AWS Compute Blog

Scheduling SSH jobs using AWS Lambda

by Vyom Nagrani | on 12 FEB 2016 | Permalink | Comments | Share

Puneet Agarwal, AWS Solution Architect

With the addition of the Scheduled Events feature, you can now set up AWS Lambda to invoke your code on a regular, scheduled basis. You can now schedule various AWS API activities in your account (such as creation or deletion of CloudFormation stacks, EBS volume snapshots, etc.) with AWS Lambda. In addition, you can use AWS Lambda to connect to your Linux instances by using SSH and run desired commands and scripts at regular time intervals. This is especially useful for scheduling tasks (e.g., system updates, log cleanups, maintenance tasks) on your EC2 instances, when you don't want to manage cron or external schedulers for a dynamic fleet of instances.

In the following example, you will run a simple shell script that prints "Hello World" to an output file on instances tagged as "Environment=Dev" in your account. You will trigger this shell script through a Lambda function written in Python 2.7.

At a high level, this is what you will do in this example:

1. Create a Lambda function to fetch IP addresses of EC2 instances with "Environment=Dev" tag. This function will serve as a trigger function. This trigger function will invoke a worker function, for each IP address. The worker function will connect to EC2 instances using SSH and run a HelloWorld.sh script.
2. Configure **Scheduled Event** as an event source to invoke the trigger function every 15 minutes.
3. Create a Python deployment package (.zip file), with worker function code and other dependencies.
4. Upload the worker function package to AWS Lambda.

Advantages of Scheduled Lambda Events over Ubiquitous Cron

Cron is indeed simple and well understood, which makes it a very popular tool for running scheduled operations. However, there are many architectural benefits that make scheduled Lambda functions and custom scripts a better choice in certain scenarios:

- Decouple job schedule and AMI: If your cron jobs are part of an AMI, each schedule change requires you to create a new AMI version, and update existing instances running with that AMI. This is both cumbersome and time-consuming. Using scheduled Lambda functions, you can keep the job schedule outside of your AMI and change the schedule on the fly.
- Flexible targeting of EC2 instances: By abstracting the job schedule from AMI and EC2 instances, you can flexibly target a subset of your EC2 instance fleet based on tags or other conditions. In this example, we are targeting EC2 instances with the “Environment=Dev” tag.
- Intelligent scheduling: With scheduled Lambda functions, you can add custom logic to you abstracted job scheduler.

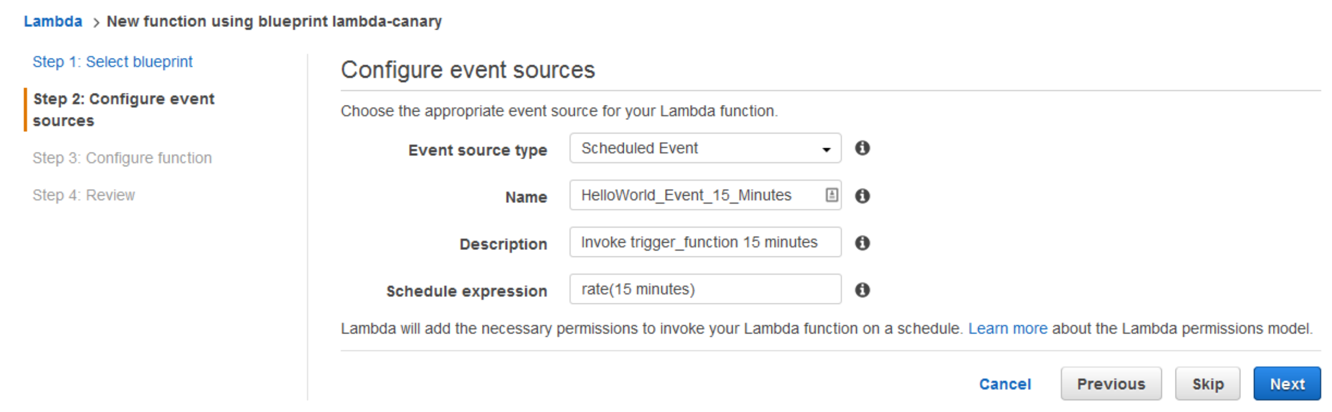
While there are many ways of achieving the above benefits, scheduled Lambda functions are an easy-to-use option in your toolkit.

