 1: Create a HelloWorld serverless function with AWS Lambda

Introduction

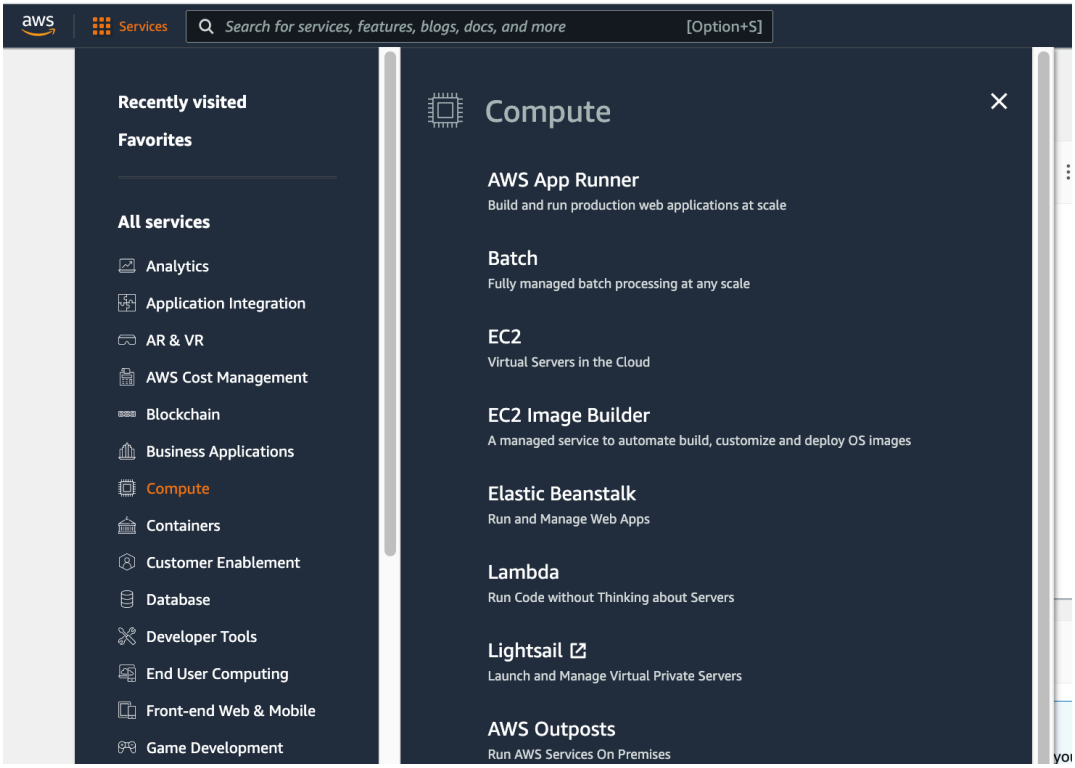
In this tutorial, you will learn the basics of running code on AWS Lambda without provisioning or managing servers. We will walk through how to create a Hello World Lambda function using the AWS Lambda console. We will then show you how to manually invoke the Lambda function using sample event data and review your output metrics.

Everything done in this tutorial is free tier eligible.

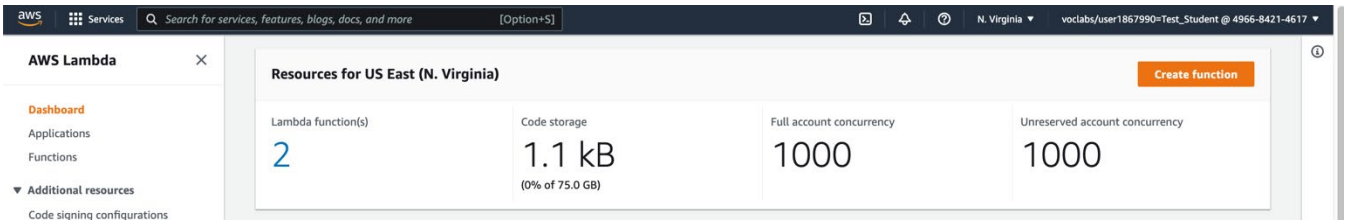
Step 1: Enter the Lambda Console.

a. Go to the AWS Academy, start a new lab session. Then, browse the AWS Management Console attached to your lab session. We learnt to do this in **Week-2 Lectorial**.

b. Find **AWS Lambda** under the *Compute* category either by using the *search bar* available in the AWS Management Console or by expanding the *Services* drop-downbox. Click to go to AWS Lambda Console or via other suitable means.



Step 2: From AWS Lambda dashboard, select **Create function**.



Step 3: In the “Create function” wizard, select **Use a blueprint** to run a sample Lambda function provided by AWS.

Blueprints provide example code to do some minimal processing. Most blueprints process events from specific event sources, such as Amazon S3, DynamoDB, or a custom application.

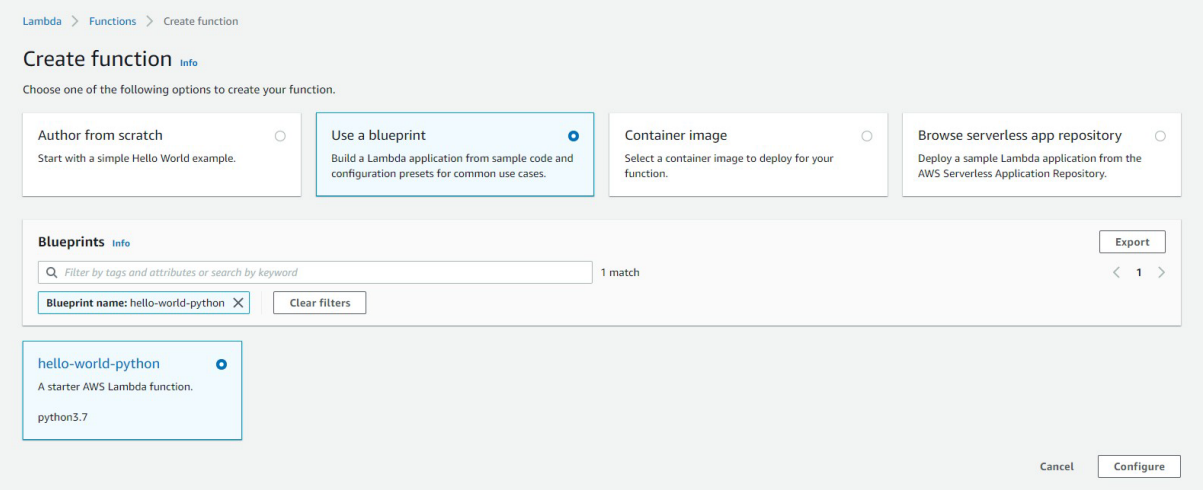
a. In the AWS Lambda console, select Create a Function.

Note: The console shows this page only if you do not have any Lambda functions created. If you have created functions already, you will see the *Lambda* > *Functions* page. On the list page, choose *Create a function* to go to the Create function page.

b. Select *Use a blueprint* out of the options available to create a function.

c. In the *Blueprints* search box, type in *hello-world-python* and select the hello-world-python blueprint.

d. Then click *Configure*.



Step 3: Configure and Create Your Lambda Function.

A Lambda function consists of code you provide, associated dependencies, and configuration. The configuration information you provide includes the compute resources you want to allocate (for example, memory), execution timeout, and an IAM role that AWS Lambda can assume to execute your Lambda function on your behalf.

a. You will now enter *Basic Information* about your Lambda function,

such as

- Name: You can name your Lambda function here. For this tutorial, enter *hello-world-python*.

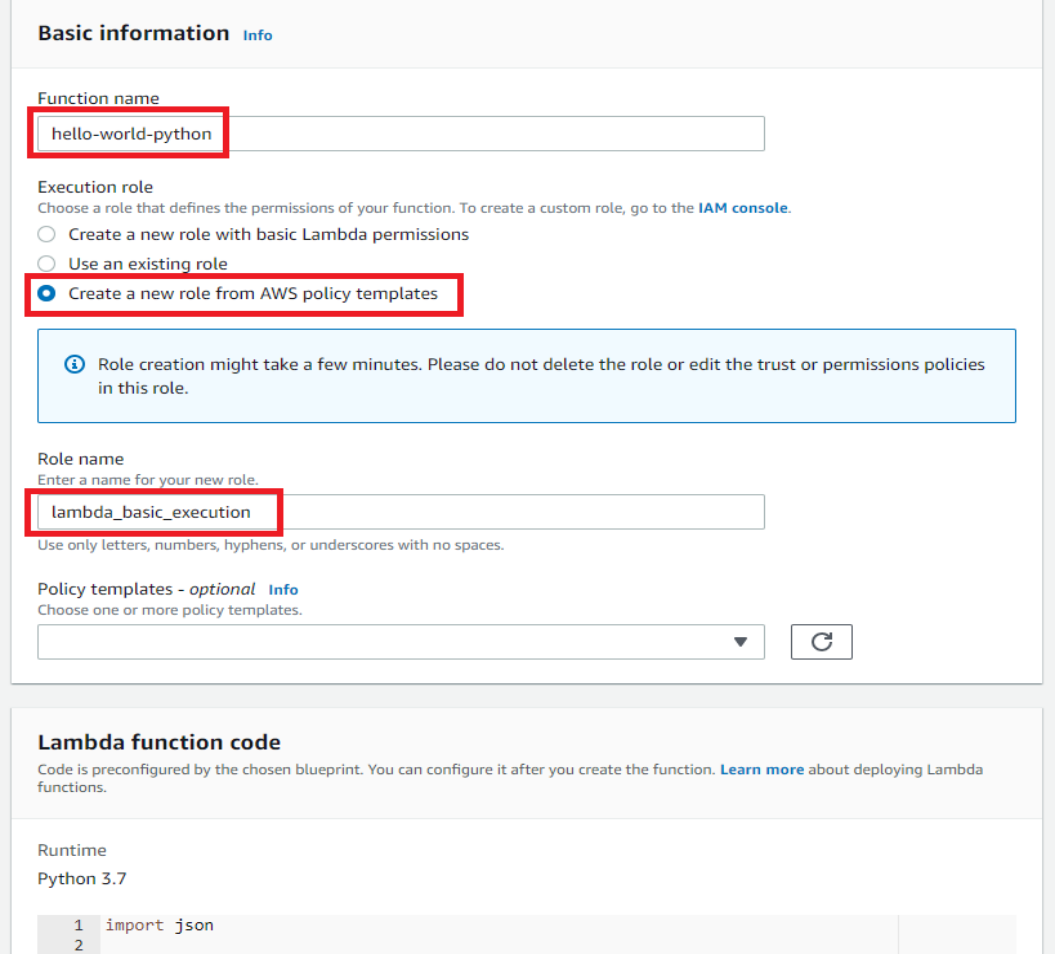
- Role: You can either create an IAM role (referred as the execution role) with the necessary permissions that AWS Lambda can assume to invoke your Lambda function on your behalf using *Create new role from template(s)* or select an existing Role using *Use an existing role* option.

  Note: AWS Academy provides restricted access to students on creating roles as well as several other Identity and Access Management related tasks. Therefore, in this exercise, we will use **Use an existing role** option and select a role that AWS Academy already created for us.

- Role name: Select **LabRole** from the drop-down list.

