

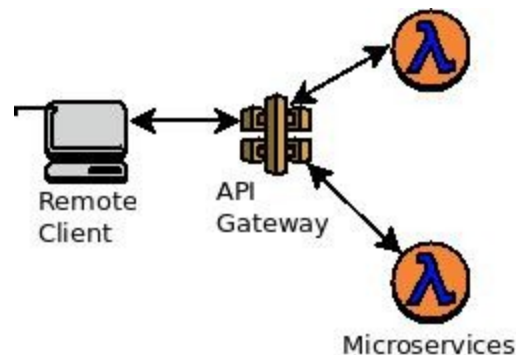
Tutorial 4 - Introduction to AWS Lambda with the Serverless Application Analytics Framework (SAAF)

Disclaimer: Subject to updates as corrections are found

Version 0.17

Scoring: 40 pts maximum

The purpose of this tutorial is to introduce creating Function-as-a-Service functions on the AWS Lambda FaaS platform, and then to create a simple two-service application where the application flow control is managed by the client:



This tutorial will focus on developing Lambda functions in Java using the Serverless Application Analytics Framework (SAAF). SAAF enables identification of the underlying cloud infrastructure used to host FaaS functions while supporting profiling performance and resource utilization of functions. SAAF helps identify infrastructure state to determine COLD vs. WARM infrastructure to help track and understand performance implications resulting from the serverless Freeze-Thaw infrastructure lifecycle .

1. Download SAAF

To begin, using git, clone SAAF.

If you do not already have git installed, please do so.

On ubuntu see the official documentation:

<https://help.ubuntu.com/its/serverguide/git.html.en>

For a full tutorial on the use of git, here is an old tutorial for TCSS 360:

http://faculty.washington.edu/wlloyd/courses/tcss360/assignments/TCSS360_w2017_Tutorial_1.pdf

If you prefer using a GUI-based tool, on Windows/Mac check out the GitHub Desktop:

<https://desktop.github.com/>

Once having access to a git client, clone the source repository:

```
git clone https://github.com/wlloyduw/SAAF.git
```

For tutorial #1, we will focus on using the SAAF provided AWS Lambda Java function template provided as a maven project. If you're familiar with Maven as a build environment, you can simply edit your Java Lambda function code using any text editor such as vi, emacs, pico/nano. However, working with an IDE tends to be easier, and many Java IDEs will open maven projects directly.

Next update your apt repository and local Ubuntu packages:

```
sudo apt update  
sudo apt upgrade
```

To install maven on Ubuntu:

```
sudo apt install maven
```

2. Build the SAAF Lambda function Hello World template

If you have a favorite Java IDE with maven support, feel free to try to open and work with the maven project directly. This is confirmed to work in Apache Netbeans 11 LTS. Other popular Java IDEs include Eclipse and IntelliJ.

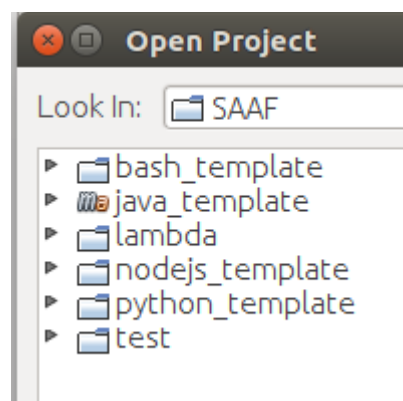
Download Netbeans 11.0

<https://netbeans.apache.org/download/nb110/nb110.html>

Once you've downloaded Netbeans, you'll be able to open the project directly.

Select "File | Open Project".

Then locate your clone git project, and drill down: SAAF → java_template



Then on the left-hand side, expand the “Source Packages” folder.

You’ll see a “java_template” folder.
Expand this.

This contains a Hello World Lambda function as a starter template.

There are three class files.

Hello.java	Provides an example implementation of a Java AWS Lambda Function. The <code>handleRequest()</code> method is called by Lambda as the entry point of your Lambda FaaS function. You’ll see where in this function template your code should be inserted. <code>Hello.java</code> uses a <code>HashMap</code> for the request and response JSON objects. The incoming response JSON is “serialized” into a <code>hashmap</code> automatically by Lambda. The outgoing response JSON is created based on the <code>HashMap</code> that is returned.
HelloPOJO.java	<code>HelloPOJO.java</code> is the same as <code>Hello.java</code> except that instead of using a <code>HashMap</code> for the Request (incoming) data, instead an explicitly defined <code>Request</code> class is defined with getter and setter methods to accept input from the user. The advantage with <code>HelloPOJO</code> is the <code>Request</code> object can perform post-processing on input parameters provided from the function caller before they are used. Post-processing includes operations such as formatting data or transforming values into another form before actual use in the FaaS function. User inputs to the FaaS function could trigger other behavior in the FaaS function automatically when the values are loaded.
HelloMain.java	<code>HelloMain.java</code> is identical to <code>Hello.java</code> except that it also contains a public static void <code>main()</code> method to allow command line execution of the function package. This template is provided as an example. This allows Lambda functions to be first tested locally on the command line before deployment to Lambda. The local implementation could also be used to facilitate off-line unit testing of FaaS functions. As you develop your FaaS function, it will be necessary to continue to add to the implementation of the <code>main()</code> method to include required parameters for interacting with the function. The <code>main()</code> method creates a mock <code>Context()</code> object which fools the program into thinking it is running in context of AWS Lambda.
Request.java	This class is a Plain Old Java Object (POJO). You’ll want to define getter and setter methods and private variables to capture data sent from the client to the Lambda function. JSON that is sent to your Lambda function is automatically marshalled into this Java object for easy consumption at runtime.
Response.java	(REMOVED) There is no longer a <code>Response</code> class POJO. This has been removed in favor of simply using a <code>HashMap</code> . A <code>Response</code> POJO could be implemented alternatively to add logic to getter and setter

methods to perform data formatting, transformation, or validation operations.

