### What is AWS Lambda?

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). Users of AWS Lambda create functions, self-contained applications written in one of the supported languages and runtimes, and upload them to AWS Lambda, which executes those functions in an efficient and flexible manner.  
  
The Lambda functions can perform any kind of computing task, from serving web pages and processing streams of data to calling APIs and integrating with other AWS services.  
  
The concept of “serverless” computing refers to not needing to maintain your own servers to run these functions. AWS Lambda is a fully managed service that takes care of all the infrastructure for you. And so “serverless” doesn’t mean that there are no servers involved: it just means that the servers, the operating systems, the network layer and the rest of the infrastructure have already been taken care of, so that you can focus on writing application code.

### How does AWS Lambda work?

Each Lambda function runs in its own container. When a function is created, Lambda packages it into a new container and then executes that container on a multi-tenant cluster of machines managed by AWS. Before the functions start running, each function’s container is allocated its necessary RAM and CPU capacity. Once the functions finish running, the RAM allocated at the beginning is multiplied by the amount of time the function spent running. The customers then get charged based on the allocated memory and the amount of run time the function took to complete.  
  
The entire infrastructure layer of AWS Lambda is managed by AWS. Customers don’t get much visibility into how the system operates, but they also don’t need to worry about updating the underlying machines, avoiding network contention, and so on—AWS takes care of this itself.  
  
And since the service is fully managed, using AWS Lambda can save you time on operational tasks. When there is no infrastructure to maintain, you can spend more time working on the application code—even though this also means you give up the flexibility of operating your own infrastructure.  
  
One of the distinctive architectural properties of AWS Lambda is that many instances of the same function, or of different functions from the same AWS account, can be executed concurrently. Moreover, the concurrency can vary according to the time of day or the day of the week, and such variation makes no difference to Lambda—you only get charged for the compute your functions use. This makes AWS Lambda a good fit for deploying highly scalable cloud computing solutions.

### Why is AWS Lambda an essential part of the Serverless architecture?

When building Serverless applications, AWS Lambda is one of the main candidates for running the application code. Typically, to complete a Serverless stack you’ll need:

* a computing service;
* a database service; and
* an HTTP gateway service.

Lambda fills the primary role of the compute service on AWS. It also integrates with many other AWS services and, together with API Gateway, DynamoDB and RDS, forms the basis for Serverless solutions for those using AWS. Lambda supports many of the most popular languages and runtimes, so it’s a good fit for a wide range of Serverless developers.

### What are the most common use cases for AWS Lambda?

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

* individual tasks run for a short time;
* each task is generally self-contained;
* there is a large difference between the lowest and highest levels in the workload of the application.

Some of the most common use cases for AWS Lambda that fit these criteria are:  
Scalable APIs. When building APIs using AWS Lambda, one execution of a Lambda function can serve a single HTTP request. Different parts of the API can be routed to different Lambda functions via Amazon API Gateway. AWS Lambda automatically scales individual functions according to the demand for them, so different parts of your API can scale differently according to current usage levels. This allows for cost-effective and flexible API setups.  
‍  
Data processing. Lambda functions are optimized for event-based data processing. It is easy to integrate AWS Lambda with datasources like Amazon DynamoDB and trigger a Lambda function for specific kinds of data events. For example, you could employ Lambda to do some work every time an item in DynamoDB is created or updated, thus making it a good fit for things like notifications, counters and analytics.

### Task automation

With its event-driven model and flexibility, AWS Lambda is a great fit for automating various business tasks that don’t require an entire server at all times. This might include running scheduled jobs that perform cleanup in your infrastructure, processing data from forms submitted on your website, or moving data around between different datastores on demand.

### Supported languages and runtimes

As of now, AWS Lambda doesn’t support all programming languages, but it does support a number of the most popular languages and runtimes. This is the full list of what’s supported:

* Node.js 8.10
* Node.js 10.x (normally the latest LTS version from the 10.x series)
* Node.js 12.x (normally the latest LTS version from the 12.x series)
* Python 2.7
* Python 3.6
* Python 3.7
* Python 3.8
* Ruby 2.5
* Java 8 - This includes JVM-based languages that can run on Java 8’s JVM — the latest Clojure 1.10 and Scala 2.12 both run on Java 8 so can be used with AWS Lambda
* Java 11
* Go 1.x (latest release)
* C# — .NET Core 1.0
* C# — .NET Core 2.1
* PowerShell Core 6.0

All these runtimes are maintained by AWS and are provided in an Amazon Linux or Amazon Linux 2 environment. For each of the supported languages, AWS provides an SDK that makes it easier for you to write your Lambda functions and integrate them with other AWS services.  
‍  
A few additional runtimes are still in the pre-release stage. These runtimes are being developed as a part of AWS Labs and are not mentioned in the official documentation:

* Rust 1.31
* C++

The C++ runtime also serves as an example for creating custom runtimes for AWS Lambda. See the AWS docs for the details of how to create a custom runtime if your language isn’t supported by default.