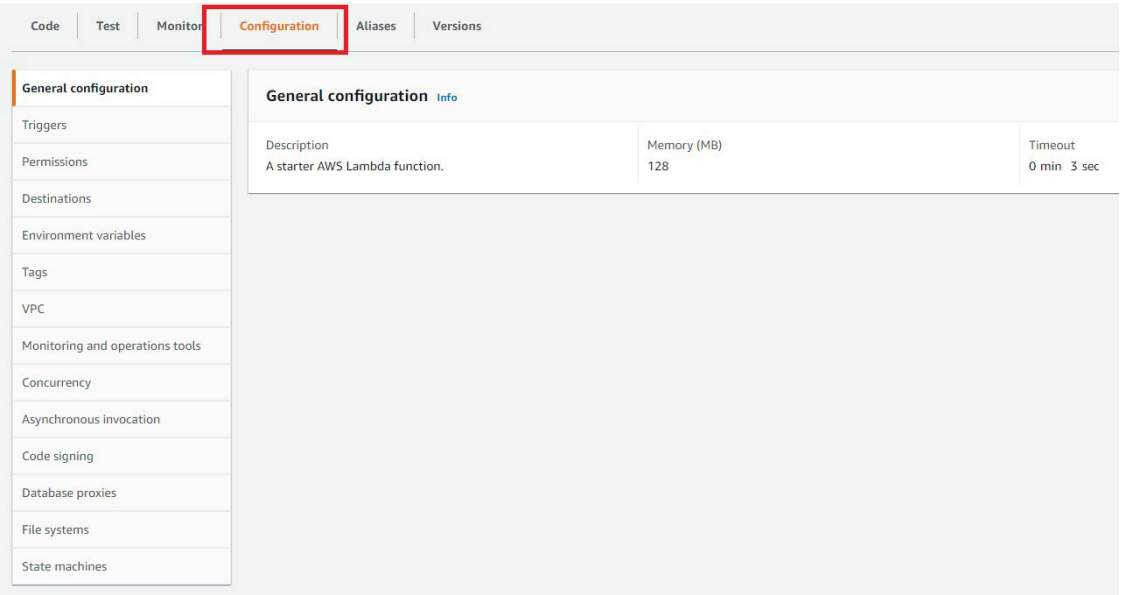
- Lambda Function Code: In this section, you can review the example code authored in Python. We will leave the existing implementation as is.
- Handler: You can specify a handler (a method/function in your code) where AWS Lambda can begin executing your code. AWS Lambda provides event data as inputto this handler, which processes the event.

In this example, Lambda identifies this from the code sample, and this should be pre-populated with *lambda_function.lambda_handler*.



b. Go to the bottom of the page and select **Create function**.

c. Click on the *Configure* tab from the AWS Lambda options menu to configure the memory, timeout, and VPC settings, etc of your lambda function. For this tutorial, leave the default Lambda function configuration values as is.



Step 4: Invoke Lambda Function and Verify Results.

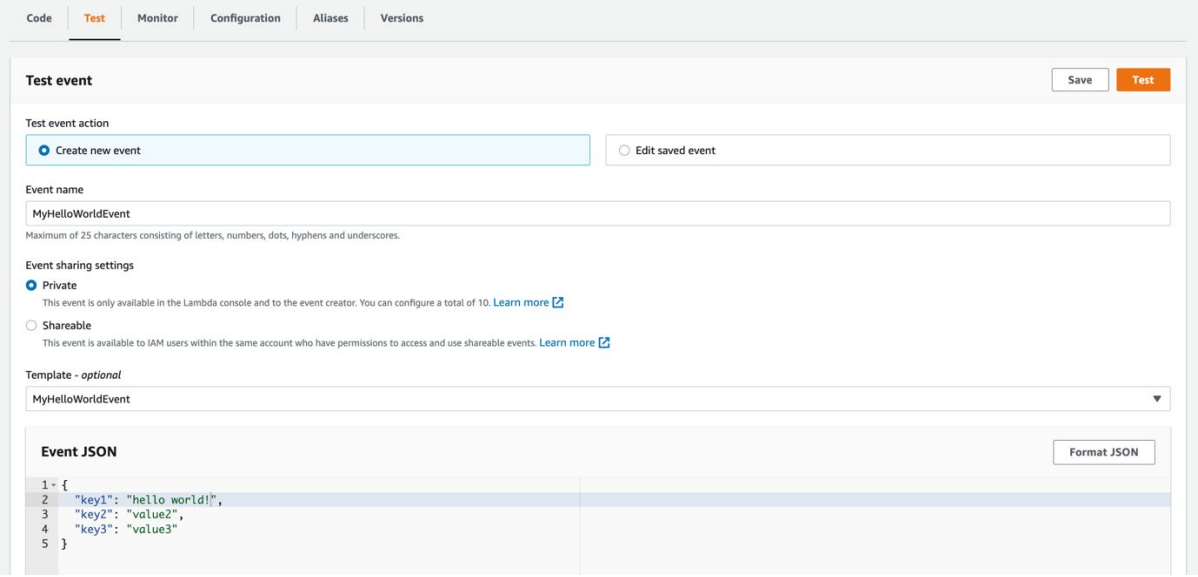
Now, the AWS Lambda console shows the **hello-world-python** Lambda function - you can now test the function, verify results, and review the logs.

a. Select *Test* tab from the AWS Lambda options menu and select **Create new event** as the **Test event action** to create a new test event.

b. When you are in the test event creation wizard,

- Type in an **Event name** like *HelloWorldEvent.*
- Leave event sharing settings as **Private**, which is its default value.
- Choose **hello-world** from the sample event **Template** list.
- You can change the values in the sample JSON, but don't change the event structure. For this tutorial, replace *value1* with the phrase *hello, world!*.

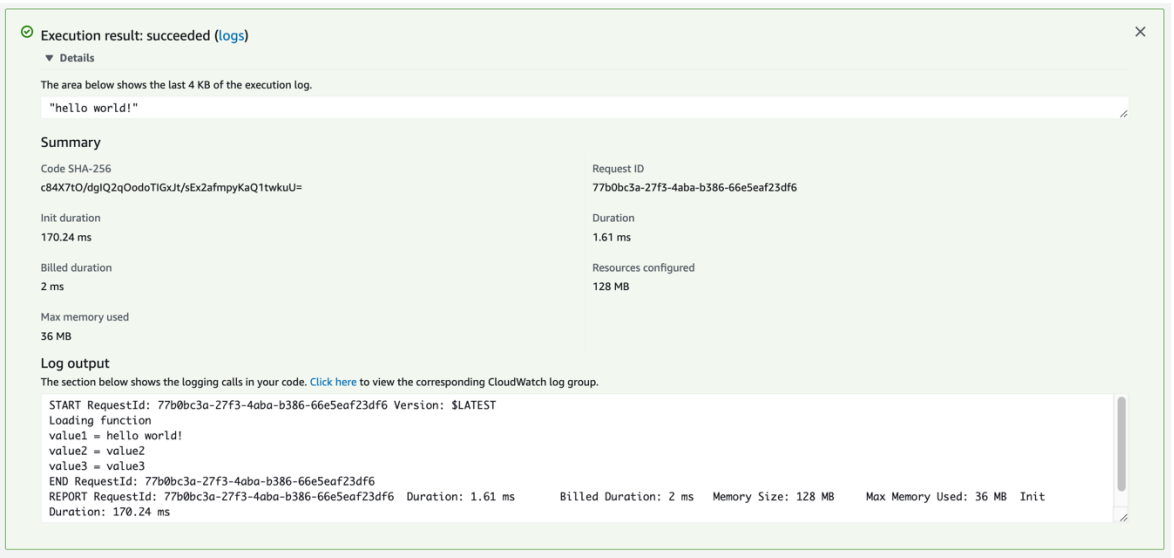
Select *Save changes*.



c. Click *Test*.

d. Upon successful execution, view the results in the console:

- The Execution results section verifies that the execution succeeded.
- The Summary section shows the key information reported in the Log output.
- The Log output section will show the logs generated by the Lambda function execution.



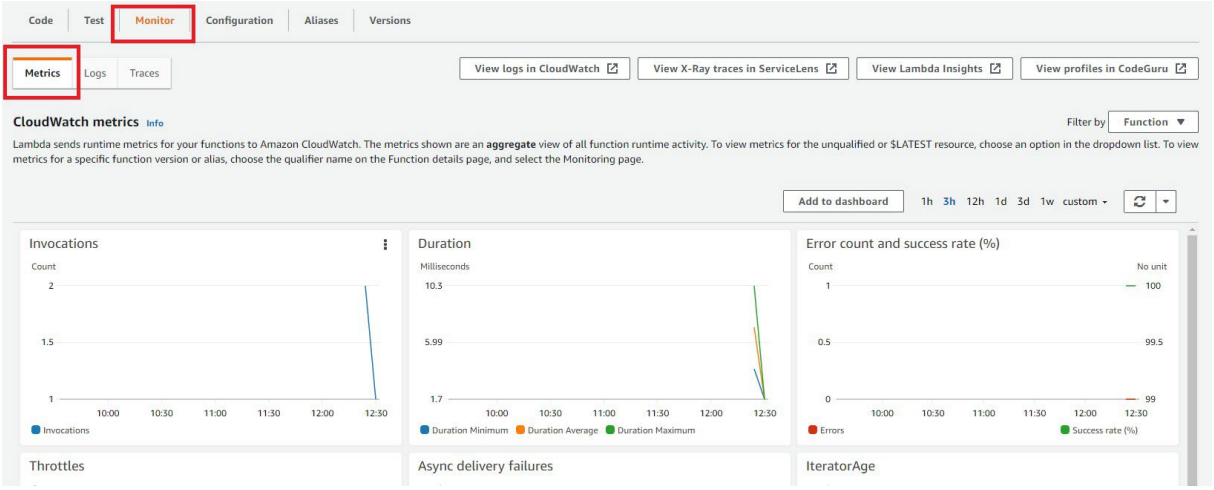
Step 5: Monitor Your Metrics

AWS Lambda automatically monitors Lambda functions and reports metrics through Amazon CloudWatch. To help you monitor your code as it executes, Lambda automatically tracks the number of requests, the latency per request, and the number of requests resulting in an error and publishes the associated metrics.

a. Invoke the Lambda function a few more times by repeatedly clicking the **Test** button. This will generate the metrics that can be viewed in the next step.

b. Select *Monitor* tab from the AWS Lambda options menu to view the results.

c. Click the *Metrics* option to view metrics of your Lambda function. Lambda metrics are reported through Amazon CloudWatch. You can leverage these metrics to set custom alarms. For more information about CloudWatch, see the Amazon CloudWatch Developer Guide.



The *Metrics* tab will show seven CloudWatch metrics: *Invocations, Duration, Error count and success rate (%), Throttles, Async delivery failures, IteratorAge and Concurrent executions.*

With AWS Lambda, you pay for what you use. After you hit your AWS Lambda free tier limit, you are charged based on the number of requests for your functions (invocation count) and the time your code executes (invocation duration). For more information, see AWS Lambda Pricing.

Step 6: Delete the Lambda Function

While you will not get charged for keeping your Lambda function, you can easily delete it from the AWS Lambda console.

a. Select the **Actions** button and click **Delete**.

b. You will be asked to confirm the deletion of your Lambda function - select **Delete**.

Task 2: Create a Lambda function in Python 3.8 using AWS CLI

Prerequisites

This section describes the tools and resources required to complete the steps in the tutorial.