Now compile the project using maven from the command line (or your IDE):

From the “SAAF/java_template” directory:

```
# Clean and remove old build artifacts
mvn clean -f pom.xml
```

Then rebuild the project jar file:

```
# Rebuild the project jar file
mvn verify -f pom.xml
```

In NetBeans right click on the name of the project “java_template” in the left-hand list of Projects and click “Clean and Build”.

3. Test Lambda function locally before deployment

From a terminal, navigate to:

```
cd {base directory where project was cloned}/SAAF/java_template/target
```

Execute your function from the command line to first test your Lambda function locally:

```
java -cp lambda_test-1.0-SNAPSHOT.jar lambda.HelloMain Susan
```

Output should be provided as follows:

```
cmd-line param name=Susan
function result:{cpuType=Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz,
cpuNiceDelta=0, vmuptime=1570245300, cpuModel=94, linuxVersion=#193-Ubuntu
SMP Tue Sep 17 17:42:52 UTC 2019, cpuSoftIrqDelta=0, cpuUshrDelta=1,
uuid=e5faf33b-154b-4224-bb45-2904abfb9897, platform=Unknown Platform,
contextSwitches=3034407068, cpuKrn=9920122, cpuIdleDelta=7,
cpuIowaitDelta=0, newcontainer=0, cpuNice=33510, lang=java,
cpuUshr=19443782, majorPageFaultsDelta=0, freeMemory=1743632, value=Hello
Susan! This is from a response object!, frameworkRuntime=61,
contextSwitchesDelta=133, vmcpusteal=0, cpuKrnDelta=0, cpuIdle=32013339,
runtime=73, message=Hello Susan! This is a custom attribute added as
output from SAAF!, version=0.31, cpuIrqDelta=0, pageFaultsDelta=324,
cpuIrq=0, totalMemory=32318976, cpuCores=4, cpuSoftIrq=60350,
cpuIowait=582306, majorPageFaults=11984, vmcpustealDelta=0,
pageFaults=953729377, userRuntime=11}
```

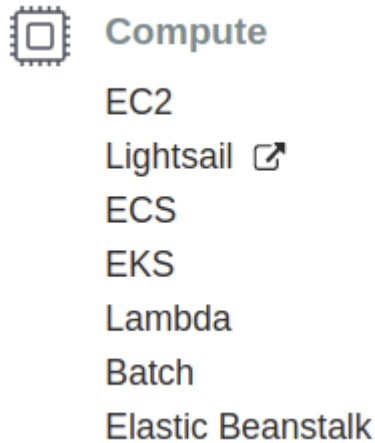
Whoa! That’s a lot of output! The actual Lambda function output is highlighted.

Other values represents data collected by the framework. Of course since you're testing locally, this data is for your local Linux environment, not the cloud.

4. Deploy the function to AWS Lambda

If the Lambda function has worked locally, the next step is to deploy to AWS Lambda.

Log into your AWS account, and locate under "Compute" services, "Lambda":



Click the button to create a new Function:

Create function

Using the wizard, use the "Author from scratch" mode. provide the following values:

Basic information

Function name
Enter a name that describes the purpose of your function.
hello

Runtime [Info](#)
Choose the language to use to write your function.
Java 8

Permissions [Info](#)
Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can configure and modify permissions further when you add triggers.

▼ Choose or create an execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

☐ Create a new role with basic Lambda permissions
☐ Use an existing role
☒ Create a new role from AWS policy templates

Role creation might take a few minutes. The new role will be scoped to the current function. To use it with other functions, you can modify it in the IAM console.

Role name
Enter a name for your new role.
simple_microservice_role

Policy templates - optional [Info](#)
Choose one or more policy templates.
Simple microservice permissions X
DynamoDB

Function name: hello
Runtime: Java 8
Execution Role: "Create a new role from AWS policy templates"
Role name: simple_microservice_role
(Roles can be inspected under IAM | Roles in the AWS Management Console)
Policy templates: "Simple microservice permissions"

Once filling the form, click the button:

Create function

Next, upload your compiled Java JAR file to AWS Lambda:

Click the "Upload" button to navigate and locate your JAR file. The jar file is under the "target" directory. It should be called "lambda_test-1.0-SNAPSHOT.jar".

Once selecting the file, change the "Handler" to:

```
lambda.Hello::handleRequest
```

***** IF THE HANDLER IS NOT UPDATED, LAMBDA WILL NOT BE ABLE TO LOCATE THE ENTRY POINT TO YOUR CODE. THE LAMBDA FUNCTION WILL FAIL TO RUN *****

Then press the "Save" button in the upper-righthand corner of the screen.

5. Create an API-Gateway REST URL

Next, in the AWS Management Console, navigate to the **API Gateway**.

This appears under the Network & Content Delivery services group:



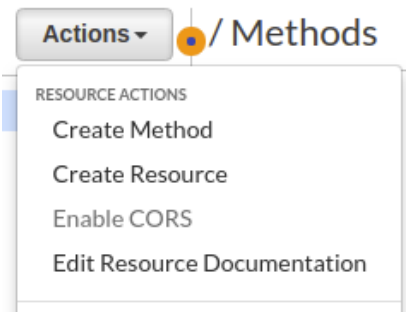
The very first time you visit the API Gateway console screen, there will be a “splash” screen. Select the button:



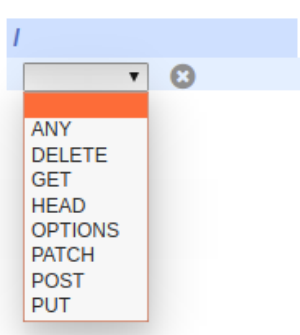
A message may be displayed regarding a sample API (Pet Store) developed with Swagger 2.0. Click OK. We will instead create a new REST API.