### Benefits of using AWS Lambda

AWS Lambda has a few unique advantages over maintaining your own servers in the cloud. The main ones are:  
  
Pay per use. In AWS Lambda, you pay only for the compute your functions use, plus any network traffic generated. For workloads that scale significantly according to time of day, this type of billing is generally more cost-effective.  
‍  
Fully managed infrastructure. Now that your functions run on the managed AWS infrastructure, you don’t need to think about the underlying servers—AWS takes care of this for you. This can result in significant savings on operational tasks such as upgrading the operating system or managing the network layer.  
‍  
Automatic scaling. AWS Lambda creates the instances of your function as they are requested. There is no pre-scaled pool, no scale levels to worry about, no settings to tune—and at the same time your functions are available whenever the load increases or decreases. You only pay for each function’s run time.  
‍  
Tight integration with other AWS products. AWS Lambda integrates with services like DynamoDB, S3 and API Gateway, allowing you to build functionally complete applications within your Lambda functions.

### Limitations of AWS Lambda

While AWS Lambda has many advantages, there are a few things you should know before using it in production.

#### Cold start time

When a function is started in response to an event, there may be a small amount of latency between the event and when the function runs. If your function hasn’t been used in the last 15 minutes, the latency can be as high as 5-10 seconds, making it hard to rely on Lambda for latency-critical applications. There are ways to work around it, including a method we wrote about in our blog.

#### Function limits

The Lambda functions have a few limits applied to them:  
‍  
Execution time/run time. A Lambda function will time out after running for 15 minutes. There is no way to change this limit. If running your function typically takes more than 15 minutes, AWS Lambda might not be a good solution for your task.  
‍  
Memory available to the function. The options for the amount of RAM available to the Lambda functions range from 128MB to 3,008MB with a 64MB step.  
‍  
Code package size. The zipped Lambda code package should not exceed 50MB in size, and the unzipped version shouldn’t be larger than 250MB.  
‍  
Concurrency. By default, the concurrent execution for all AWS Lambda functions within a single AWS account are limited to 1,000. (You can request a limit increase for this number by contacting AWS support.)  
  
Any Lambda executions triggered above your concurrency limit will be throttled and will be forced to wait until other functions finish running.  
‍  
Payload size. When using Amazon API Gateway to trigger Lambda functions in response to HTTP requests (i.e. when building a web application), the maximum payload size that API Gateway can handle is 10MB.

#### Not always cost-effective

On AWS Lambda, you pay only for the used function runtime (plus any associated charges like network traffic). This can produce significant cost savings for certain usage patterns, for example, with cron jobs or other on-demand tasks. However, when the load for your application increases, the AWS Lambda cost increases proportionally and might end up being higher than the cost of similar infrastructure on AWS EC2 or other cloud providers.

#### Limited number of supported runtimes

While AWS Lambda allows adding custom runtimes, creating them can be a lot of work. So if the version of the programming language you are using isn’t supported on Lambda, you might be better off using AWS EC2 or a different cloud provider.

### AWS Lambda pricing

A number of AWS Lambda executions are included with the AWS Free Tier with every AWS account. Unlike some other services, the Lambda free tier isn’t limited to 12 months. Both existing and new accounts get 1 million AWS Lambda requests plus 400,000 GB-seconds per month — Lambda’s measure of function runtime and the memory allocated to a function.  
‍  
Beyond the free tier the pricing for AWS Lambda is as follows:

**Aspect**

**Pricing**

**Comment**

Requests

$0.20 per 1M requests

The free tier includes 1M requests.

Function memory and run time

$0.0000166667 per GB-second

The free tier includes 400,000 GB-seconds.

Inbound network traffic to the Lambda function

Free

Outbound network traffic—within the same AWS region

$0.01/GB

Outbound network traffic—to other AWS regions

$0.02/GB

Outbound network traffic—to public internet

$0.09/GB

Lower pricing per GB applies starting at 10TB/month.

Amazon API Gateway

$3.50 per 1M requests

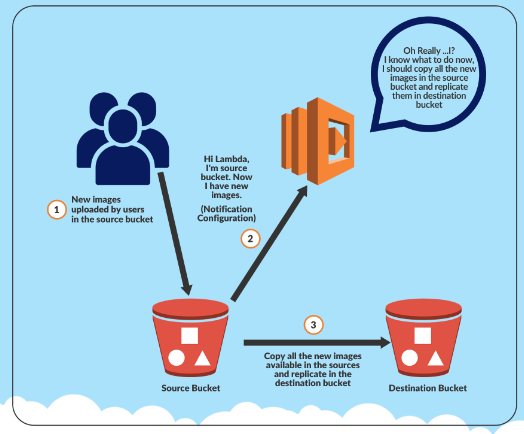
If you are using API Gateway with Lambda functions. Lower pricing from 333M requests per month and up.

For each execution, the total cost will be the sum of all applicable factors, including the cost per request, the memory and run time, and the network traffic.