1. Install the AWS CLI

The AWS CLI is an open-source tool that enables you to interact with AWS services using commands in your command line shell. To complete the steps in this section, you must have the following:

- AWS CLI – Install version 2

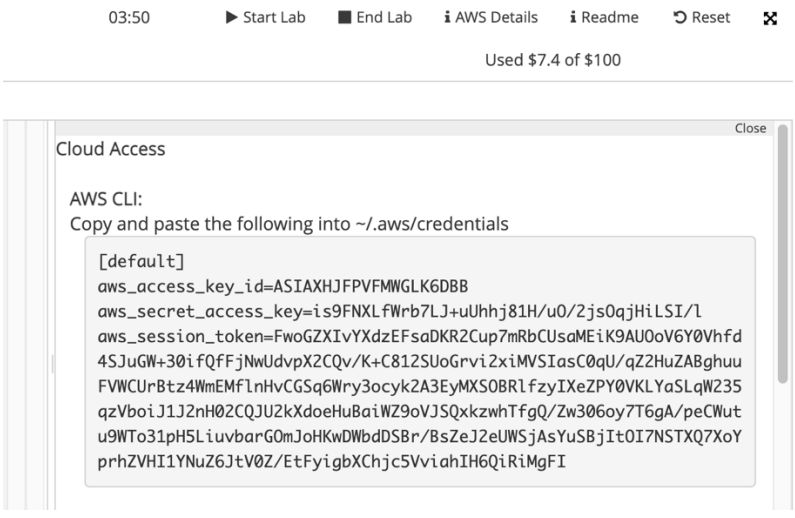
In addition, you also need to make sure that the AWS access credentials are properly configured in your localcomputer/development environment using the steps below.

To configure the access credentials,

a. Log in to your AWS Academy account, start a lab session if you do not have one running already.

b. Click *AWS Details* menu item from the “Learner Lab – Associate services” section, and copy the access credentials generated (I.e. aws_access_key_id, aws_secrete_key, aws_session_token) for your AWS Academy lab session.

Note: Make sure the credentials you are using are not expired and are associated with your current lab session in AWS Academy portal.



c. Save the credentials into the *~/.aws/credentials* file that can be located in the logged in user's home directory on **MacOS/Linux** systems. On **Windows**, you will find this file in *%userprofile%\.aws\credentials.* If the file named *credentials* does not exist, create it.

If you are using your personal AWS accounts,

- Run AWS CLI – Quick configuration with *aws configure* command, as below.

```
C02ZD019LVDF-AmaniSoysa:bqlab3 amani.soysa$ aws configure
AWS Access Key ID [****************AUSN]: xxxxxxxxxx
AWS Secret Access Key [****************1p8n]: xxxxxxxxxx
Default region name [us-east-1]:
Default output format [None]:
C02ZD019LVDF-AmaniSoysa:bqlab3 amani.soysa$
```

*** Run the following step ONLY if you are using your personal AWS accounts ***

2. Create an execution role

Your Lambda function's execution role is an AWS Identity and Access Management (IAM) role that grants your function permission to access AWS services and resources. Create an execution role in IAM with Invoke permission.

To create the execution role

1. Create a *policy.json* file with the following content within your preferred location in thelocal file-system.

```
{"Version": "2012-10-17","Statement": [{"Effect": "Allow", "Principal":
{"Service": "lambda.amazonaws.com"}, "Action": "sts:AssumeRole"}]}
```

2. While you are in the same directory, open a command prompt and use the create-role command to create an execution role named `LabRole`.

```
aws iam create-role --role-name LabRole --assume-role-policy-document
file://policy.json
```

This command produces the following output. Save the value returned in `Arn`.

```
{
        "Role": {
            "Path": "/",
            "RoleName": "LabRole",
            "RoleId": "AROAWNZPPVHULXRJXQJD5",
            "Arn": "arn:aws:iam::your-account-id:role/LabRole",
            "CreateDate": "2021-01-05T18:00:30Z",
            "AssumeRolePolicyDocument": {
                "Version": "2012-10-17",
                "Statement": [
                    {
                        "Effect": "Allow",
                        "Principal": {
                            "Service": "lambda.amazonaws.com"
                        },
                        "Action": "sts:AssumeRole"
                    }
                ]
            }
        }
    }
```

Copy `arn:aws:iam::your-account-id:role/LabRole` into a .txt file to use later.

3. Use the attach-role-policy command to add `AWSLambdaBasicExecutionRole` permissions to the role.

```
$ aws iam attach-role-policy --role-name LabRole --policy-arn
arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

This command produces no output.

3. Creating a function without runtime dependencies

A dependency can be any package, module or other assembly dependency that is not included with the Lambda runtime environment for your function's code. For more information, see What is a runtime dependency?.

For more information, see What is a runtime dependency?

This section describes how to create a Lambda function without runtime dependencies.

Overview

In this tutorial, you will use a sample Lambda function implementation available in AWS SDK examples Github repository, to create a Lambda function using the AWS CLI. You will learn how to:

- Set up the directory structure of a deployment package (.zip file).
- Create a deployment package for a Lambda function without any runtime dependencies.
- Use the AWS CLI to upload the deployment package and create the Lambda function.
- Invoke the Lambda function to return a mathematical calculation.

The sample code contains standard math and logging Python libraries, which are used to return a calculation based on user input. Standard Python libraries are included with the `python3.8` runtime. Although the function's code doesn't depend on any other Python libraries and has no additional application dependencies, AWS Lambda still requires a deployment package to create and deploy a Lambda function.

Create the deployment package

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives.

Create the .zip file that Lambda uses as your deployment package.

To create the deployment package

1. Open a command prompt and create a `my-math-function` project directory. For example, on macOS:

```
mkdir my-math-function
```

2. Navigate to the `my-math-function` project directory.

```
cd my-math-function
```

3. Copy the contents of this sample Lambda function implementation and save it in a new filenamed `lambda_function.py`. Your directory structure should look like this.

```
my-math-function$
| lambda_function.py
```

4. Create a `my-deployment-package.zip` file adding the `lambda_function.py` tothe root of the zip file.

   This generates a `my-deployment-package.zip` file in your project directory.

Create the Lambda function

Lambda needs to know the runtime environment to use for your function's code, the handler in your function code, and the execution role that it can use to invoke your function.

Create the Lambda function using the execution role and deployment package that you created in the previous steps.

To create the function

1. Navigate to the `my-math-function` project directory.

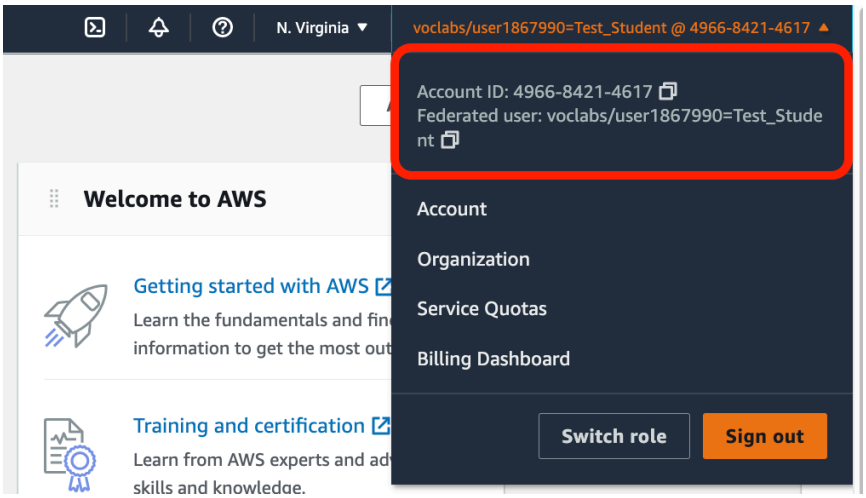
```
cd my-math-function
```

2. Create a function named `my-math-function`.

If you are using your own AWS account for this exercise, substitute the value for `--role` with the `Arn` you copied in Step 2.

If you are using an AWS Academy account, follow the steps below to generate the `Arn` of the **LabRole**, which is to be used with the `--role` directive.

a. Find the **Account ID** associated with your AWS Academy account by clicking on the user info drop-down box to the top right hand side corner of the AWS Management Console, as shown below.



b. Now, substitute this Account ID for the your-account-id in the role ARN **arn:aws:iam::your-account-id:role/LabRole**

E.g. **arn:aws:iam::496684214617/LabRole**

Now, run the following command.

```
aws lambda create-function --function-name my-math-function --zip-file
fileb://my-deployment-package.zip --handler lambda_function.lambda_handler
--runtime python3.8 --role arn:aws:iam::your-account-id:role/LabRole
```

This command produces an output similar to what is shown below.

```
{
    "FunctionName": "my-math-function",
    "FunctionArn": "arn:aws:lambda:us-east-1:496684214617:function:my-math-function",
    "Runtime": "python3.8",
    "Role": "arn:aws:iam::496684214617:role/LabRole",
    "Handler": "lambda_function.lambda_handler",
    "CodeSize": 795,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2022-03-29T19:49:04.632+0000",
    "CodeSha256": "xbLyNtTM63CN0RUSMoB8UwJHJuFw86Tq/K4d4w1Kp/0=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "e0959a8d-b1ed-41c1-b094-5923faf11faa",
    "State": "Pending",
    "StateReason": "The function is being created.",
    "StateReasonCode": "Creating",
    "PackageType": "Zip"
}
```


The Lambda function in this step uses a function handler named `lambda_function.lambda_handler`. For more information about function handler naming conventions, see Naming in **AWS Lambda function handler in Python**.

Invoke the Lambda function

Invoke the Lambda function synchronously using the event input defined for the sample code. For more information, see How it works in **AWS Lambda function handler in Python**.

To invoke the function

- Create *input.json* file containing the following content.

```
{"action": "square","number": 3}
```

- Use the invoke command.

Note: If you are using an AWS CLI version prior to v2, remove the following directive from your command.
`--cli-binary-format raw-in-base64-out`

```
aws lambda invoke --function-name my-math-function --payload
file://input.json output.txt --cli-binary-format raw-in-base64-out
```

This command produces the following output.

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

For the `RequestResponse` invocation type, the status code is `200`. For more information, see the Invoke API reference.

You should see the following mathematical calculation in `output.txt`:

```
{"result": 9}
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them for further experimentation. By deleting AWS resources that you are no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda function

- Use the delete-function command.

```
aws lambda delete-function --function-name my-math-function
```

This command produces no output.

Creating a function with runtime dependencies

A dependency can be any package, module or other assembly dependency that is not included with the Lambda runtime environment for your function's code. For more information, see what is a runtime dependency?.

For more information, see what is a runtime dependency?

This section describes how to create a Lambda function with runtime dependencies.

Overview

In this tutorial, you use sample code to create a Lambda function using the AWS CLI. The sample code uses the **requests** python package/library to get the HTML code associated with the web page https://docs.aws.amazon.com/. The aforementioned **requests** package/library is one developed by a third-party and therefore, not included with the core `python3.8` runtime. So, you need to install it and package it into your deployment package.

In this exercise, you will learn to

- Set up the directory structure of a deployment package (.zip file) with runtime dependencies.
- Create a deployment package for a Lambda function with runtime dependencies.
- Use the AWS CLI to upload the deployment package and create the Lambda function.
- Invoke the Lambda function to return the source code.

Create the deployment package

Your AWS Lambda function's code consists of scripts or compiled programs and their dependencies. You use a *deployment package* to deploy your function code to Lambda. Lambda supports two types of deployment packages: container images and .zip file archives. In this exercise, we will use the latter.

Create the .zip file that Lambda uses as your deployment package.