Trigger Function

This is a simple Python function that extracts IP addresses of all instances with the “Environment=Dev” tag and invokes the worker function for each of the instances. Decoupling the trigger function from the worker function enables a simpler programming model for parallel execution of tasks on multiple instances.

Steps:

1. Sign in to the AWS Management Console and open the AWS Lambda console.
2. Choose **Create a Lambda function**.
3. On the **Select blueprint** page, type *cron* in the search box.
4. Choose **lambda-canary**.
5. On the **Configure event sources** page, **Event source type** defaults to **Scheduled Event**. You can create a new schedule by entering a name for the schedule, or can select one of your existing schedules. For **Schedule expression**, you can specify a fixed rate (number of minutes, hours, or days between invocations) or you can specify a cron-like expression. Note that rate frequencies of less than five minutes are not supported at this time.


Lambda > New function using blueprint lambda-canary

Step 1: Select blueprint
Step 2: Configure event sources
Step 3: Configure function
Step 4: Review

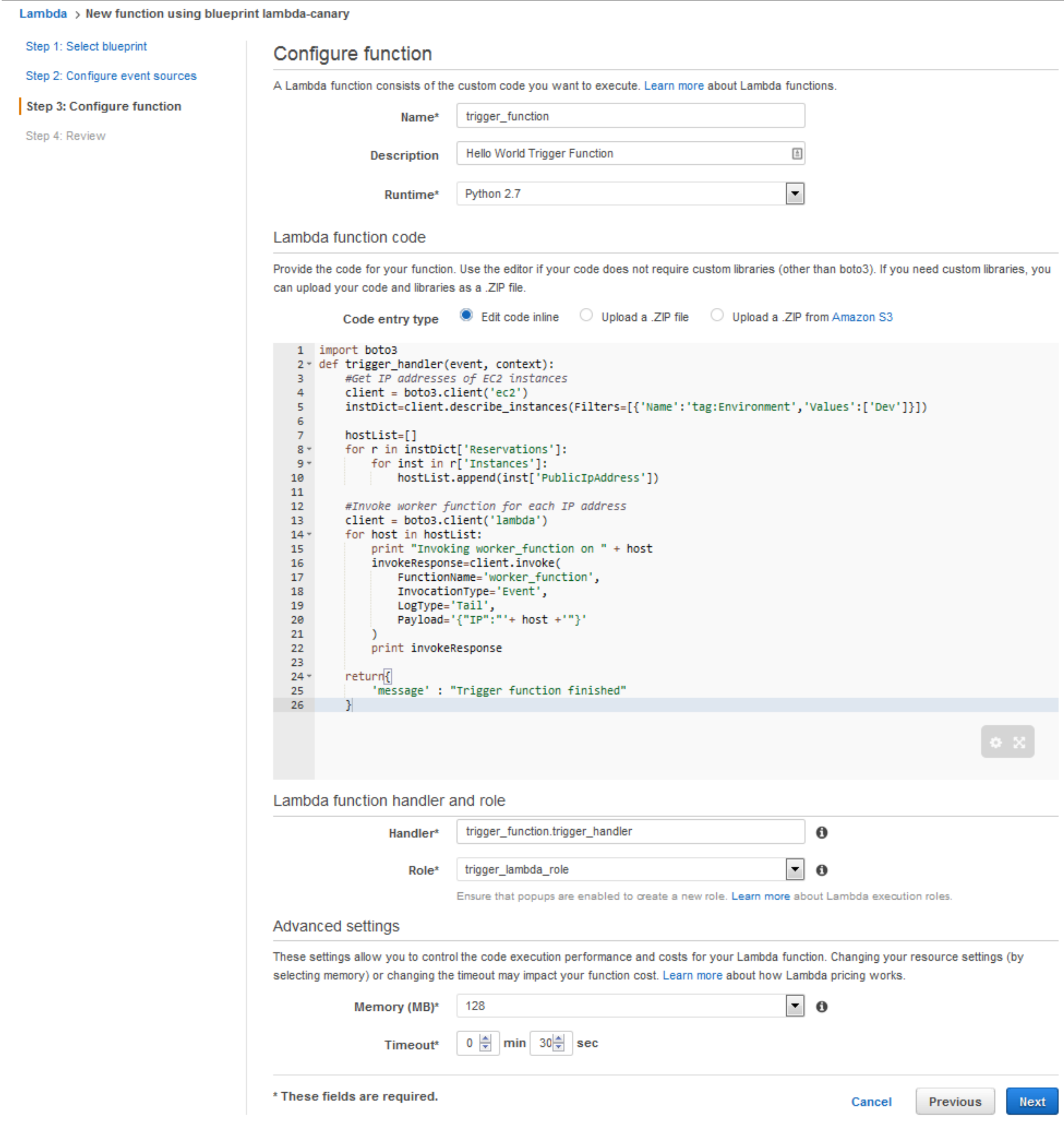
Configure event sources

Choose the appropriate event source for your Lambda function.

Event source type	Scheduled Event
Name	HelloWorld_Event_15_Minutes
Description	Invoke trigger_function 15 minutes
Schedule expression	rate(15 minutes)

Lambda will add the necessary permissions to invoke your Lambda function on a schedule. Learn more about the Lambda permissions model.

6. Choose **Next**. The **Configure Function** page appears.



Lambda › New function using blueprint lambda-canary

Step 1: Select blueprint
Step 2: Configure event sources
Step 3: Configure function
Step 4: Review

Configure function

A Lambda function consists of the custom code you want to execute. Learn more about Lambda functions.

Name* trigger_function

Description Hello World Trigger Function

Runtime* Python 2.7

Lambda function code

Provide the code for your function. Use the editor if your code does not require custom libraries (other than boto3). If you need custom libraries, you can upload your code and libraries as a .ZIP file.

Code entry type [✓ Edit code inline] Upload a .ZIP file Upload a .ZIP from Amazon S3

```
import boto3
def trigger_handler(event, context):
    #Get IP addresses of EC2 instances
    client = boto3.client('ec2')
    instDict=client.describe_instances(Filters=[{'Name':'tag:Environment','Values':['Dev']}])

    hostList=[]
    for r in instDict['Reservations']:
        for inst in r['Instances']:
            hostList.append(inst['PublicIpAddress'])

    #Invoke worker function for each IP address