#### Example cost calculations

Scheduled jobs. Imagine that you run cron jobs in your AWS infrastructure that perform database rollovers. The job runs every night and the average run time is 10 minutes. The function uses 1GB of memory while running. There is no outbound network traffic generated by the function as it connects to your RDS database in the same availability zone.  
‍  
Total compute: 30 days x 600 GB-seconds (10 minutes) = 18,000 GB-seconds Total requests: 30 days x 1 request per day = 30 requests  
‍  
This usage is within the AWS free tier, so you’ll be charged $0 for AWS Lambda. Note: You’ll still pay for the RDS usage according to the RDS price list.  
‍  
HTTP API. Let’s assume that you’re building a web application based entirely on an AWS Lambda backend. Let’s also assume that you’re great at marketing, so after a few months you’ll have 10,000 users in the app every day on average.  
  
Each user’s actions within the app will result in 100 API requests per day, again, on average. Your API runs in Lambda functions that use 512MB of memory, and serving each API request takes 1 second.  
‍  
Total compute: 30 days x 10,000 users x 100 requests x 0.5GB RAM x 1 second = 15,000,000 GB-seconds Total requests: 30 days x 10,000 users x 100 requests = 30,000,000 requests  
‍  
For the 30M requests you’ll pay 30 x $0.20/1M requests = $6/month on AWS Lambda.  
‍  
All these requests go through Amazon API Gateway, so there for the 30M requests you’ll pay 30 x $3.50/1M requests = $105/month on API Gateway.  
‍  
For the monthly 15M GB-seconds of compute on AWS Lambda you’ll pay 15M \* $0.0000166667/GB-second ~= $250/month.  
‍  
So the total cost of the API layer will be around $360/month with this load.

**Architecture Diagram**



# Lab Steps

## Task 1: Sign in to AWS Management Console

1. Click on the **Open Console** button, and you will get redirected to AWS Console in a new browser tab.
2. On the AWS sign-in page,

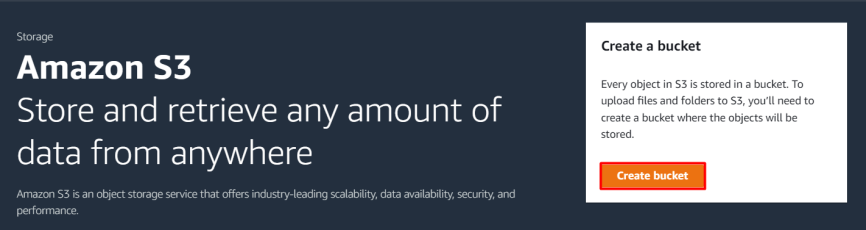
* Leave the Account ID as default. Never edit/remove the 12 digit Account ID present in the AWS Console. otherwise, you cannot proceed with the lab.
* Now copy your **User Name** and **Password** in the Lab Console to the **IAM Username and Password** in AWS Console and click on the **Sign in** button.

1. Once Signed In to the AWS Management Console, Make the default AWS Region as **US East (N. Virginia) us-east-1.**

## Task 2: Create Two Amazon S3 Buckets

In this task, we will create two AWS S3 buckets i.e the source bucket and the destination bucket by providing the required configurations like name, region etc.

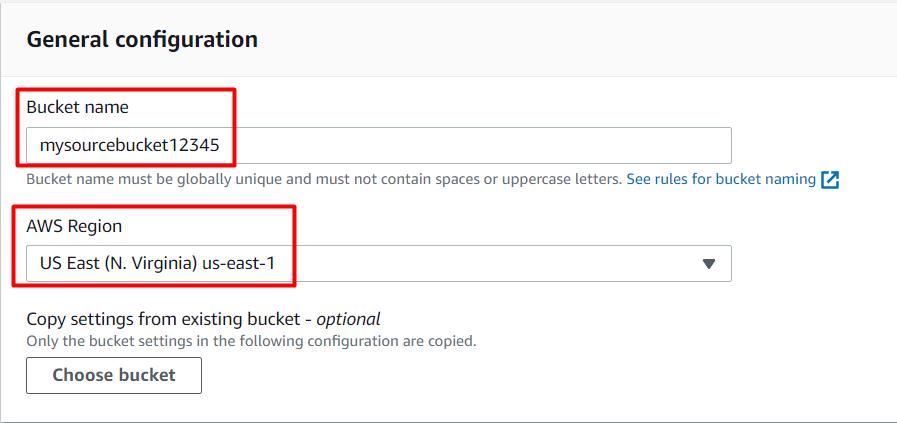
1. Navigate to the **Services**menu in the top, then click on **S3** in the **Storage** section.
2. Click on **Create Bucket**button.



     3. Bucket Name: Enter ***mysourcebucket12345***

* **Note:** Every S3 bucket name is unique globally, so create the bucket with a name not currently in use.