To create the deployment package

1. Open a command prompt and create a `my-advanced-function` project directory. Forexample, on macOS:

```
mkdir my-advanced-function
```

2. Navigate to the `my-advanced-function` project directory.

```
cd my-advanced-function
```

3. Copy the contents of the following sample Python code and save it in a new filenamed `lambda_function.py`:

```
import requests
def lambda_handler(event, context):
    response = requests.get("https://docs.aws.amazon.com")
    print(response.text)
    return response.text
```

Your directory structure should look like this:

```
my-advanced-function
| lambda_function.py
```

4. Install the requests library to a new `package` directory, which resides in the `my-advanced-function` directory.

```
pip install --target ./package requests
```

5. Create a deployment package `my-deployment-package.zip` with the `lambda_function.py` and all the installed libraries at the root within the zip file.

This generates a `my-deployment-package.zip` file in your project directory.

Note: Do not bundle the package directory. You only need to copy the directories inside the package directory into the root directory within the zip file.

Create the Lambda function

Lambda needs to know the runtime environment to use for your function's code, the handler in your function code, and the execution role that it can use to invoke your function.

Create the Lambda function using the execution role and deployment package that you created in the previous steps.

To create the function

1. Navigate to the `my-advanced-function` project directory.

```
cd my-advanced-function
```

2. Create a function named `my-advanced-function`. Substitute the value for `role` with the `Arn` you copied in previous steps.

```
aws lambda create-function --function-name my-advanced-function --zip-
file fileb://my-deployment-package.zip --handler
lambda_function.lambda_handler --runtime python3.8 --role
arn:aws:iam::your-account-id:role/LabRole
```

This command produces an output similar to what is shown below.

```
{
    "FunctionName": "my-advanced-function",
    "FunctionArn": "arn:aws:lambda:us-east-1:496684214617:function:my-advanced-function",
    "Runtime": "python3.8",
    "Role": "arn:aws:iam::496684214617:role/LabRole",
    "Handler": "lambda_function.lambda_handler",
    "CodeSize": 788251,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2022-03-29T20:22:25.225+0000",
    "CodeSha256": "OB1mpS6aQFEGDsO98G11Bi65JNqpd3L0iNY6Z/DTlWE=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "e9298995-8d59-42c3-9273-61f648fefafc",
    "State": "Pending",
    "StateReason": "The function is being created.",
    "StateReasonCode": "Creating",
    "PackageType": "Zip"
}
```


The Lambda function in this step uses a function handler of `lambda_function.main`. For more information about function handler naming conventions, see Naming in **AWS Lambda function handler in Python**.

Invoke the Lambda function

Invoke the Lambda function synchronously using the event input defined for the sample code. For more information, see How it works in **AWS Lambda function handler in Python**.

To invoke the function

- Create *input.json* file with the following content

```
{"dummyKey": "dummyValue"}
```

- Use the invoke command.

```
aws lambda invoke --function-name my-advanced-function --cli-binary-format
raw-in-base64-out --payload file://input.json output.txt
```

Note: If you are using an AWS CLI version prior to v2, remove the following directive from thecommand:

```
--cli-binary-format raw-in-base64-out
```

This command produces the following output:

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

For the `RequestResponse` invocation type, the status code is `200`. For more information, see the Invoke API reference.

You should see the HTML code of the web page `https://docs.aws.amazon.com` in `output.txt`.

Updating a Lambda function in Python 3.8

To update a Python function

1. Add updated function code files to the root of your deployment package.
2. Use the fileb:// prefix to upload the binary .zip file to Lambda and update the function code.

```
~/my-function$ aws lambda update-function-code --function-name my-
advanced-function --zip-file fileb://my-deployment-package.zip
```

Clean up your resources

You can now delete the resources that you created for this tutorial, unless you want to retain them. By deleting AWS resources that you're no longer using, you prevent unnecessary charges to your AWS account.

To delete the Lambda functions

- Use the delete-function command.

```
aws lambda delete-function --function-name my-advanced-function
```

This command produces no output.

Note: Also delete the execution role (if you used your personal AWS account for this exercise).

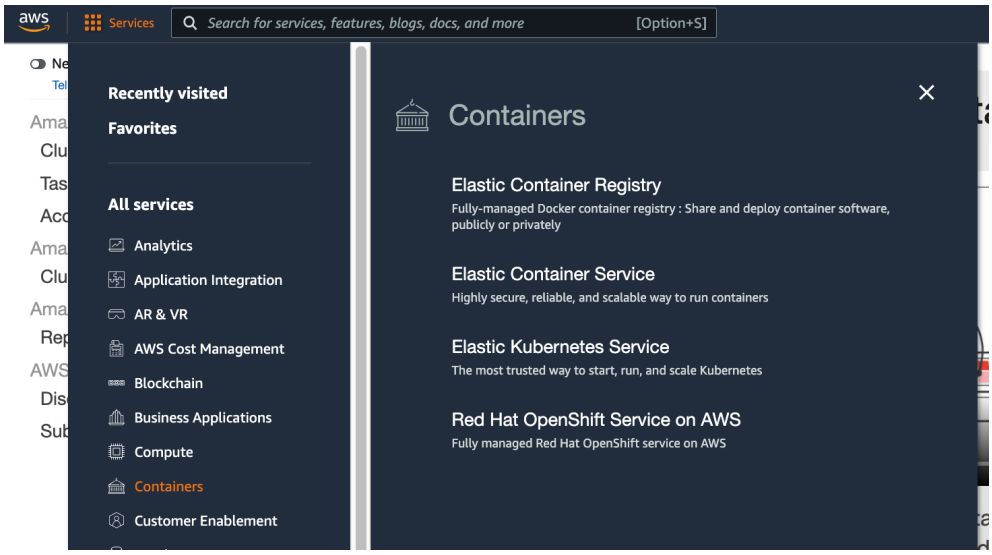
Task 3: Deploy Docker Containers on Amazon Elastic Container Service (Amazon ECS)

Amazon Elastic Container Service (Amazon ECS) is the Amazon Web Service you use to run Docker applications on a scalable cluster of compute resources. In this tutorial, you will learn how to run a Docker-enabled sample application on an Amazon ECS cluster behind a load balancer, test the sample application, and delete your resources to avoid charges.

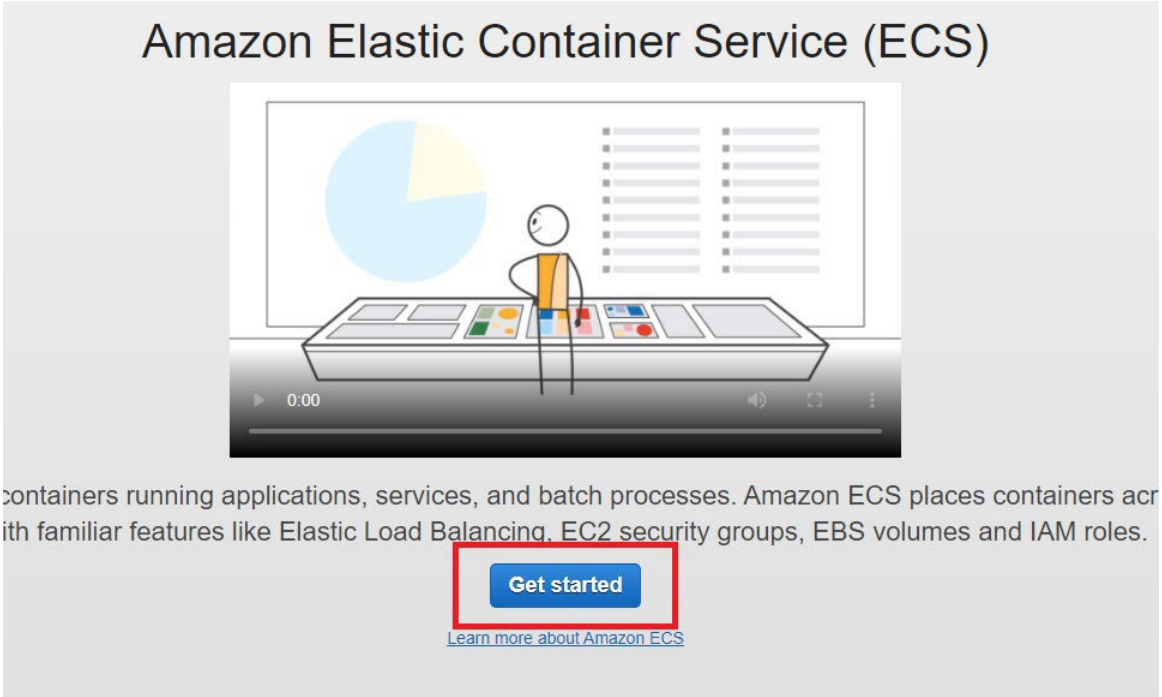
Everything done in this tutorial is free tier eligible.

Step 1: Set up your first ECS cluster and run it

Select AWS Elastic Container Service from the list of services available in your AWS Management Console under the **Containers** category.



The Amazon ECS *Get started* wizard will guide you through the process of creating a cluster and launching a sample web application. In this step, you will enter the Amazon ECS console and launch the wizard by clicking on the *Get started* button.



Step 2: Create a container and task definitions

A *container definition* is used in task definitions to describe the different containers that are launched as part of a task.

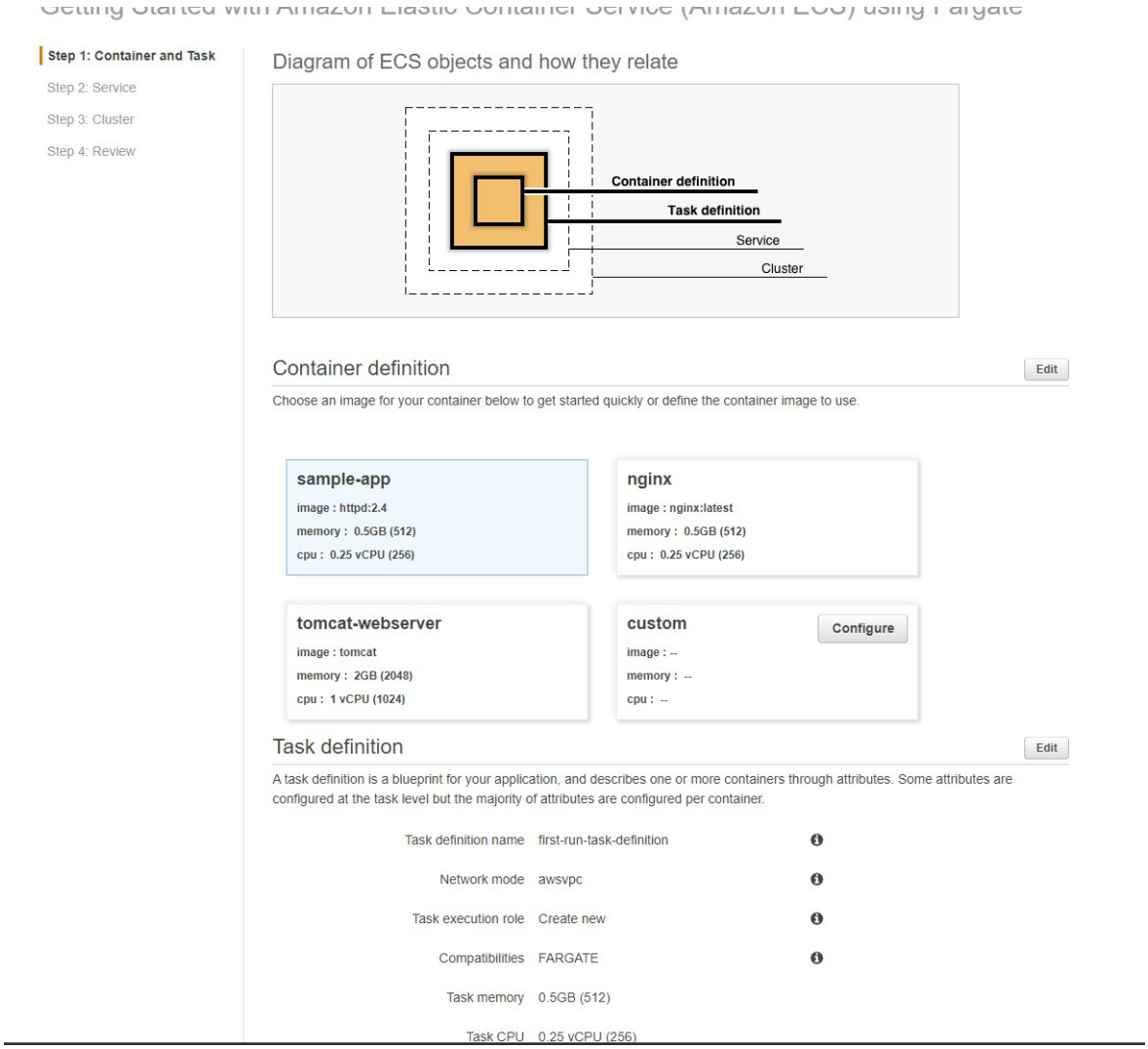
A *task definition*, meanwhile, is like a blueprint for your application. In this step, you will select a container definition and task definition, so Amazon ECS knows which Docker image to use for containers, how many containers to use in the task, and the resource allocation for each container.