Choose the Protocol: select REST
Create new API: select New API
Settings:
API Name: hello_562
Description: <can leave blank>
Endpoint Type: <can accept default of Regional>

Next, pull down the Actions button-menu, and select “Create Method”:



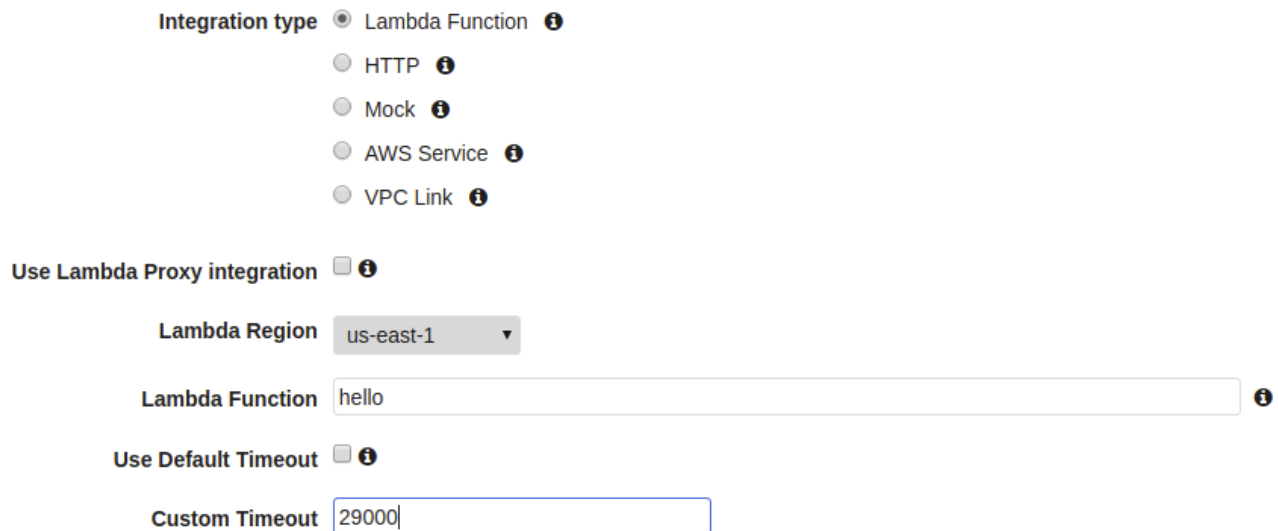
Select, drop down the menu and select “Post”:



Next, press the “checkmark” icon so it turns green:

A snippet of the AWS API Gateway console. It shows a blue header bar with a white checkmark icon. Below it, a dropdown menu is set to 'POST'. To the right of the dropdown are two circular icons: a green checkmark and a grey 'X'.

Then complete the form.

A screenshot of the AWS API Gateway console form for creating a new integration. The 'Integration type' is set to 'Lambda Function'. The 'Use Lambda Proxy integration' checkbox is unchecked. The 'Lambda Region' is set to 'us-east-1'. The 'Lambda Function' field contains the text 'hello'. The 'Use Default Timeout' checkbox is unchecked. The 'Custom Timeout' field contains the text '29000'. Information icons are present next to several fields.

Fill in “Lambda Function” to match your function name “hello”.

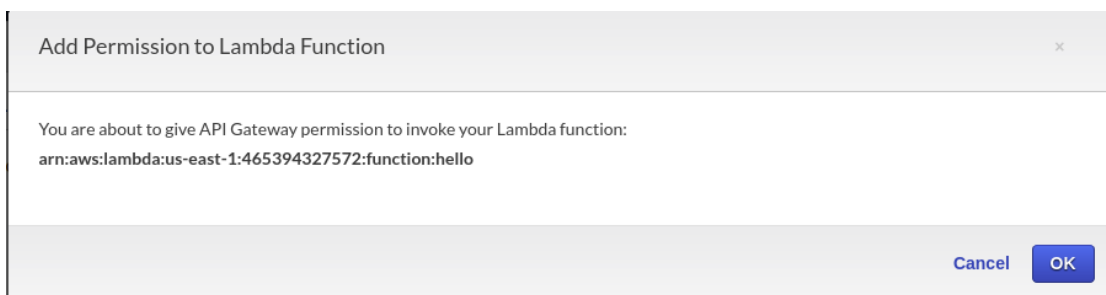
Uncheck the “Use Default Timeout”.

The API Gateway default time out for synchronous calls can be set between 50 and 29,000 milliseconds. Here provide the maximum synchronous timeout “29000”.

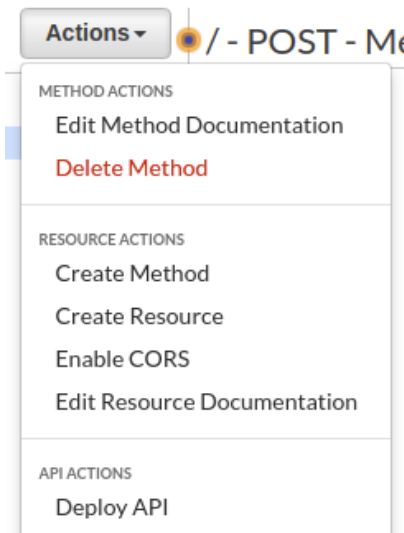
Then click Save:

A blue rectangular button with the word 'Save' in white text.