     4. AWS Region: Select **US East (N. Virginia) us-east-1**

****

     5. Leave other settings as default and click on the **Create bucket** button.

     6. Once the bucket is created successfully, **Select** your S3 bucket.

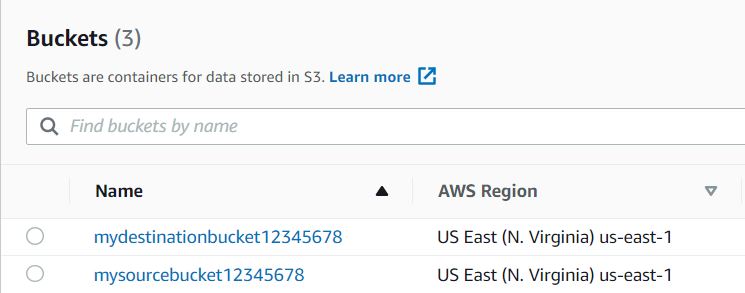
* Click on the **Copy ARN** button to copy the ARN.
* Save the source bucket ARN in a text file for later use.
  + **arn:aws:s3:::mysourcebucket12345**

7. **Create Destination Bucket**

* Click on **Create bucket** button.
* Bucket Name: Enter ***mydestinationbucket12345***
  + **Note:** Every S3 bucket name is unique globally, so create the bucket with a name not currently in use.
* AWS Region: Select **US East (N. Virginia) us-east-1**
* Leave other settings as default and click on the **Create bucket** button.

    8. Once the bucket is created successfully, **Select** your S3 bucket.

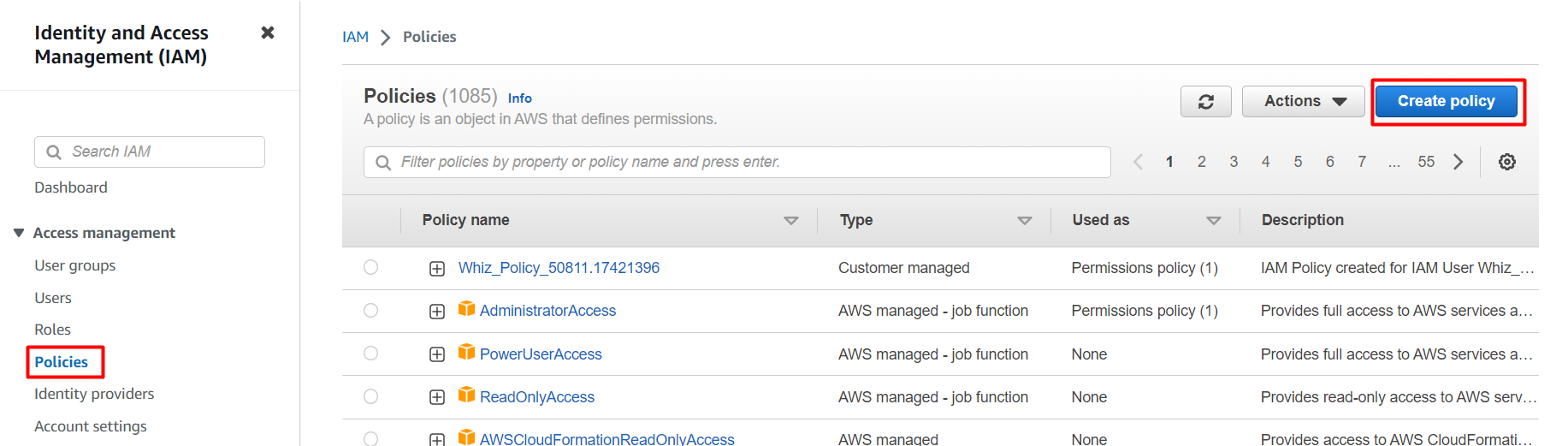
* Click on the **Copy ARN** button to copy the ARN.
* Save the destination bucket ARN in a text file for later use.
  + **arn:aws:s3:::mydestinationbucket12345**



1. Now we have two S3 buckets (Source and Destination). We will make use of our AWS Lambda function to copy the content from source bucket to destination bucket.

## Task 3: Create an IAM Policy

1. As a pre-requisite for creating the Lambda function, we need to create a user role with a custom policy.
2. Go to **Services** and Select **IAM** under **Security, Identity and Compliance.**
3. Click on **Policies**in the left navigation bar and click on the **Create policy**button.