    client = boto3.client('lambda')
    for host in hostList:
        print "Invoking worker_function on " + host
        invokeResponse=client.invoke(
            FunctionName='worker_function',
            InvocationType='Event',
            LogType='Tail',
            Payload='{"IP":"'+ host +'"}'
        )
        print invokeResponse

    return{
        'message' : "Trigger function finished"
    }
```

Lambda function handler and role

Handler* trigger_function.trigger_handler

Role* trigger_lambda_role

Ensure that popups are enabled to create a new role. Learn more about Lambda execution roles.

Advanced settings

These settings allow you to control the code execution performance and costs for your Lambda function. Changing your resource settings (by selecting memory) or changing the timeout may impact your function cost. Learn more about how Lambda pricing works.

Memory (MB)* 128

Timeout* 0 min 30 sec

* These fields are required.

Cancel Previous Next

Here, you can enter the name and description of your function. Replace the sample code here with the following code.

trigger_function.py

```
import boto3

def trigger_handler(event, context):
    #Get IP addresses of EC2 instances
    client = boto3.client('ec2')
    instDict=client.describe_instances(
            Filters=[{'Name':'tag:Environment','Values':['Dev']}]
```

```
        )

    hostList=[]
    for r in instDict['Reservations']:
        for inst in r['Instances']:
            hostList.append(inst['PublicIpAddress'])

    #Invoke worker function for each IP address
    client = boto3.client('lambda')
    for host in hostList:
        print "Invoking worker_function on " + host
        invokeResponse=client.invoke(
            FunctionName='worker_function',
            InvocationType='Event',
            LogType='Tail',
            Payload='{"IP":"'+ host +'"}'
        )
        print invokeResponse

    return{
        'message' : "Trigger function finished"
    }
```

7. After adding the trigger code in the console, create the appropriate execution role and set a timeout. Note that the execution role must have permissions to execute EC2 DescribeInstances and invoke Lambda functions. Example IAM Policies for the trigger Lambda role are as follows:
    - Basic execution policy: https://gist.github.com/apun/8f8c0c0cbea38d7e0bdc (automatically created by AWS Console).
    - Trigger Policy: https://gist.github.com/apun/33c2fd954a8e238bbcb0 (EC2:Describe* and InvokeFunction permissions to invoke worker_function). After role creation, you can add this policy to the trigger_lambda_role using the IAM console.
8. Choose **Next**, choose **Enable later**, and then choose **Create function**.