Next, acknowledge the permission popup:

A dialog box titled 'Add Permission to Lambda Function'. It contains the text: 'You are about to give API Gateway permission to invoke your Lambda function: arn:aws:lambda:us-east-1:465394327572:function:hello'. At the bottom right, there are two buttons: 'Cancel' and 'OK'.

Then, select the Actions drop-down and select Deploy API:



Next complete the form:

Deploy API

Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.

Deployment stage

[New Stage]

Stage name*

hello_dev

Stage description

Deployment description

Cancel

Deploy

The API-Gateway allows many URLs to be configured as REST webservice backends. The API-Gateway is not limited to AWS Lambda functions. It can also point to other backends hosted by AWS. The most common is to specify a “HTTP” path. This causes the API-Gateway to provide a new AWS hosted URL that is a proxy to an existing one. This allows all traffic to be routed to the URL to go through the API-Gateway for logging and/or processing.

Using the API-Gateway it is also possible to host multiple implementations of a function to support Agile software development processes. An organization may want to maintain

multiple live version of a function in various stages of development such as : (dev)elopment, test, staging and (prod)uction.
When complete, press the [Deploy] button.
The Stage name is appended to the end of the URL.

A stage editor should then appear with a REST URL to your AWS Lambda function.

COPY THIS URL TO THE CLIPBOARD:

Mouse over the URL, -right-click- and select "Copy link address":

hello_dev Stage Editor

Invoke URL: https://XXXXXXXXXXXX.execute-api.us-east-1.amazonaws.com/hello_dev

6. Install packages and configure your client to call Lambda

Next, return to the command prompt and create and navigate to a new directory

```
cd {base directory where project was cloned}/SAAF/java_template/  
mkdir test  
cd test
```

Using a text editor such as vi, pico, nano, vim, or gedit, create a file called "callservice.sh"

Locate the lines:

```
#!/bin/bash  
  
# JSON object to pass to Lambda Function  
json="{\"name\":\"Susan\\u0020Smith\", \"param1\":\"1\", \"param2\":\"2\", \"param3\":\"3\"}"  
  
echo "Invoking Lambda function using API Gateway"  
time output=`curl -s -H "Content-Type: application/json" -X POST -d $json {INSERT API GATEWAY URL HERE}`  
echo ""  
  
echo "Invoking Lambda function using AWS CLI"  
time output=`aws lambda invoke --invocation-type RequestResponse --function-name {INSERT AWS FUNCTION NAME HERE} --region us-east-1 --payload $json /dev/stdout | head -n 1 | head -c -2 ; echo`  
  
echo ""  
echo "JSON RESULT:"  
echo $output | jq  
echo ""
```

Replace {INSERT API GATEWAY URL HERE} with your URL.
Save the script and provide execute permissions:

```
chmod u+x callservice.sh
```

Be sure to include the small quote mark at the end: `

This quote mark is next to the number 1 on US keyboards.

Next, locate the lines:

```
echo "Invoking Lambda function using AWS CLI"  
time output=`aws lambda invoke --invocation-type RequestResponse  
--function-name {INSERT AWS FUNCTION NAME HERE} --region us-east-1  
--payload $json /dev/stdout | head -n 1 | head -c -2 ; echo`
```

Replace {INSERT AWS FUNCTION NAME HERE} with your Lambda function name “hello”.

Before running this script, it is necessary to install some packages.

You should have curl installed from tutorial #2. If not, please install it:

```
sudo apt install curl
```

Next, install the AWS command line interface:

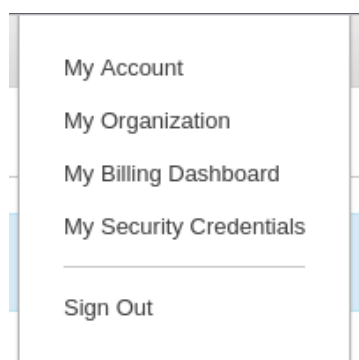
```
sudo apt install awscli
```

Next, configure the AWS CLI with your AWS account credentials:

You will need to acquire a AWS Access Key and an AWS Secret Access Key to use the AWS CLI.

In the far upper right-hand corner, locate your Name, and drop-down the menu.

Select “My Security Credentials”:

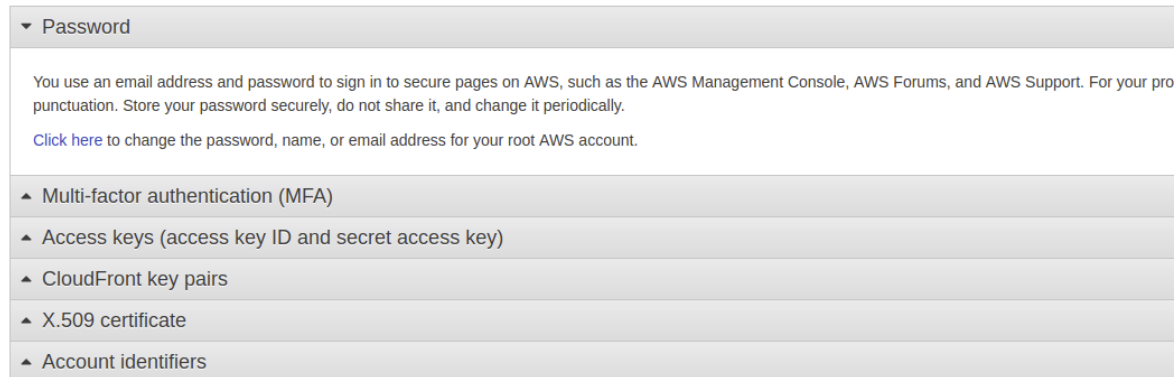


Then expand the menu option for “Access keys (access key ID and secret access key):

Your Security Credentials

Use this page to manage the credentials for your AWS account. To manage credentials for AWS Identity and Access Management (IAM) users, use the [IAM Console](#) .