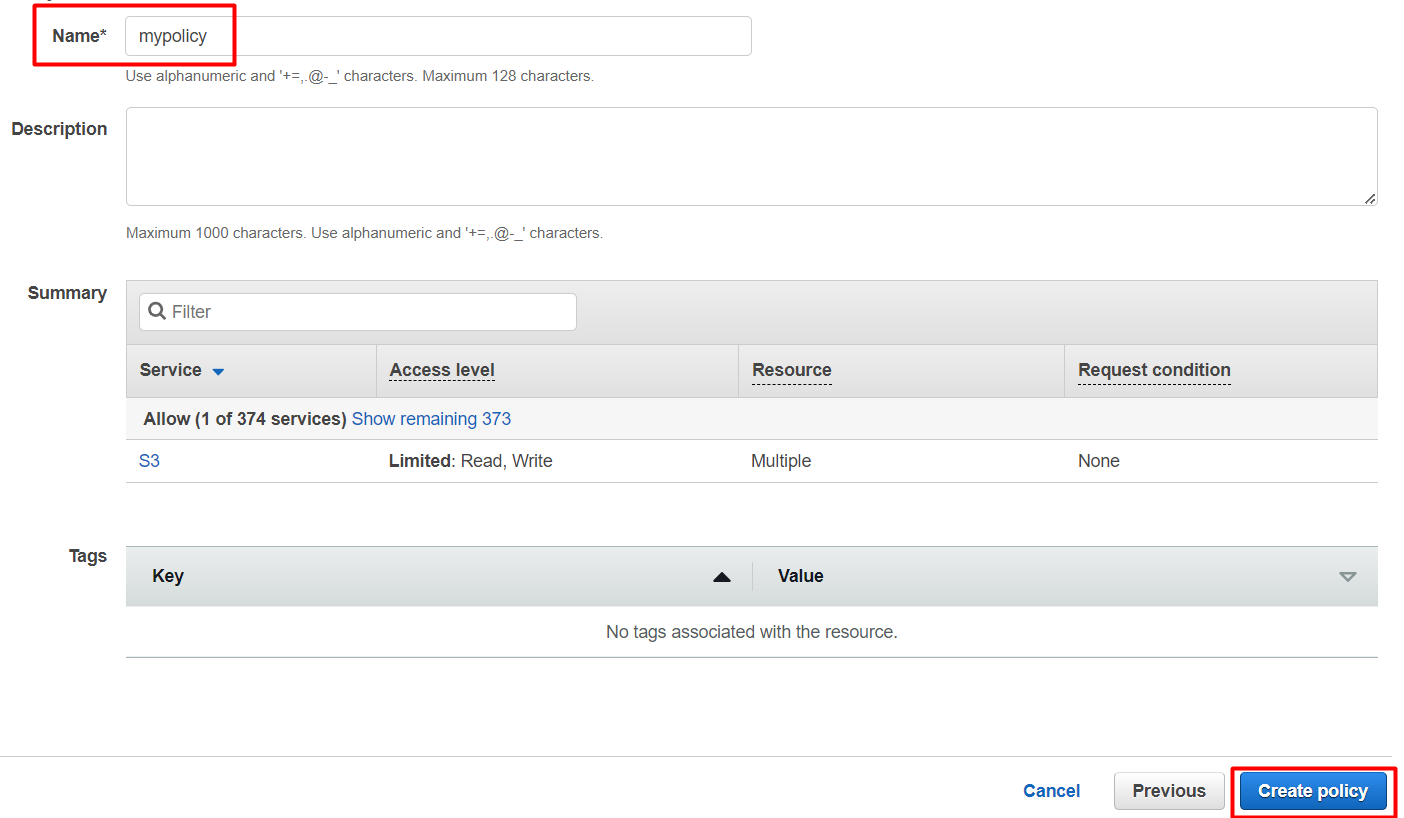


    4. Click on the **JSON** tab, Remove the existing code and copy-paste the below policy statement into the editor:

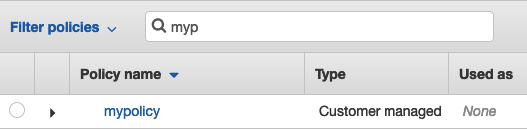
* Policy JSON:

|  |
| --- |
| {      "Version":"2012-10-17",      "Statement":[         {            "Effect":"Allow",            "Action":[               "s3:GetObject"            ],            "Resource":[               "arn:aws:s3:::mysourcebucket12345/\*"            ]         },         {            "Effect":"Allow",            "Action":[               "s3:PutObject"            ],            "Resource":[               "arn:aws:s3:::mydestinationbucket12345/\*"            ]         }      ]   } |

* Replace the **Source**and **destination ARN name**of the bucket (which you have saved before) in the option **Resource**. Make sure to add **/\***at the end of the **ARN name.**
* Leave everything as default and click on **Next**button.
* On the Review Policy Page:
  + Policy Name: Enter **mypolicy**
  + Click on the **Create policy** button.