The task definition comes pre-loaded with default configuration values.

- Review the default values and select *Next* Step.

If you prefer to modify the configurations or would like to learn more, see Task Definition Parameters.

Important note: If you are using an AWS Academy account for this exercise, remember to choose **LabRole** as your **Task execution role**.

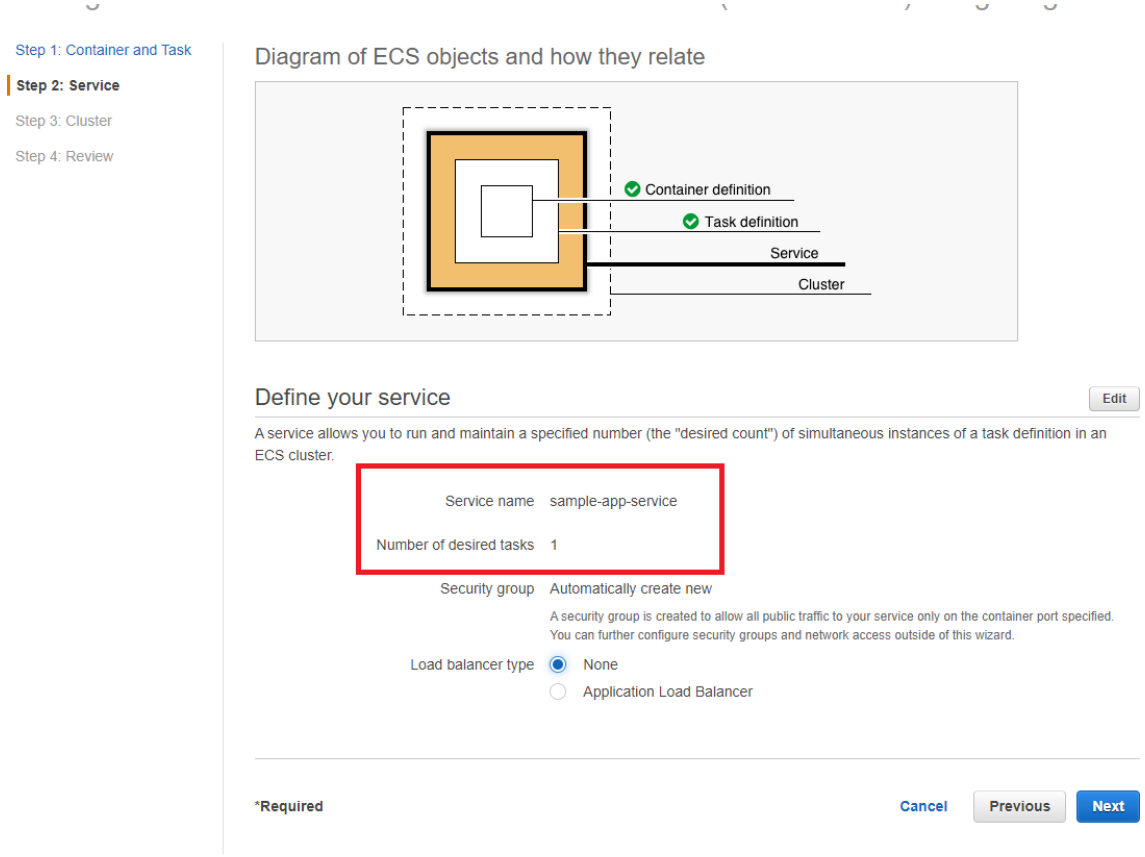


Step 3: Configure your service

Now that you have created a container and task definition, you will configure the Amazon ECS *service*. A service launches and maintains copies of the task definition in your cluster. For example, by running an application as a service, Amazon ECS will auto-recover any stopped tasks and maintain the number of copies you specify.

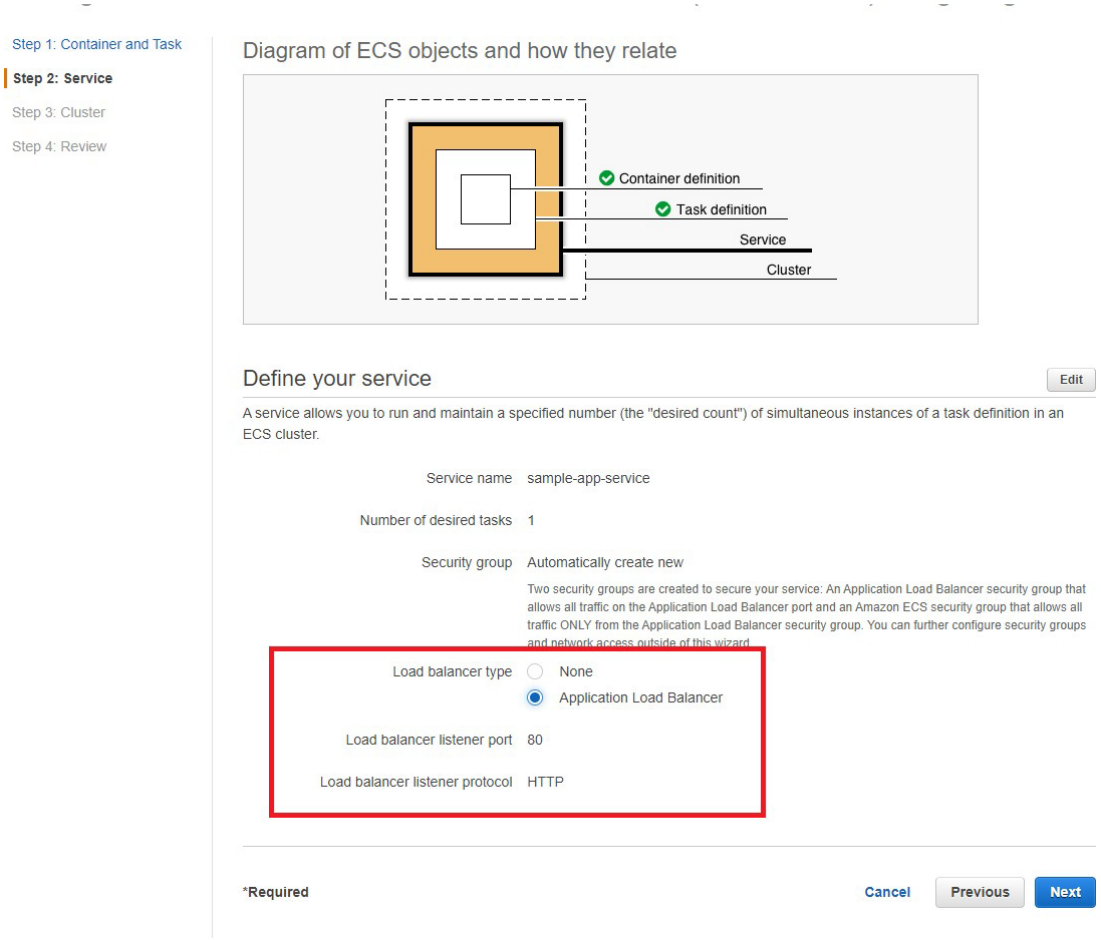
a. Configure service options:

- Service Name: The default *sample-app-service* is a web-based "Hello World" application provided by AWS. It is meant to run indefinitely, so by running it as a service, it will restart if the task becomes unhealthy or unexpectedly stops.
- Desired number of tasks: To stay within the AWS free tier, leave the default value of *1*. This will create 1 copy of your task.



b. Elastic load balancing: You have the option to use a load balancer with your service. Amazon ECS can create an Application Load Balancer (ALB) to distribute the traffic across the container instances your task is launched on.

- The default values for A*LB listener protocol* and *listener port* are set up for the sample application. For more information on load balancing configuration, see Service Load Balancing.



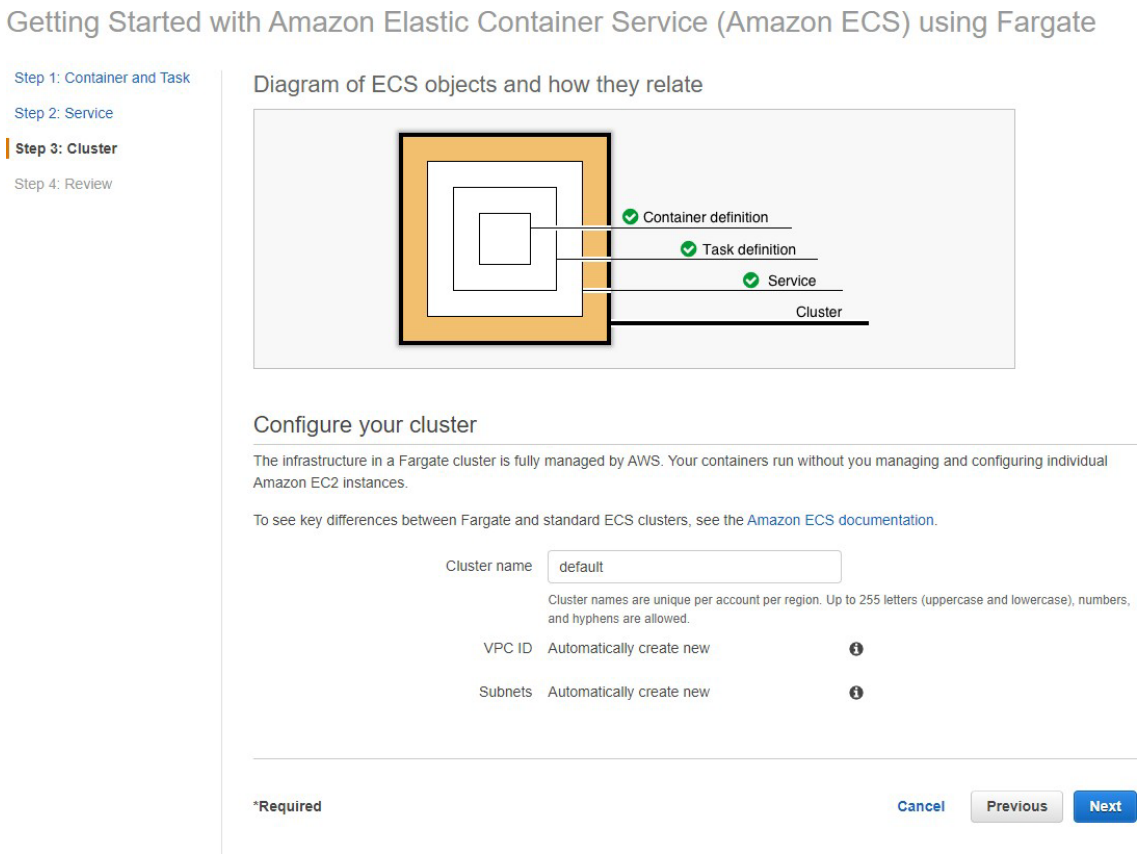
d. Review your settings and select Next Step.