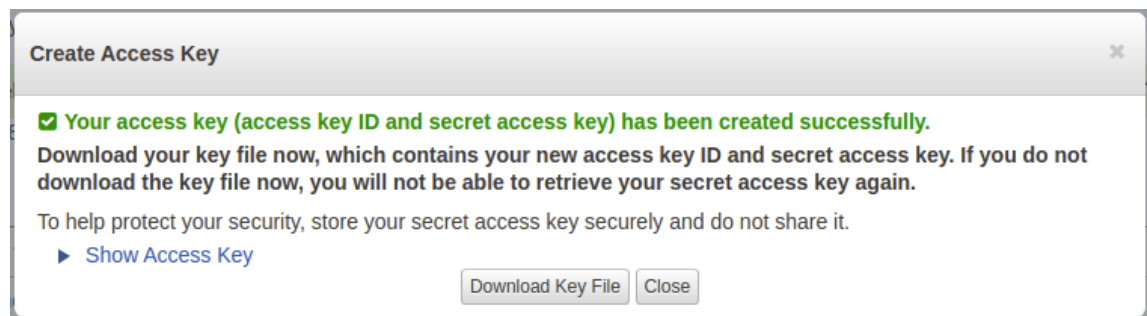
To learn more about the types of AWS credentials and how they're used, see [AWS Security Credentials](#) in AWS General Reference.



Click the blue button:

Create New Access Key

In the dialog, expand the “Show Access Key” section:



Copy and paste the Access Key ID and the Secret Access Key to a safe place:

Access Key ID:
Secret Access Key:

The values are blurred-out above.

Next configure your AWS CLI.

It is recommended to use the Virginia region (us-east-1).

```
$ aws configure
AWS Access Key ID [None]: <enter your access key>
AWS Secret Access Key [None]: <enter your secret key>
Default region name [None]: us-east-1
Default output format [None]:
```

This creates two hidden files at:
/home/ubuntu/.aws/config
/home/ubuntu/.aws/credentials

Use “ls -alt /home/ubuntu/.aws” to see them.

At any time, if needing to update the configuration, these files can be edited manually, or “aws configure” can be re-run. Amazon suggests changing the access key and secret access key every 90 days.

**NEVER UPLOAD YOUR ACCESS KEYS TO A GIT REPOSITORY.
AVOID HARD CODING THESE KEYS DIRECTLY IN SOURCE CODE WHERE FEASIBLE.**

Now install the “jq” package if you haven’t already from tutorial #2:

```
sudo apt install jq
```

7. Test your Lambda function using the API-Gateway and AWS CLI

It should now be possible to test your Lambda function using the callservice.sh script.

Run the script:

```
./callservice.sh
```

Output should be provided (abbreviated below):

```
Invoking Lambda function using API Gateway

real    0m3.622s
user    0m0.100s
sys     0m0.020s

Invoking Lambda function using AWS CLI

real    0m1.875s
user    0m0.524s
sys     0m0.096s

JSON RESULT:
{
  "cpuType": "Intel(R) Xeon(R) Processor @ 2.50GHz",
  "cpuNiceDelta": 0,
  "vmuptime": 1571198442,
  "cpuModel": "62",
```

```

"linuxVersion": "#1 SMP Wed Aug 7 22:41:25 UTC 2019",
"cpuSoftIrqDelta": 0,
"cpuUshrDelta": 0,
"uuid": "7b40cab1-5389-4667-8db9-3d703a982b18",
"platform": "AWS Lambda",
"contextSwitches": 20319,
"cpuKrn": 65,
"cpuIdleDelta": 1,
"cpuIowaitDelta": 0,
"newcontainer": 0,
"cpuNice": 0,
"lang": "java",
"cpuUshr": 93,
"majorPageFaultsDelta": 0,
"freeMemory": "458828",
"value": "Hello Susan Smith! This is from a response object!",
.....

```

The script calls Lambda twice. The first instance uses the API gateway. As a synchronous call the curl connection is limited to 29 seconds.

The second instance uses the AWS command line interface. This runtime is limited by the AWS Lambda function configuration. It can be set to a maximum of 15 minutes. The default is 15 seconds. **Both of these calls are performed synchronously to AWS Lambda.**

Function Deployment from the Command Line and Use of Availability Zones

SAAF provides a command line tool that automates deploying and updating FaaS functions to different cloud providers. Here, we demonstrate the use for the hello function for AWS Lambda.