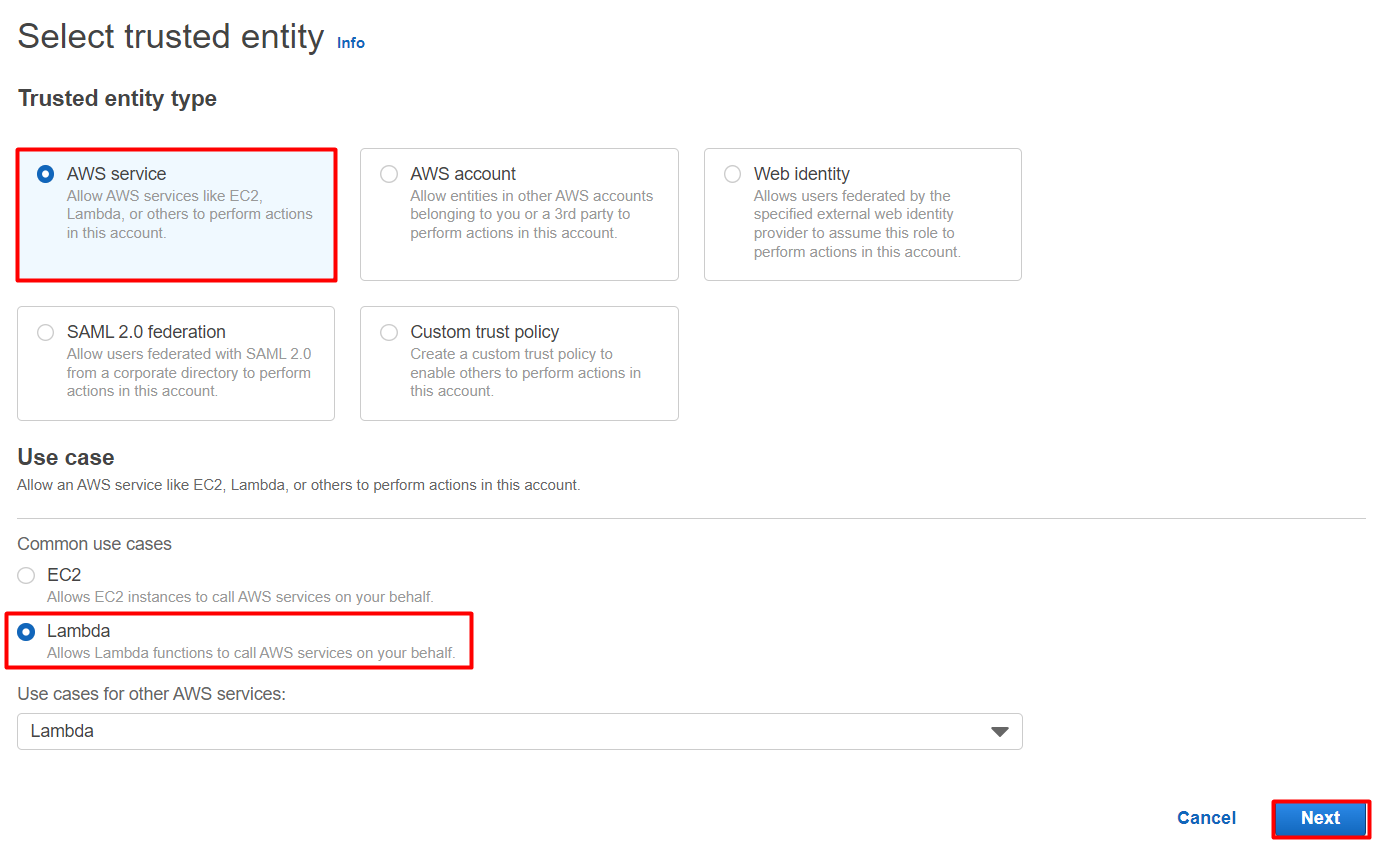
1. An IAM Policy with the name **mypolicy** is created.



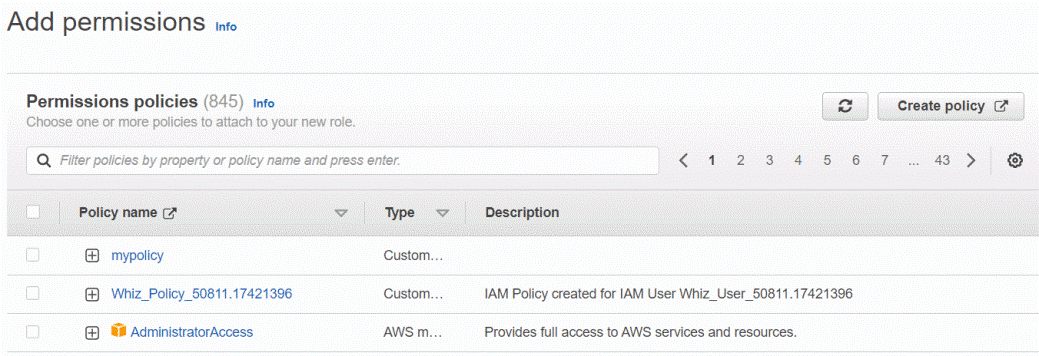
## Task 4: Create an IAM Role

1. In the left menu, click on **Roles.**Click on the **Create role**button.
2. Select **Lambda** from AWS Services list.

* From **Trusted Entity Type**: Select **AWS Service**
* From **Use case**: Select **Lambda**
* Click on **Next** button.