Worker Function

Next, put together the worker Lambda function that connects to an Amazon EC2 instance using SSH, and then run the HelloWorld.sh script. To initiate SSH connections from the Lambda client, use the Paramiko library. Paramiko is an open source Python implementation of the SSHv2 protocol, providing both client and server functionality. Worker function will irst download a private key file from a secured Amazon S3 bucket to the local /tmp folder, and then use that key file to connect to the EC2 instances by using SSH. You must keep your private key secure and make sure that only the worker function has read access to the file on S3. Assuming that

EC2 instances have S3 access permissions through an EC2 role, worker function will download the HelloWorld.sh script from S3 and execute it locally on each EC2 instance.

Steps:

1. Create worker_function.py file on your local Linux machine or on an EC2 instance using following code **worker_function.py**

```
import boto3
import paramiko
def worker_handler(event, context):

    s3_client = boto3.client('s3')
    #Download private key file from secure S3 bucket
    s3_client.download_file('s3-key-bucket','keys/keyname.pem', '/tmp/keyname.

    k = paramiko.RSAKey.from_private_key_file("/tmp/keyname.pem")
    c = paramiko.SSHClient()
    c.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    host=event['IP']
    print "Connecting to " + host
    c.connect( hostname = host, username = "ec2-user", pkey = k )
    print "Connected to " + host

    commands = [
        "aws s3 cp s3://s3-bucket/scripts/HelloWorld.sh /home/ec2-user/HelloWo
        "chmod 700 /home/ec2-user/HelloWorld.sh",
        "/home/ec2-user/HelloWorld.sh"
        ]
    for command in commands:
        print "Executing {}".format(command)
        stdin , stdout, stderr = c.exec_command(command)
        print stdout.read()
        print stderr.read()

    return
    {
        'message' : "Script execution completed. See Cloudwatch logs for compl
    }
```

Now, creating a deployment package is straightforward. For this example, create a deployment package using Virtualenv.

2. Install Virtualenv on your local Linux machine or an EC2 instance.

```
$ pip install virtualenv
```

3. Create a virtual environment named “helloworld-env“, which will use a Python2.7 interpreter.

```
$ virtualenv –p /usr/bin/python2.7 path/to/my/helloworld-env
```

4. Activate helloworld-env.

```
source path/to/my/helloworld-env/bin/activate
```

5. Install dependencies.

```
$pip install pycrypto
```

PyCrypto provides the low-level (C-based) encryption algorithms we need to implement the SSH protocol.

```
$pip install paramiko
```