Navigate to:

```
cd {base directory where project was cloned}/SAAF/java_template/deploy
```

Backup the config.json script:

```
cp config.json config.json.bak
```

Now modify config.json to deploy your hello function:

```

{
    "README": "See ./tools/README.md for help!",

    "functionName": "hello",

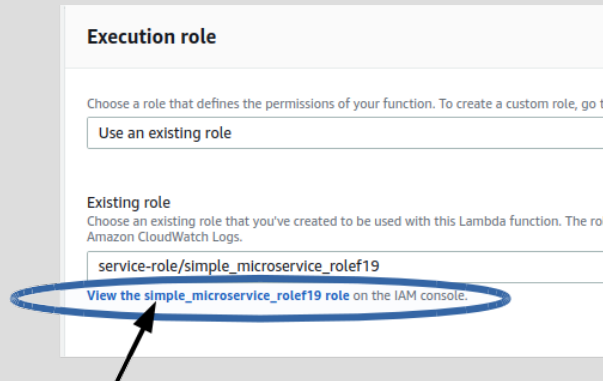
    "lambdaRoleARN": "arn:aws:iam:465394327572:role/service-role/simple_microservice_rolef19",
    "lambdaSubnets": "",
    "lambdaSecurityGroups": "",

    "test": {
        "name": "Bob"
    }
}

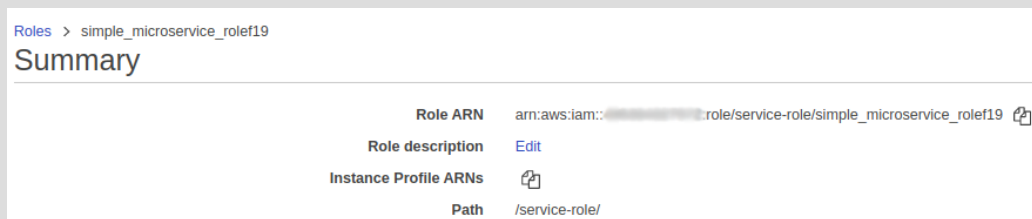
```

Function name: specify your function name 'hello'.

LambdaRoleARN: This is the Amazon Resource Name (ARN) for the Lambda Role previously created for the Lambda function. The ARN can be found by editing the Lambda function configuration in the AWS management console web GUI. Scroll down and locate the **Execution role**. Under existing role, click on the **blue link** that says “View the <role name> role”.



This opens the role for editing in the IAM console.



At the top of the Role Summary you'll see the Role ARN name. Click on the COPY icon to copy the ARN name to the clipboard. Paste this into your **config.json** file for the ARN.

The other attributes of note include **lambdaSubnets** and **lambdaSecurityGroups**. A subnet specifies a virtual network within a Virtual Private Cloud (VPC). Selecting a subnet allows the function to be deployed to a specific Availability Zone within an AWS Region. An availability zone is equivalent to a separate physical data center facility. These facilities are miles apart and considered physically separate locations.

The motivation to locate a Lambda function in an availability zone is to co-locate the function with other cloud resources that share the VPC. This way virtual machines and Lambda functions can be assigned to exist only in the same availability zone. This co-location reduces network latency as all network traffic is local. The network communication between resources does not have to leave the physical building.

To create a Lambda function in a VPC, the Execution Role must be modified to include the **AWSLambdaVPCAccessExecutionRole** policy. This policy can be added when copying the ARN name to setup **config.json**. Click on the blue button to attach a new policy:

Attach policies

Then search for “VPC” policies and select the policy by finding the policy and checkmarking it: **AWSLambdaVPCAccessExecutionRole**. Then press the blue button:

Attach policy

This will attach the policy to your Lambda Execution role. **This is required to deploy a Lambda function to a VPC.**

In the AWS Lambda function GUI, explore these options under **network**. To deploy a function to a VPC, first select the Default VPC. This enables the Subnets drop-down list. By default AWS has provided a subnet for each availability zone in the Region. These subnet IDs are what is added to **config.json** to deploy the Lambda function to a specific availability zone.

You may ignore the error message that says: “We recommend that you choose at least 2 subnets for Lambda to run functions in high availability mode.”

High availability is a great feature for production deployment.

For development, experimentation, performance testing, and research however, we’re interested in reproducing our results on the same hardware everyday. As a developer, how do you know if you’ve made your code faster if you constantly run it on different computers having CPUs running at different speeds? Choosing multiple zones increases the hardware heterogeneity of your Lambda function deployment and will increase performance variance.

Explore the GUI to write down subnet IDs and security group IDs for **config.json**:

Once you've configured `config.json` it's very easy to recompile and deploy your Lambda function using the command line. Simply run the script `publish.sh` with the arguments below. The last argument is the desired function memory size.

```
# Deploy to AWS with 3GBs:  
./publish.sh 1 0 0 0 3008
```

Additional documentation on the deploy tool can be found here:

https://github.com/wlloyduw/SAAF/tree/master/java_template/deploy

8. Parallel Client Testing of AWS Lambda

SAAF provides the “FaaS Runner” Python-based client tool for orchestrating multi-threaded concurrent client tests against FaaS function end points. “FaaS Runner” allows the function end points to be defined in JSON objects, and for the repeatable experiments to be defined as JSON objects.

Before starting, install dependencies for FaaS Runner:

```
sudo apt install python3 python3-pip  
pip3 install requests boto3 botocore
```

For detailed instructions on the FaaS Runner, please refer to the GitHub repository mark down documentation page:

FaaS Runner Documentation:

<https://github.com/wlloyduw/SAAF/tree/master/test>

There also exists a Bash-based client for performing multi-threaded concurrent tests that is available on request.

To tryout the FaaS Runner, navigate to the “test” directory:

```
cd {base directory where project was cloned}/SAAF/test
```

First, create a function JSON file under the `SAAF/test/functions` directory that describes your AWS Lambda function.

```
cd functions  
cp exampleFunction.json hello562.json
```

Edit the file `hello562.json` function file to specifically describe your Lambda function:

```
{  
  "function": "hello",  
  "platform": "AWS Lambda",  
  "source": "../java_template",  
  "endpoint": ""  
}
```

Function is the name of your AWS Lambda function.