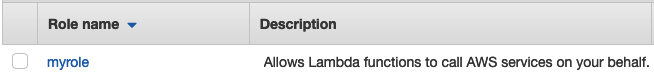


* Filter Policies: Now you can see a list of policies. Here you have to select two policies: **mypolicy** and **AWSLambdaBasicExecutionRole**

****

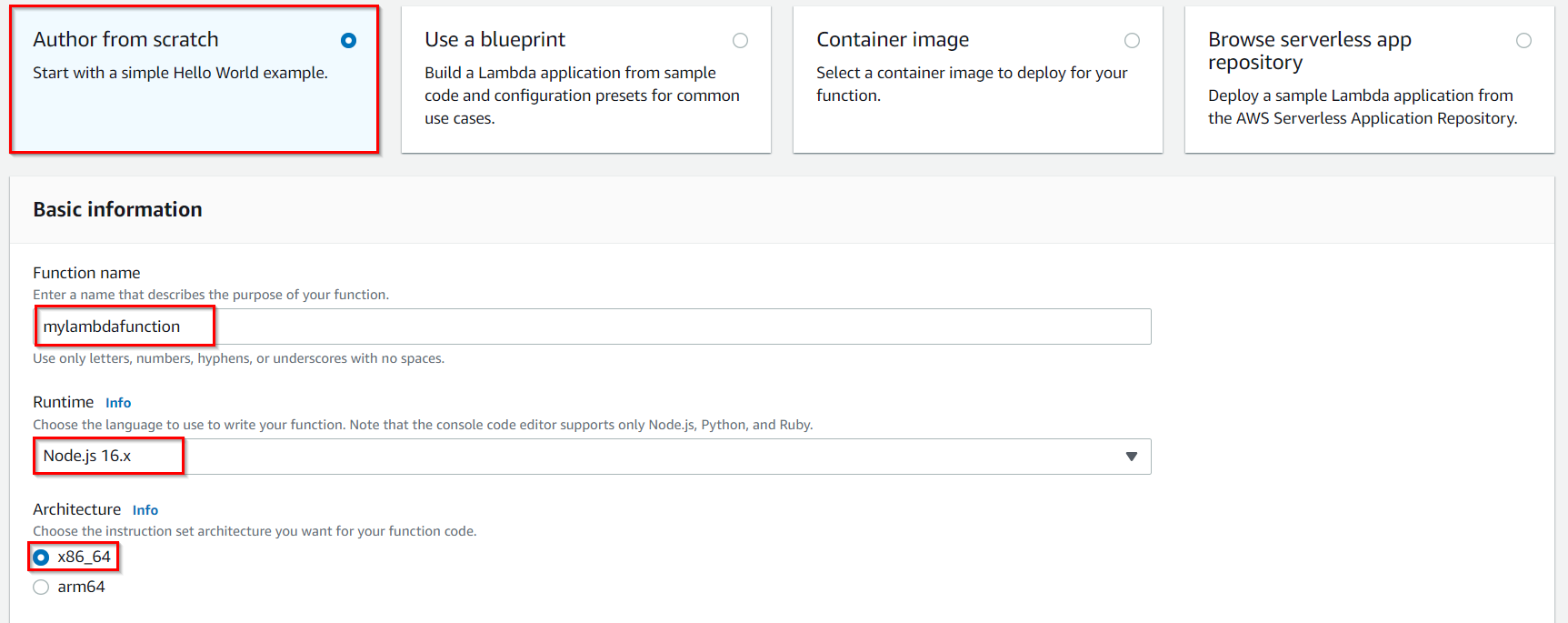
* Select your policy and click on the **Next** button.
* **Role Name:** Enter **myrole**
* Click on the **Create Role**button.

1. You have successfully created an IAM role by name **myrole**.

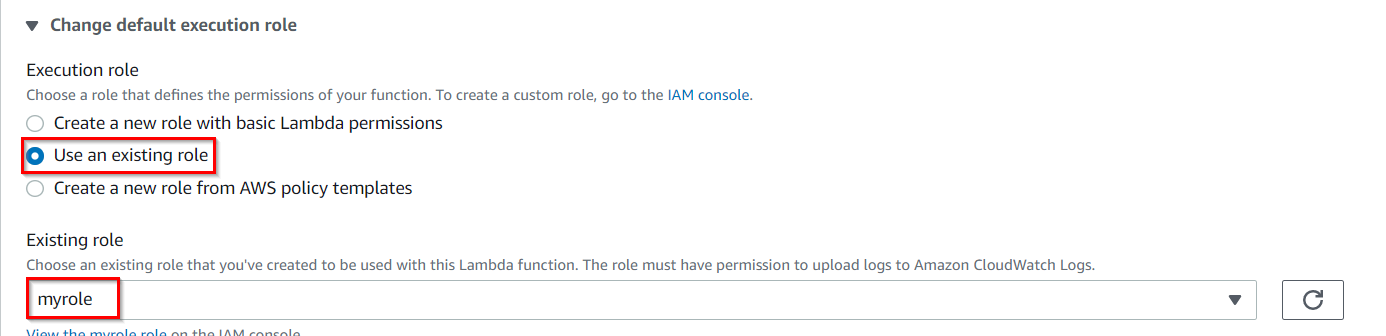


## Task 5: Create a Lambda Function

1. Make sure you are in the **US East (N. Virginia)** region.
2. Go to the **Services**menu and click on**Lambda** under **Compute**section.
3. Click on the **Create a function**button.
   * Choose **Author from scratch**
   * Function name : Enter ***mylambdafunction***
   * Runtime : Select **Node.js 16x**

****

* Role: In the permissions section, click on **Change default execution role**and then select **Use an existing role**.
* Existing role**:** Select **myrole** from the list.