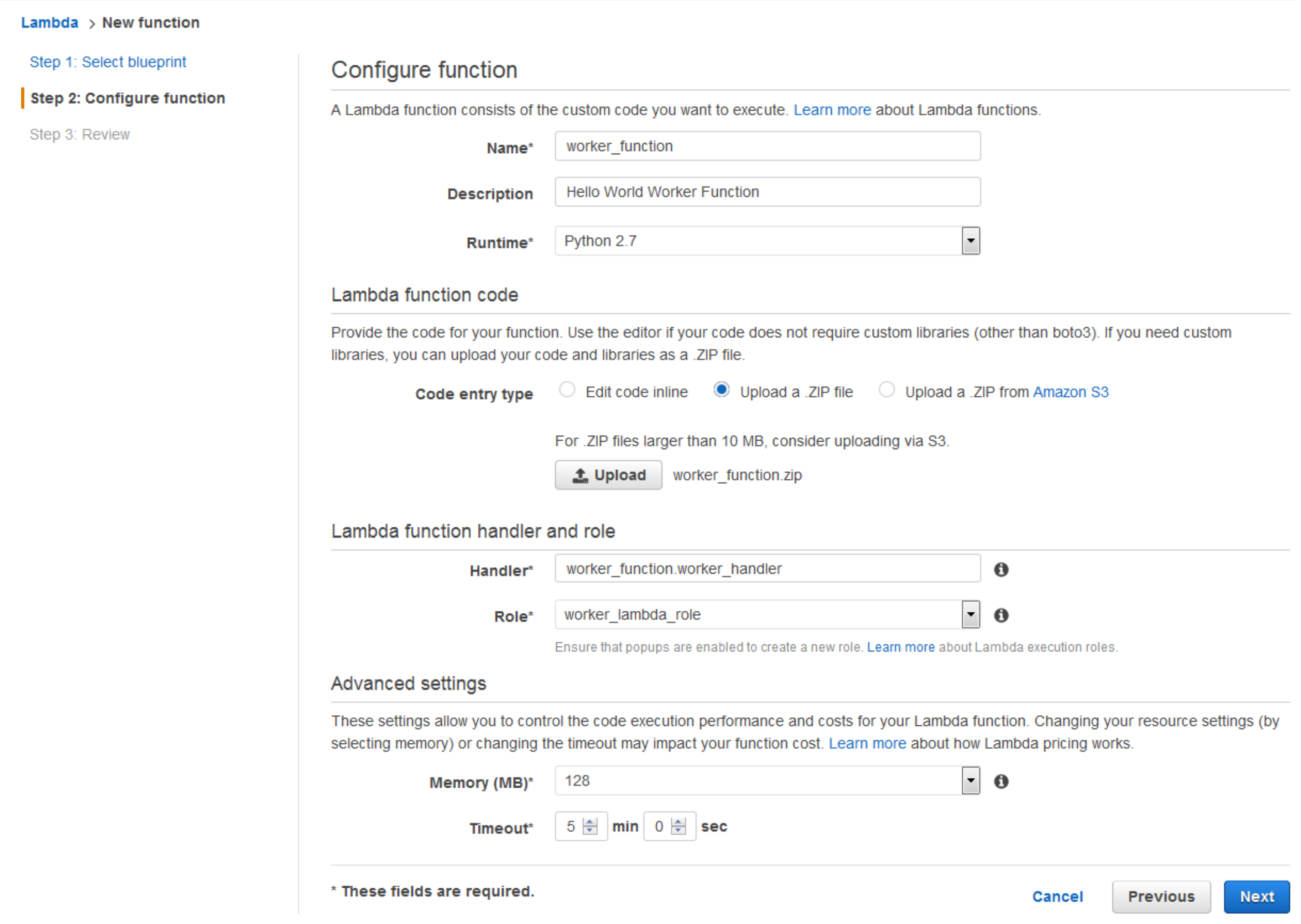
6. Add worker_function.py to the zip file.

```
$zip path/to/zip/worker_function.zip worker_function.py
```

7. Add dependencies from helloworld-env to the zip file.

```
$cd path/to/my/helloworld-env/lib/python2.7/site-packages
$zip –r path/to/zip/worker_function.zip
$cd path/to/my/helloworld-env/lib64/python2.7/site-packages
$zip –r path/to/zip/worker_function.zip
```

Using the AWS console (skip the blueprint step) or AWS CLI, create a new Lambda function named worker_function and upload worker_function.zip.



Lambda > New function

Step 1: Select blueprint
Step 2: Configure function
Step 3: Review

Configure function

A Lambda function consists of the custom code you want to execute. Learn more about Lambda functions.

Name*: worker_function
Description: Hello World Worker Function
Runtime*: Python 2.7

Lambda function code

Provide the code for your function. Use the editor if your code does not require custom libraries (other than boto3). If you need custom libraries, you can upload your code and libraries as a .ZIP file.

Code entry type: Edit code inline [✓ Upload a .ZIP file] Upload a .ZIP from Amazon S3

For .ZIP files larger than 10 MB, consider uploading via S3.

Upload worker_function.zip

Lambda function handler and role

Handler*: worker_function.worker_handler
Role*: worker_lambda_role

Ensure that popups are enabled to create a new role. Learn more about Lambda execution roles.