Platform describes the FaaS platform where the function is deployed.

Source points to the source directory tree of the function.

Endpoint is used to specify a API Gateway URL.

If endpoint (URL) is left blank, the function can be invoked if the callWithCLI is set to true in the experiment file described below.

Next, create an experiment JSON file to describe your experiment again using the example template provided:

```
$ cd ..
$ cd experiments/
$ cp exampleExperiment.json hello562.json
```

Next edit the hello562.json experiment file to specifically describe your desired experiment using the hello function:

```
{
  "callWithCLI": true,
  "memorySettings": [0],
  "payloads": [
    { "name": "Bob" },
    { "name": "Joe" },
    { "name": "Steve" }
  ],
  "runs": 50,
  "threads": 50,
  "iterations": 3,
  "sleepTime": 5,
  "randomSeed": 42,

  "outputGroups": ["uuid", "cpuType", "vmuptime", "newcontainer", "endpoint", "containerID", "vmID",
"zAll", "zTenancy[vmID]", "zTenancy[vmID[iteration]]"],
  "outputRawOfGroup": ["zTenancy[vmID[iteration]]", "zTenancy[vmID]", "cpuType"],
  "showAsList": ["vmuptime", "cpuType", "endpoint", "containerID", "vmID", "vmID[iteration]"],
  "showAsSum": ["newcontainer"],
  "ignoreFromAll": ["zAll", "lang", "version", "linuxVersion", "platform", "hostname"],
  "ignoreFromGroups": ["1_run_id", "2_thread_id", "cpuModel", "cpuIdle", "cpuIowait", "cpuIrq",
"cpuKrn", "cpuNice", "cpuSoftIrq", "cpuUsr", "finalCalc"],
  "ignoreByGroup": {
"containerID": ["containerID"],
"cpuType": ["cpuType"],
"vmID": ["vmID"],
  "zTenancy[vmID]": ["cpuType"],
  "zTenancy[vmID[iteration]]": ["cpuType"]
},

  "invalidators": {},
  "removeDuplicateContainers": false,
  "openCSV": true,
  "combineSheets": false,
  "warmupBuffer": 1
}
```

A detailed description of experiment configuration parameters is included on the GitHub page. Please modify the following:

Runs: This is the total number of function calls. **Set this to 100.**

Threads: This is the total number of threads used to invoke the **Runs**. **Set this to 100.** Keeping a 1 : 1 ratio between runs and threads ensures that each run will be performed by the client in parallel using a dedicated thread.

Iterations: This is number of times the experiment will be repeated. **Set this to 1.**

openCSV: If your platform has a spreadsheet application that will automatically open CSV files, then specify true, otherwise specify false. (Linux or MAC only)

CombineSheets: When set to true, this will combine multiple **iterations** into one spreadsheet. Since we are only performing 1 iteration, set this to 0.

Important Note about FaaS performing testing: If the client is an underpowered computer, (e.g. older laptop with 2 CPU cores and limited memory/network), or if having a slow network connection the client may fail to invoke 100 concurrent FaaS function calls in parallel at the same time. This results in infrastructure reuse on the FaaS platform. Containers and/or VMs will be reused in a single “experiment” by the cloud provider. If the goal is to force the cloud provider into creating unique and dedicate infrastructure for 100 concurrent calls, a more powerful client computer may be needed. As a workaround, the FaaS function can be made more complex so that it runs longer. This is as simple as adding sleep time in your Java code:

```
// Sleep for 10 seconds  
Thread.sleep(10000);
```

Now try the FaaS Runner python tool.

Before trying the tool, be sure to close any spreadsheets that may be open in Microsoft Excel or Open/LibreOffice Calc from previous SAAF experiment runs.

```
# navigate back to the test directory  
cd {base directory where project was cloned}/SAAF/test  
  
# Requires python3  
python3 faas_runner.py -f functions/hello562.json -e experiments/hello562.json
```

If your platform has a spreadsheet or tool configured to automatically open CSV files, then the CSV file may automatically open once it is created. It is important that only the comma (“,”) be used as a field/column delimiter.

Explore the CSV output using a spreadsheet application to determine the following.

Answer these questions and write the answers in a PDF file to upload to Canvas.

Include your Name, Function Name, AWS Region, VPC (+ Availability Zone), or no VPC

0. Did you add Thread.sleep(10000) ? Yes / No
1. The total number of “Successful Runs”
2. The total number of unique container IDs
3. The total number of unique VM IDs
4. The number of runs with newcontainer=0 (these are recycled runtime environments)

5. The number of runs with newcontainer=1 (these are newly created runtime environments)
6. The zAll row aggregates performance results for all tests. Looking at this row, what is the:
- avg_runtime for your function calls? (measured on the server side)
 - avg_roundTripTime for your function calls? (measured from the client side)
 - avg_cpudelta for your function calls? (units are in centiseconds)

cpudelta time is measured in centiseconds. Multiply this by 10 to obtain milliseconds. Try adding "Thread.sleep(10000)" to your hello function. Linux time accounting is possible using SAAF. Simply add up the available CPU metric deltas and divide by the number of CPU cores (2 for AWS Lambda) to obtain an estimate of the wall clock time. With Thread.sleep(10000) this value should be close to 10,000.

Difference Between AWS Lambda VPC and NO VPC function deployments:

Deploying a Lambda function into a Virtual Private Cloud is almost like deploying the function to an entirely different platform. VPC Lambda functions are backed by what we suspect are traditional XEN-based virtual machines similar to EC2 instances. These VMs appear to host multiple "containers" or function instances. When deploying Lambda functions to a VPC, you may see different types of CPUs resulting in different function runtimes. SAAF will group by the CPU type and calculate the average runtime for each CPU.