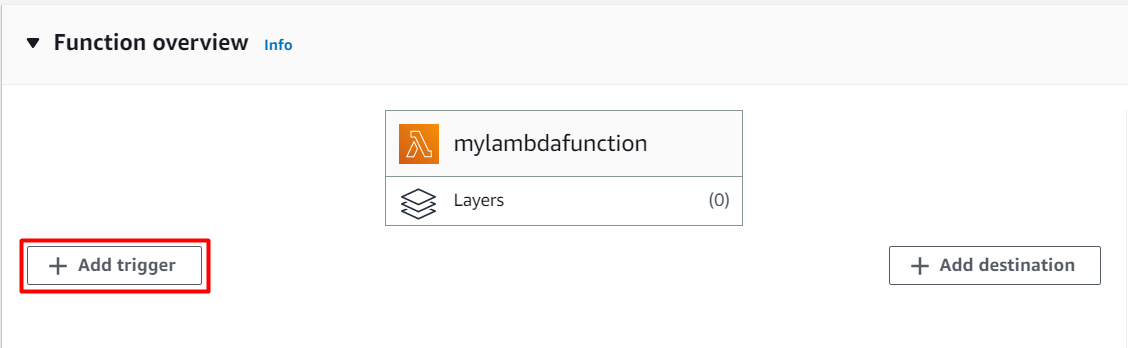
1. Click on the **Create function**button.
2. If you scroll down a little bit, you can see the **Code source**section. Here we need to write a NodeJs function that copies the object from the source bucket and paste it into the destination bucket.
3. Remove the existing code in AWS lambda index.js. Copy the below code and paste it into your lambda index.js file.

|  |
| --- |
| var AWS = require("aws-sdk");   exports.handler = async (event, context) => {     var s3 = new AWS.S3();     var sourceBucket = "mysourcebucket12345";     var destinationBucket = "mydestinationbucket12345";     var objectKey = event.Records[0].s3.object.key;     var copySource = encodeURI(`${sourceBucket}/${objectKey}`);     var copyParams = {       Bucket: destinationBucket,       CopySource: copySource,       Key: objectKey     };       try {       var result = await s3.copyObject(copyParams).promise();       console.log("S3 object copy successful.");     } catch (err) {       console.log(err, err.stack);     }  }; |

1. You need to change the**source** and **destination bucket name** in the index.js file **line number 5 & 6** based on your bucket names.
2. Save the function by clicking on **Deploy**button.

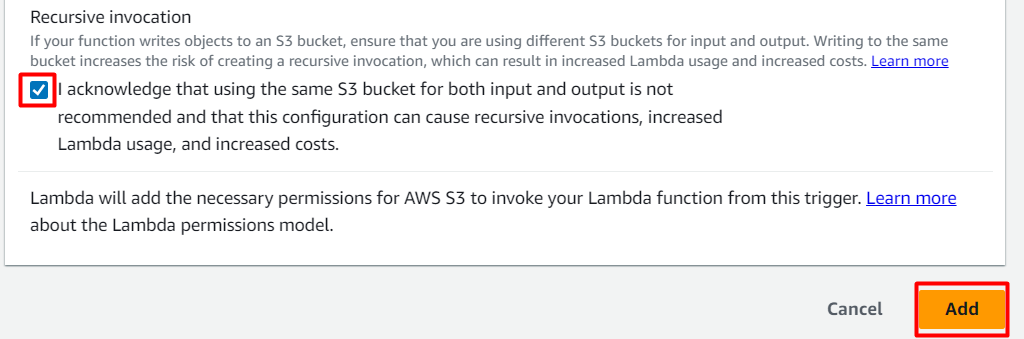
## Task 6: Adding Triggers to Lambda Function

1. Scroll up to go to **Function overview** and click on **+ Add trigger**button.



     2. Scroll down the list and select **S3** from the trigger list. Once you select S3, a form will appear. Enter these details:

* Bucket: Select your source bucket - **mysourcebucket12345**
* Event type: Select **All object create events**
* Leave other fields as default.
* And check this option of **Recursive invocation**to avoid failures in case you upload multiple files at once and **Check** the acknowledge option.
* Click on **Add**button.



## Task 7: Test Lambda function

1. If you have a test image on your local machine, you can use that image.
   * Click on [Download Me](https://labresources.whizlabs.com/a626482b5f325b4e320c63615c003b1d/smiley_26_32.jpeg) to open the image in new tab, right click the image and save it to your local.
2. Go to S3 Bucket list and click on source bucket - **mysourcebucket12345**.
3. Upload image to source S3 bucket. To do that:
   * Click on the **Upload** button.
   * Click on **Add files**button to add the files.
   * Select the image and click on the **Upload** to upload the image.
4. Now go back to the S3 list and open your destination bucket **- mydestinationbucket12345.**
5. You can see a copy of your uploaded source bucket image in the destination bucket.

### ****Do you know ?****

AWS Lambda has a generous free tier that allows you to run up to 1 million requests and 400,000 GB-seconds of compute time per month for free, making it an ideal choice for small-scale projects and experiments.