Step 4: Configure your cluster

Your Amazon ECS tasks run on a *cluster*, which is the set of container instances (I.e EC2 instances) running the Amazon ECS container agent. In this step, we create a *Fargate cluster*, which is a fully managed service offerred by AWS. In *Fargate*, your containers run without you managing and configuring individual Amazon EC2 instances.

- Leave the cluster name as *default*.

- Click *Next*.

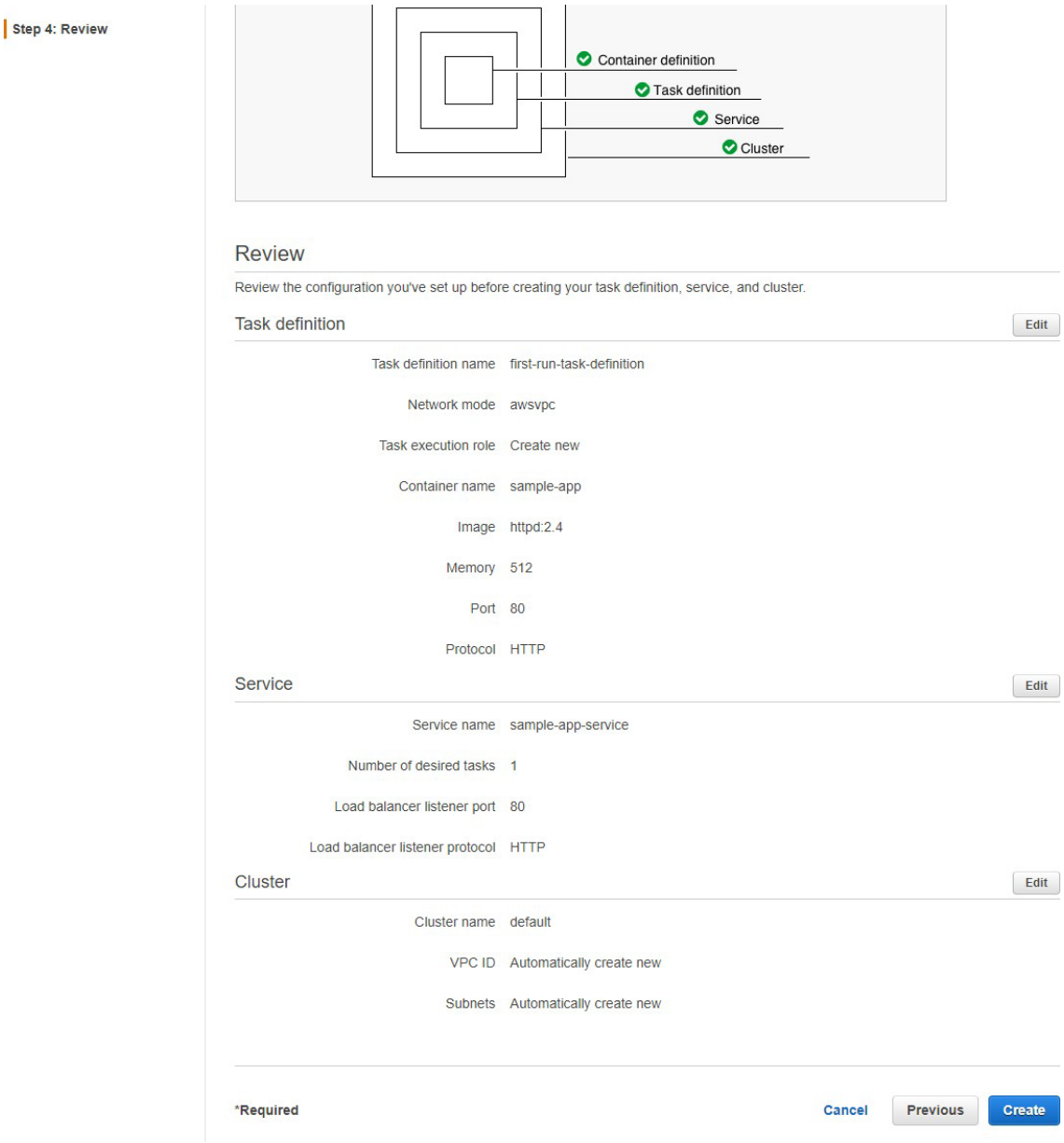


In previous steps, you have configured your container and task definition (which is like an application blueprint), the Amazon ECS service (which launches and maintains copies of your task definitions), and a Fargate cluster (which is the set of container instances running the container agent). In this step, you will review, launch, and view the resources you create.

a. You have a final chance to review your task definition, task configuration, and cluster configurations before launching.

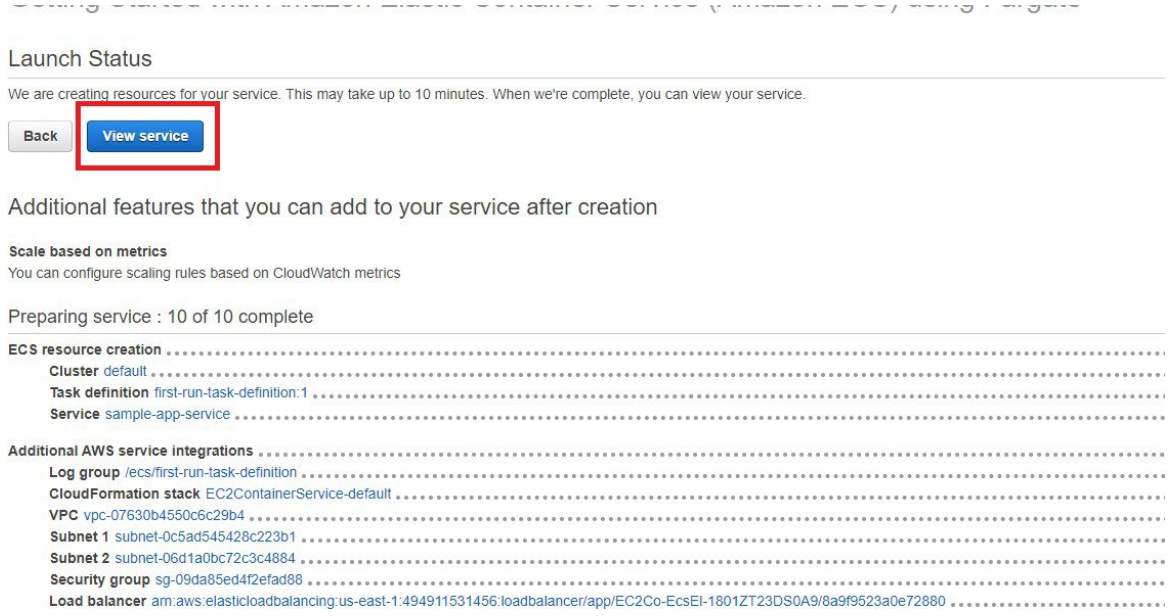
- Select *Create* to launch your resources.

  This may take a few minutes.



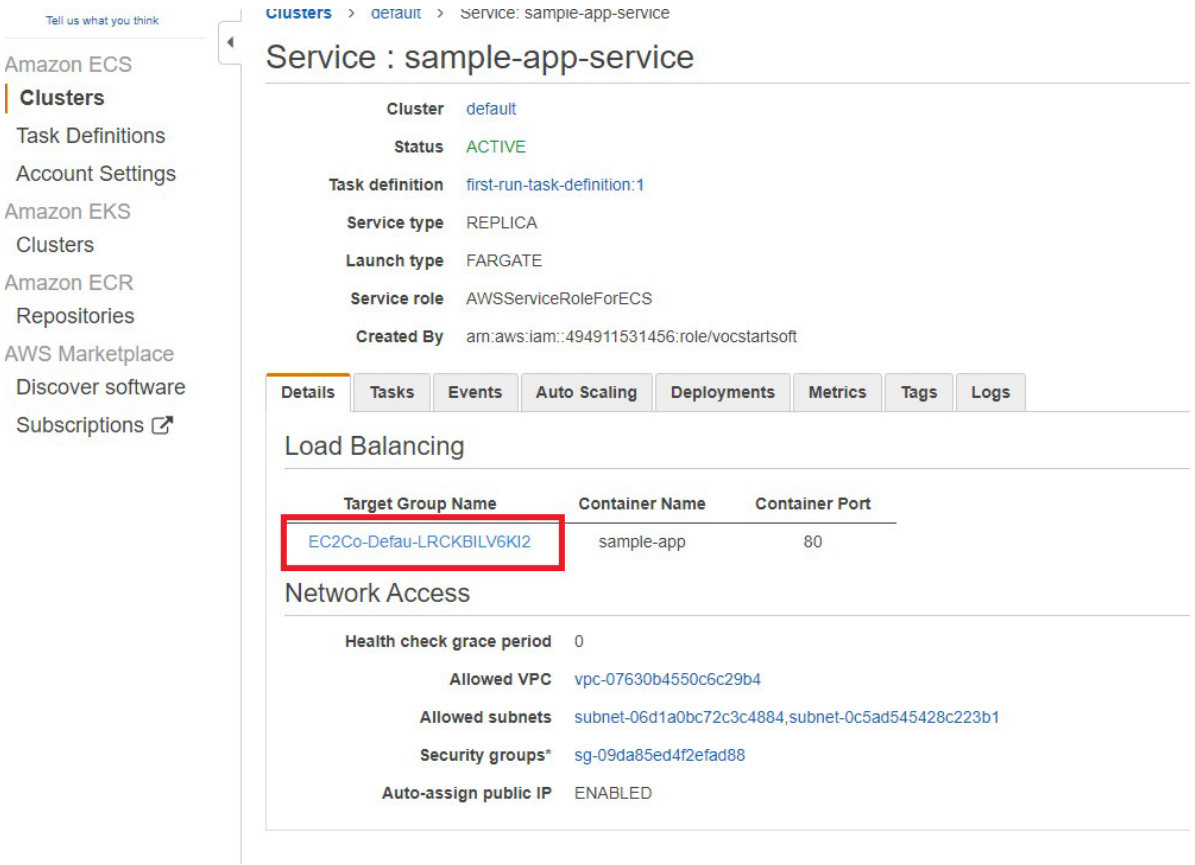
- After the launch is complete, select *View service*.

  Note: While your ECS cluster is getting provisioned, examine the types of resources it creates for your automatically.