When deploying a Lambda function without a VPC, these functions receive their own micro-VMs. AWS has announced the "Firecracker" MicroVM specifically for serverless (FaaS and CaaS) workloads. MicroVMs provide better isolation from a resource accounting point of view. Our view of the underlying hardware is more abstracted with no VPC making it more difficult to infer the cause of performance variance. CPUs on Firecracker are simply identified as: **Intel(R) Xeon(R) Processor @ 2.50GHz**. No model number is specified. This may be a virtual CPU designation provided by the hypervisor.

Firecracker MicroVM:

<https://firecracker-microvm.github.io/>

The FaaS Runner will store experiment results as CSV files under the history directory.

On some platforms, these filenames may automatically increment so they don't overwrite each other. On other platforms, it may be necessary to make a copy to preserve the files between runs.

Here is an example of making a copy:

```
cd history
cp "hello - hello562 - 0 - 0.csv" tcss562_ex1.csv
```

9. Two-Function Serverless Application: Simple Caesar Cipher

To complete tutorial #4, use the resources provided to construct a two-function serverless application.

To get started, create a new directory under /home/ubuntu
Then clone the SAAF repository twice to have two separate empty Lambdas:

```
$ cd ~
:~$ mkdir tcss562
:~$ cd tcss562
:~/tcss562$ mkdir encode
:~/tcss562$ mkdir decode
:~/tcss562$ cd encode
:~/tcss562/encode$ git clone https://github.com/wlloyduw/SAAF.git
Cloning into 'SAAF'.....
:~/tcss562/encode$ cd ..
:~/tcss562$ cd decode
:~/tcss562/decode$ git clone https://github.com/wlloyduw/SAAF.git
Cloning into 'SAAF'.....
```

Next, implement two lambda functions.

One called “Encode”, and another “Decode” that implement the simple Caesar cipher.

In the SAAF template, the verbosity level of metrics can be adjusted to provide less output.

To explore verbosity levels offered by SAAF, try adjusting the number of metrics that are returned by replacing the line of code:

```
inspector.inspectAll();
```

with one of the following or simply remove inspectAll() altogether:

inspectCPU()	reports all CPU metrics
inspectContainer()	reports all Container-level metrics (e.g. metrics from the runtime environment)
inspectLinux()	reports the version of the Linux kernel hosting the function.
InspectMemory()	reports memory metrics.
InspectPlatform()	reports platform metrics.

At the bottom, the following line of code can be commented out or replaced:

```
inspector.inspectAllDeltas();
```

Less verbose options include:

inspectCPUDelta()	Reports only CPU metric changes
-------------------	---------------------------------

inspectMemoryDelta()	Reports only memory metric utilization changes
----------------------	--

Detailed information about metrics collection by SAAF is described here:

https://github.com/wlloydw/SAAF/tree/master/java_template

For the Caesar Cipher, pass a message as a JSON object to “encode” as follows:

```
{
  "msg": "ServerlessComputingWithFaaS",
  "shift": 22,
}
```

Encode shifts the letters of an ASCII string forward to disguise the contents as follows in the JSON output (SAAF metrics mostly removed):

```
{
  "value": "OanranhaooYkilqpejcSepdBwwO",
  "uuid": "036c9df1-4a1d-4993-bb69-f9fd0ab29816",
  "error": "",
  "vmuptime": 1539943078,
  "newcontainer": 0
}
```

The second service shifts the letters back to decode the contents as shown in the JSON output:

```
{
  "value": "ServerlessComputingWithFaaS",
  "uuid": "f047b513-e611-4cac-8370-713fb2771db4",
  "error": "",
  "vmuptime": 1539943078,
  "newcontainer": 0
}
```

Notice that the two services have different uuids (container IDs) but the same vmuptime (VM/host ID). On AWS Lambda + VPC this behavior could occur that two functions share the same VMs.

Both services should accept two inputs:

integer	shift	number of characters to shift
String	msg	ASCII text message

The Internet has many examples of implementing the Caesar cipher in Java:

<https://stackoverflow.com/questions/21412148/simple-caesar-cipher-in-java>

Once implementing and deploying the two-function Caesar cipher Lambda application, modify the `call_service.sh` script and create a “`cipher_client.sh`” BASH script to serve as the client to test your two-function app.

`Cipher_client.sh` should create a JSON object to pass to the encode service. The output should be captured, parsed with `jq`, and sent to the decode service.

The result should be a simple pair of services for applying and removing the cipher. The `Cipher_client.sh` bash script acts as the client program that instruments the flow control of the two-function cipher application. Deploy all functions to operate synchronously just like the hello example service. Host functions in your account to support testing.

Use API gateway endpoints and `curl` to implement `Cipher_client.sh`. Do not use the AWS CLI to invoke Lambda functions. This will allow your two-function application to be tested using the `Cipher_client.sh` script that is submitted on Canvas.

SUBMISSION

Tutorial #4 should be completely individually. Files will be submitted online using Canvas.

For the submission, submit a working bash client script (**`Cipher_client.sh`**) that invokes both functions. SAAF verbosity may be optionally reduced to simplify the output.

Also include in the Canvas submission a zip or tar.gz file that includes all source code for your Lambda functions.

Also include a PDF file including answers to questions for #8.

Scoring

20 points	Providing a PDF file answering questions using output from the FaaS Runner for #8.
20 points	Providing a working <code>Cipher_client.sh</code> that instruments the two-functions Lambda app.