Advanced settings

These settings allow you to control the code execution performance and costs for your Lambda function. Changing your resource settings (by selecting memory) or changing the timeout may impact your function cost. Learn more about how Lambda pricing works.

Memory (MB)*: 128
Timeout*: 5 min 0 sec

* These fields are required.

Example IAM policies for the worker Lambda role are as follows:

- Basic execution policy: https://gist.github.com/apun/8f8c0c0cbea38d7e0bdc (Automatically created by AWS Console)
- Worker policy: https://gist.github.com/apun/0647280645b399917191 (GetObject permission for S3 key file)

Caution: To keep your keys secure, make sure no other IAM users or roles, other than intended users, have access to this S3 bucket.

Upload key and script to S3

All you need to do now is upload your key and script file to S3 buckets and then you are ready to run the example.

Steps:

1. Upload HellowWorld.sh to an appropriate S3 bucket (e.g., s3://s3-bucket/scripts/). HelloWorld.sh is a simple shell script that prints "Hello World from instanceID" to a log file and copies that log file to your S3 folder.

HelloWorld.sh

```
#Get instanceId from metadata
instanceid=`wget -q -O - http://instance-data/latest/meta-data/instance-id`
LOGFILE="/home/ec2-user/$instanceid.$(date +"%Y%m%d_%H%M%S").log"

#Run Hello World and redirect output to a log file
echo "Hello World from $instanceid" > $LOGFILE

#Copy log file to S3 logs folder
aws s3 cp $LOGFILE s3://s3-bucket/logs/
```

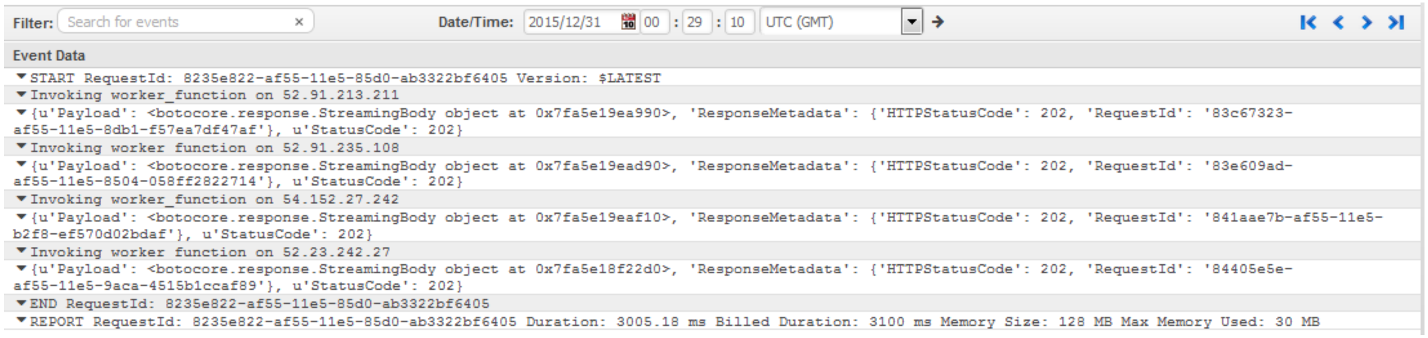
2. Upload keyname.pem file, which is your private key to connect to EC2 instances, to a secure S3 bucket (e.g., s3://s3-key-bucket/keys/keyname.pem). To keep your keys secure, make sure no IAM users or roles, other than intended users and the Lambda worker role, have access to this S3 bucket.

Running the example

As a final step, enable your trigger_function event source by choosing **trigger_function** from the list of Lambda functions, choosing the **Event sources** tab, and clicking **Disabled** in the **State** column.