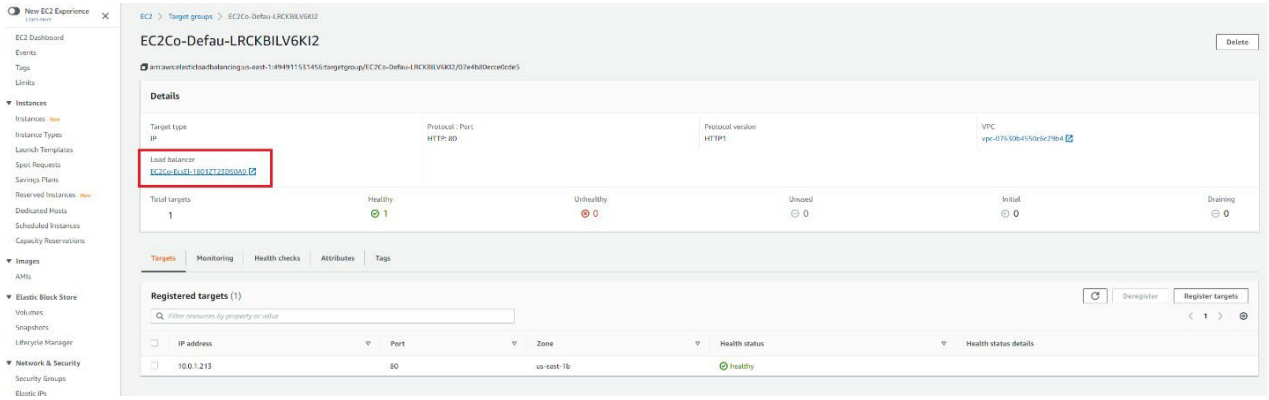


Step 6: Open the Sample Application

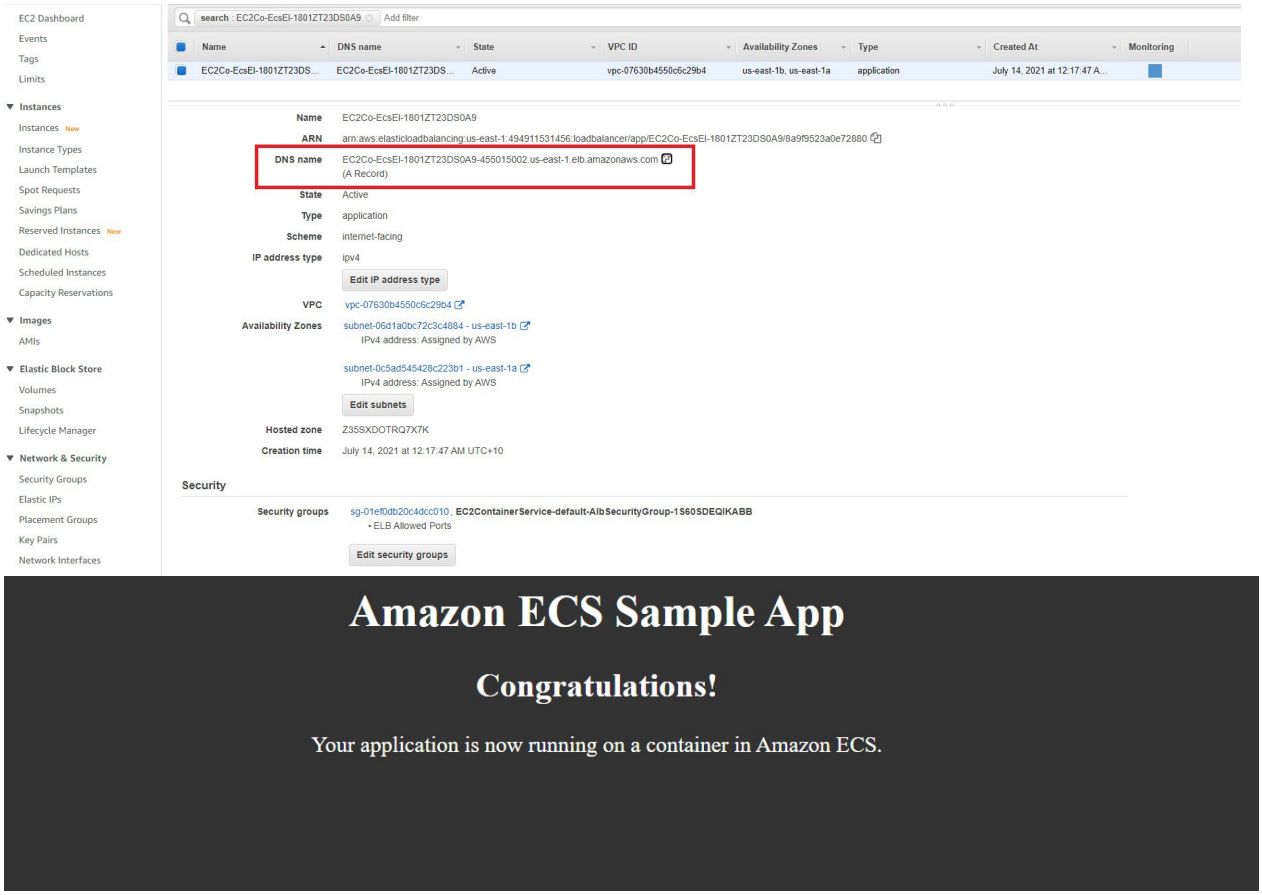
In this step, you will verify that the sample application is up and running by pointing your browser to the load balancer DNS name.



- From the *Target groups* page you were redirected to in the previous step, click the *Load balancer* link.

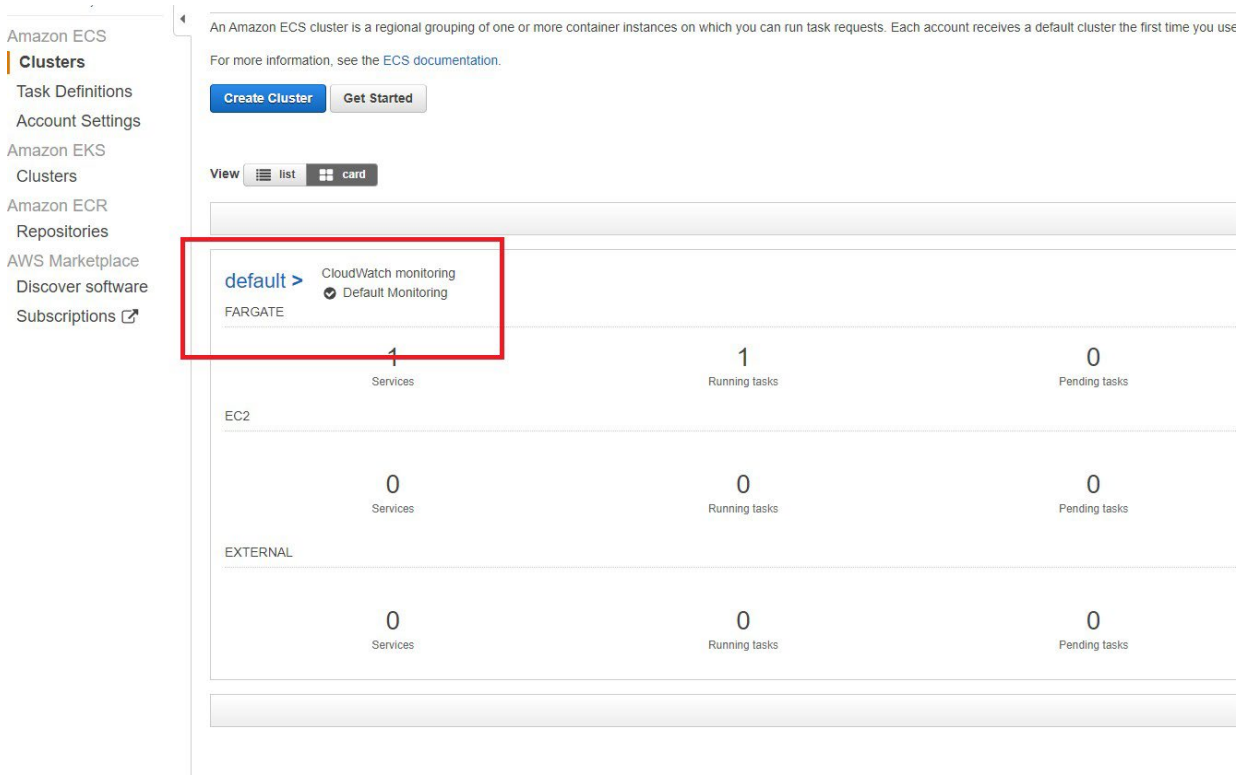


- From the load balancer's information section, copy the *DNS name* associated with it.
- Paste it into a new browser window.
- You will see the *Amazon ECS Sample App* being rendered into the browser window.

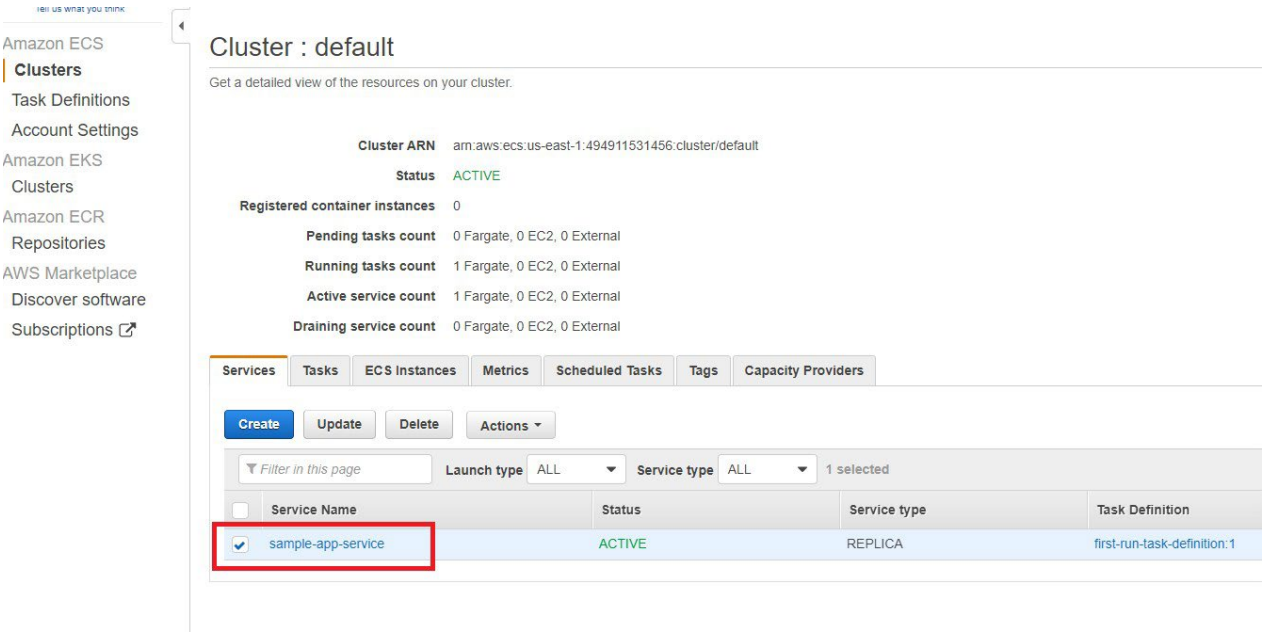


Step 7: Delete Your Resources

- Click on the cluster name (e.g. *default*).

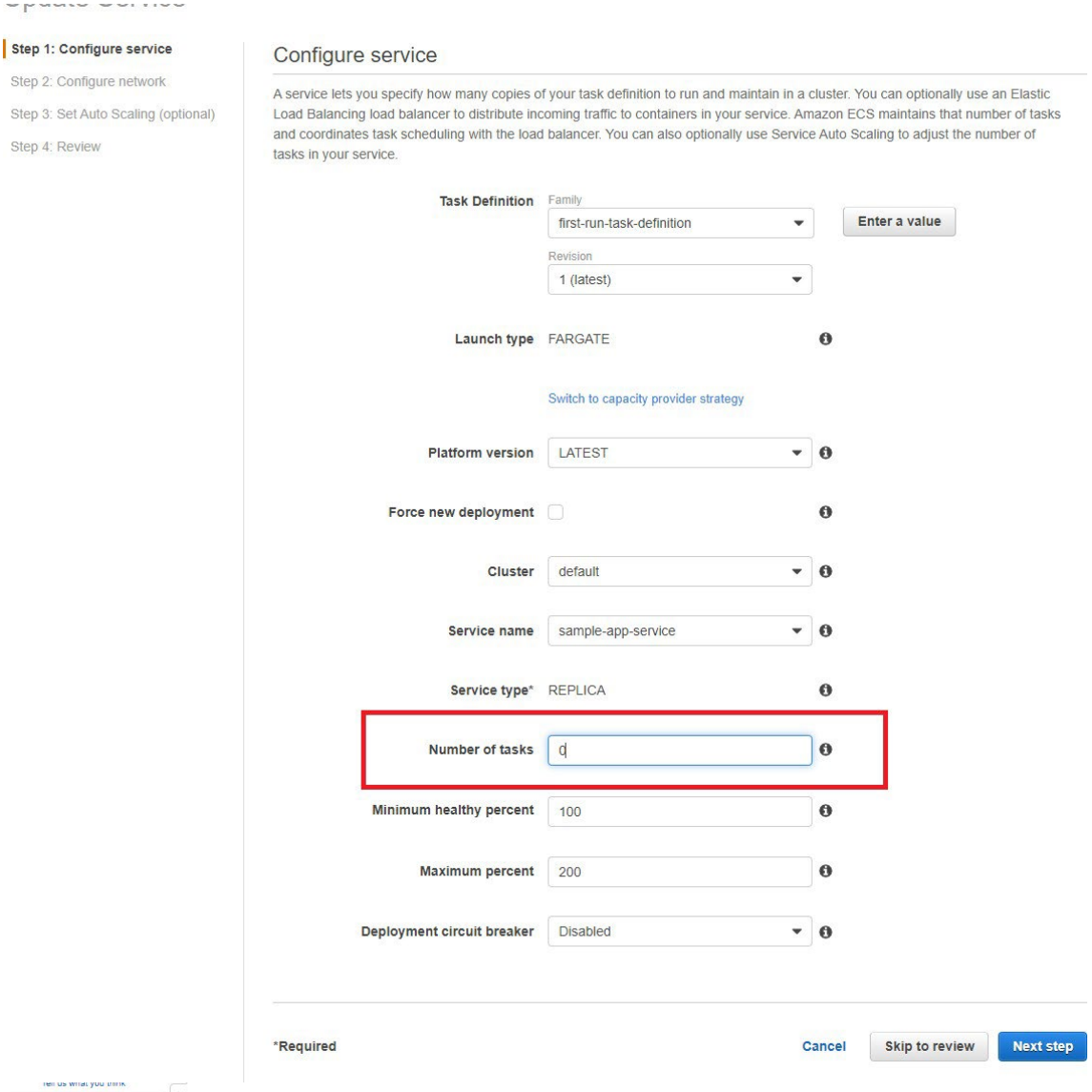


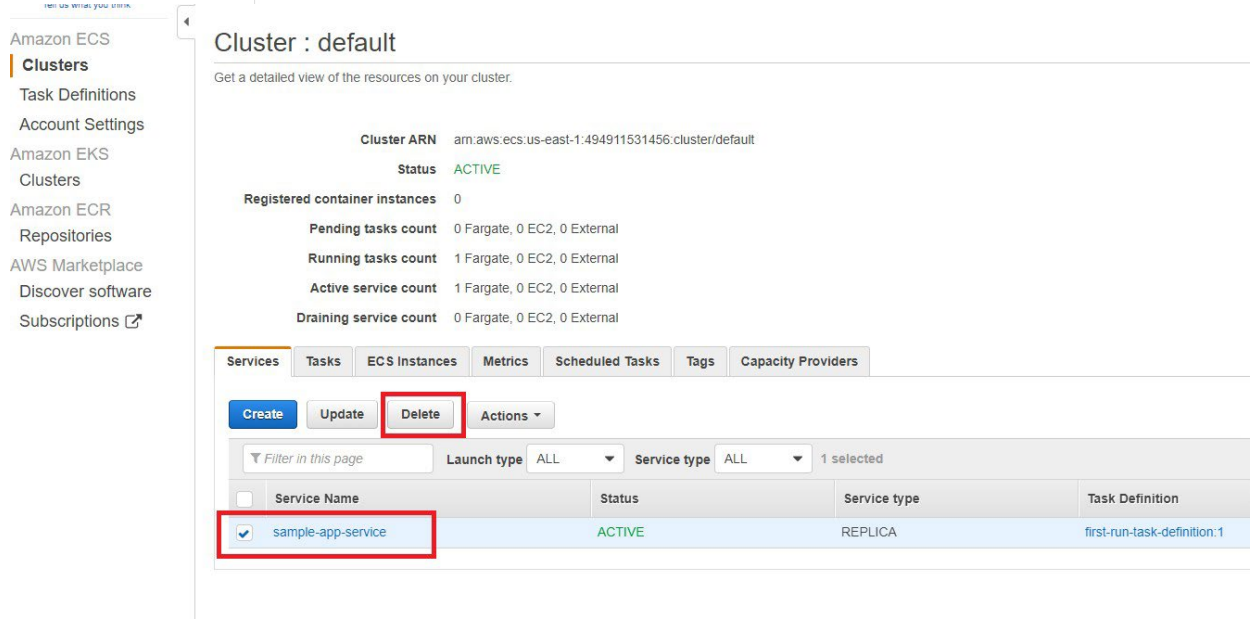
b. Select the checkbox next to *sample-app-service* and click *Update*.



c. To ensure you don't accidentally delete a service with active tasks, you need to stop all tasks before Amazon ECS will delete a service.

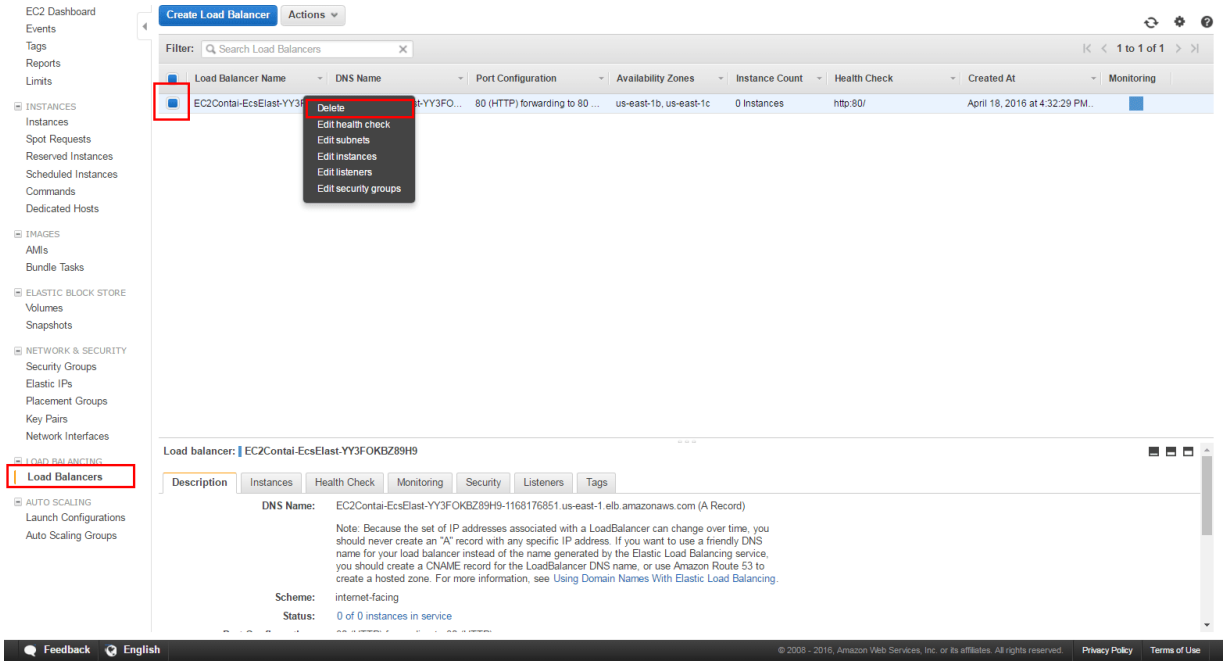
- After you update your service, select *Delete*.


Step 1: Configure service
Step 2: Configure network
Step 3: Set Auto Scaling (optional)
Step 4: Review
Configure service
A service lets you specify how many copies of your task definition to run and maintain in a cluster. You can optionally use an Elastic Load Balancing load balancer to distribute incoming traffic to containers in your service. Amazon ECS maintains that number of tasks and coordinates task scheduling with the load balancer. You can also optionally use Service Auto Scaling to adjust the number of tasks in your service.
Task Definition
Family
first-run-task-definition
Enter a value
Revision
1 (latest)
Launch type
FARGATE
Switch to capacity provider strategy
Platform version
LATEST
Force new deployment
Cluster
default
Service name
sample-app-service
Service type*
REPLICA
Number of tasks
0
Minimum healthy percent
100
Maximum percent
200
Deployment circuit breaker
Disabled
*Required
Cancel
Skip to review
Next step


Amazon ECS
Clusters
Task Definitions
Account Settings
Amazon EKS
Clusters
Amazon ECR
Repositories
AWS Marketplace
Discover software
Subscriptions
Cluster : default
Get a detailed view of the resources on your cluster.
Cluster ARN
arn:aws:ecs:us-east-1:494911531456:cluster/default
Status
ACTIVE
Registered container instances
0
Pending tasks count
0 Fargate, 0 EC2, 0 External
Running tasks count
1 Fargate, 0 EC2, 0 External
Active service count
1 Fargate, 0 EC2, 0 External
Draining service count
0 Fargate, 0 EC2, 0 External
Services
Tasks
ECS Instances
Metrics
Scheduled Tasks
Tags
Capacity Providers
Create
Update
Delete
Actions
Filter in this page
Launch type
ALL
Service type
ALL
1 selected
Service Name
Status
Service type
Task Definition
sample-app-service
ACTIVE
REPLICA
first-run-task-definition:1

d. Delete *Load balancer* created for your appication:

- Enter the Amazon EC2 console
- In the left hand panel, select *Load balancers*.
- Select the checkbox next to the created Load balancer (e.g. EC2Co-EcsEl-1801ZT23DS0A9).
- Select Actions > Delete.



Congratulations!

Congratulations! You have learned how to configure, deploy, and delete your Docker-enabled application to Amazon Elastic Container Service (Amazon ECS). Amazon ECS is a highly scalable, high performance container management service that supports Docker containers and allows you to easily run applications on a managed cluster of Amazon EC2 instances.

Additional Documents:

- Updating a Lambda function in Python 3.8. https://docs.aws.amazon.com/lambda/latest/dg/python-package-update.html
- Docker Desktop overview. https://docs.docker.com/desktop/
- A Docker Tutorial for Beginners. https://docker-curriculum.com/

Note: Terminate all your AWS resources after this lab to avoid extra costs.