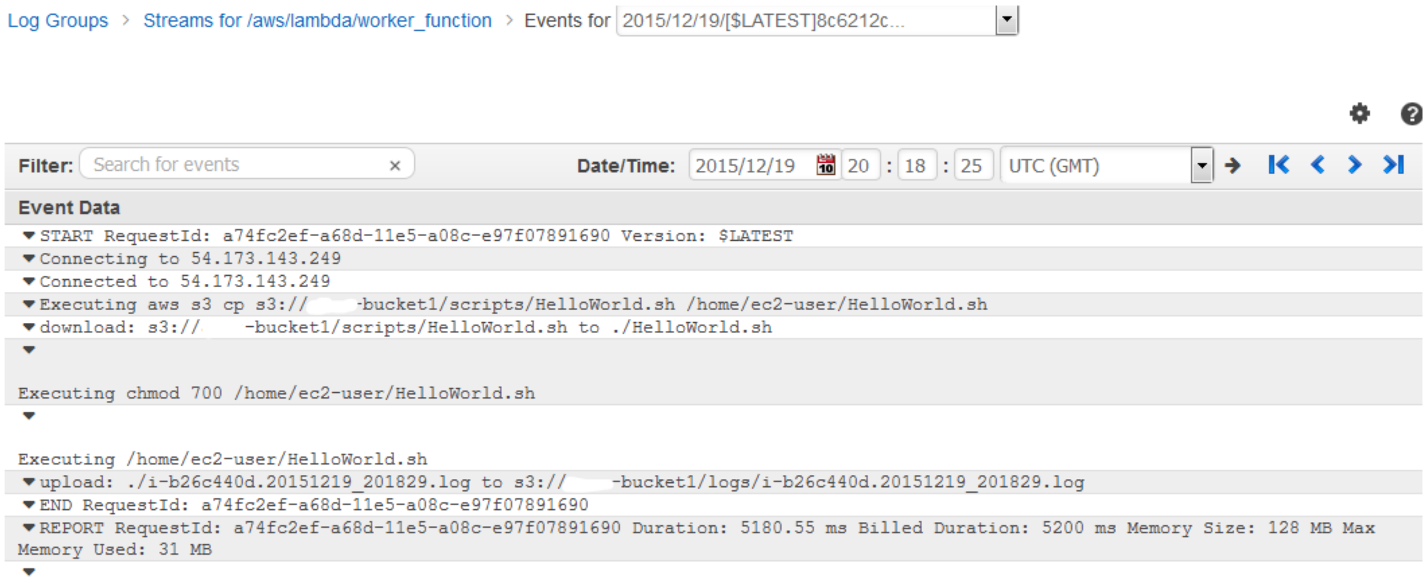
You can now test your newly created Lambda functions and monitor execution logs. AWS Lambda logs all requests handled by your function and automatically stores logs generated by your code using Amazon CloudWatch Logs. The following screenshots show my CloudWatch Logs after completing the preceding steps.

Trigger function log in CloudWatch Logs:



```
START RequestId: 8235e822-af55-11e5-85d0-ab3322bf6405 Version: $LATEST
Invoking worker_function on 52.91.213.211
{u'Payload': <botocore.response.StreamingBody object at 0x7fa5e19ea990>, 'ResponseMetadata': {'HTTPStatusCode': 202, 'RequestId': '83c67323-
af55-11e5-8db1-f57ea7df47af'}, u'StatusCode': 202}
Invoking worker function on 52.91.235.108
{u'Payload': <botocore.response.StreamingBody object at 0x7fa5e19ead90>, 'ResponseMetadata': {'HTTPStatusCode': 202, 'RequestId': '83e609ad-
af55-11e5-8504-058ff2822714'}, u'StatusCode': 202}
Invoking worker_function on 54.152.27.242
{u'Payload': <botocore.response.StreamingBody object at 0x7fa5e19eaf10>, 'ResponseMetadata': {'HTTPStatusCode': 202, 'RequestId': '841aae7b-af55-11e5-
b2f8-ef570d02bdaf'}, u'StatusCode': 202}
Invoking worker function on 52.23.242.27
{u'Payload': <botocore.response.StreamingBody object at 0x7fa5e18f22d0>, 'ResponseMetadata': {'HTTPStatusCode': 202, 'RequestId': '84405e5e-
af55-11e5-9aca-4515b1ccaf89'}, u'StatusCode': 202}
END RequestId: 8235e822-af55-11e5-85d0-ab3322bf6405
REPORT RequestId: 8235e822-af55-11e5-85d0-ab3322bf6405 Duration: 3005.18 ms Billed Duration: 3100 ms Memory Size: 128 MB Max Memory Used: 30 MB
```

Worker function log in Cloudwatch Logs:

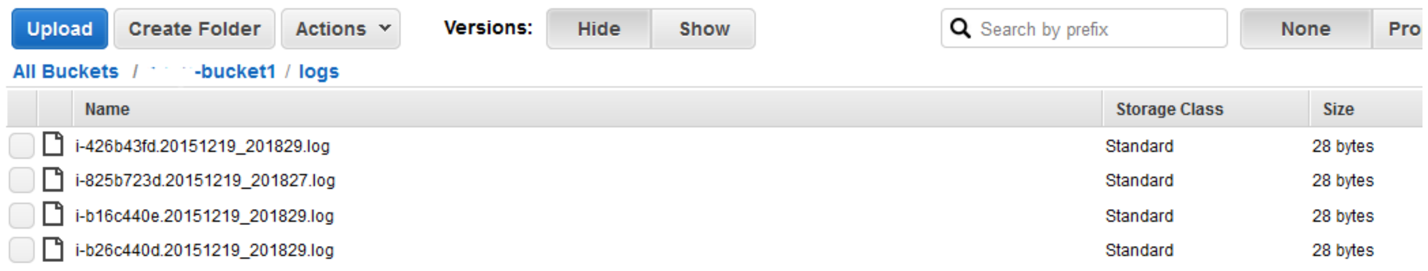

Log Groups > Streams for /aws/lambda/worker_function > Events for 2015/12/19/[$LATEST]8c6212c...
Filter: Search for events
Date/Time: 2015/12/19 20 : 18 : 25 UTC (GMT)
Event Data

```
START RequestId: a74fc2ef-a68d-11e5-a08c-e97f07891690 Version: $LATEST
Connecting to 54.173.143.249
Connected to 54.173.143.249
Executing aws s3 cp s3://      -bucket1/scripts/HelloWorld.sh /home/ec2-user/HelloWorld.sh
download: s3://      -bucket1/scripts/HelloWorld.sh to ./HelloWorld.sh

Executing chmod 700 /home/ec2-user/HelloWorld.sh

Executing /home/ec2-user/HelloWorld.sh
upload: ./i-b26c440d.20151219_201829.log to s3://      -bucket1/logs/i-b26c440d.20151219_201829.log
END RequestId: a74fc2ef-a68d-11e5-a08c-e97f07891690
REPORT RequestId: a74fc2ef-a68d-11e5-a08c-e97f07891690 Duration: 5180.55 ms Billed Duration: 5200 ms Memory Size: 128 MB Max
Memory Used: 31 MB
```

Log files that were generated in my S3 bucket:



Name	Storage Class	Size
i-426b43fd.20151219_201829.log	Standard	28 bytes
i-825b723d.20151219_201827.log	Standard	28 bytes
i-b16c440e.20151219_201829.log	Standard	28 bytes
i-b26c440d.20151219_201829.log	Standard	28 bytes

Other considerations

- With the new Lambda VPC support, you can connect to your EC2 instances running in your private VPC by providing private subnet IDs and EC2 security group IDs as part of your Lambda function configuration.
- AWS Lambda now supports a maximum function duration of 5 minutes, and so you can use scheduled Lambda functions to run jobs that are expected to finish within 5 minutes. For longer running jobs, you can use following syntax to run jobs in the background so that the Lambda function doesn't wait for command execution to finish.

```
c.exec_command(cmd + ' > /dev/null 2>&1 &')
```

Resources

Serverless Computing and Applications
Amazon Container Services
AWS Messaging
Cloud Compute with AWS
Desktop and Application Streaming

Follow

Twitter
Facebook
LinkedIn
Twitch
Email Updates

Related Posts

Detect vulnerabilities in the Docker images in your applications

Stream your FlexApps with your Amazon AppStream 2.0 environment

Announcing the Winners of AWS Educate’s Alexa Skills Challenge

Build custom chat bots for Amazon Chime

ICYMI: Serverless Q1 2019

Creating dynamic, personalized experiences in Amazon Connect

AWS Makes It Easier for Embedded Developers to Build IoT Applications with Additional Preconfigured Examples for FreeRTOS on Armv8-M Architectures

Getting more visibility into GraphQL performance with AWS AppSync logs