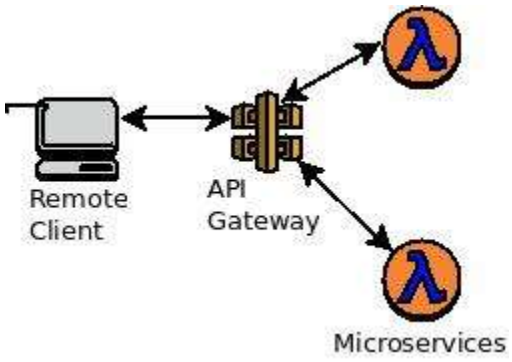
TCSS 462/562: (Software Engineering for) Cloud Computing
Fall 2022

School of Engineering and Technology
University of Washington – Tacoma

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

Disclaimer: Subject to updates as corrections are found
Version 0.11
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, plus do so.

On ubuntu see the official documentation:
https://help.ubuntu.com/lts/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 #4, 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 Apace Netbeans 15. Many students prefer using Microsoft Visual Studio Code with the "Extension Pack for Java". Other options include Eclipse, and IntelliJ.

Download Netbeans 15 (NB 15.0) Installer for your platform:
https://netbeans.apache.org/download/nb15/index.html

To install on Ubuntu, the netbeans snap package can be easily installed. (*this may take a while*)
For troubleshooting see: https://snapcraft.io/install/netbeans/ubuntu

```
sudo snap install netbeans –classic
```

Alternatively, Microsoft Visual Studio Code can be used.
For Ubuntu 22.04 installation documentation see: https://linuxiac.com/install-visual-studio-code-on-ubuntu-22-04/

Once Visual Studio Code is installed, install the **Extension Pack for Java**:
https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack

In addition, you may want (or need) to install additional plug-ins and extension packs to customize VS Code.

While the Netbeans installation may (or may not) automatically install the Java 11 Java Development Kit (JDK), it is necessary to install the Java 11 JDK for Visual Studio Code:

In Ubuntu, the Java 11 JDK can be installed with:

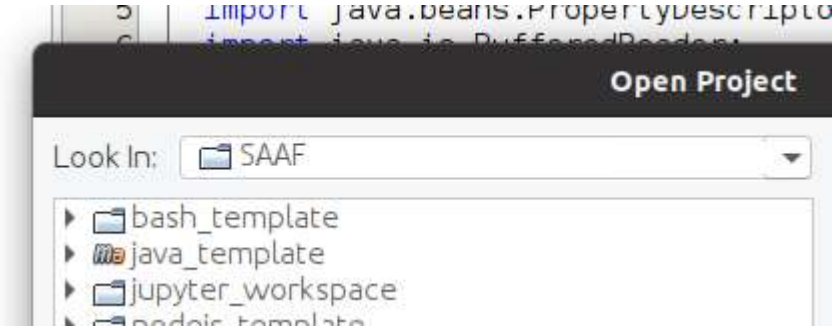
```
sudo apt install default-jdk
# After installing, verify the version of the java compiler:
javac -version
```

If working with Visual Studio Code, if the proper extensions are installed, you should be able to Open the Folder of the SAAF git project. Under the File menu, select "Open Folder" and navigate to the location where you

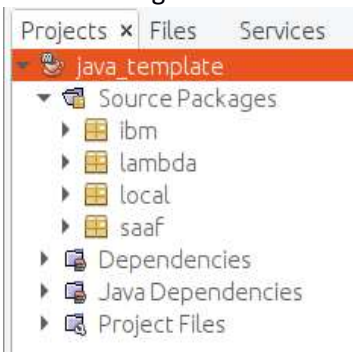
clone the SAAF git hub repository, and select the SAAF/java_template directory. If Visual Studio Code is properly configured, it will scan your project, and recognize the Java project has a Maven build file. Then, in the lower left hand corner there should be a Maven menu listed. By opening the Maven menu and right-clicking on the java_template, there should be an option for "package". If you select "package" it will build the Java JAR file that is required for deployment to AWS Lambda.

For Netbeans, once you've downloaded the IDE, you'll be able to open the project directly without install any plugins.

Select "File | Open Project", and navigate to where you have cloned the SAAF git project. Open the "SAAF" folder and then select "**java_template**":



Then on the left-hand side, expand the "Source Packages" folder:



You'll see a "lambda" package icon. Expand this.

This is where you'll find the source code for a Hello World Lambda function that is provided as a starter template.

You will see four relevant class files:

Hello.java Provides an example implementation of a Java AWS Lambda Function. The handleRequest() 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. Hello.java uses a HashMap for the request and response JSON objects. The incoming response JSON is "serialized" into a hashmap automatically by Lambda. The outgoing response JSON is created based on the HashMap that is returned.

HelloPOJO.java HelloPOJO.java is the same as Hello.java except that instead of using a HashMap for the Request (incoming) data, instead an explicitly defined Request class is defined with getter and setter methods to accept input from the user. The advantage with HelloPOJO is the Request object can perform post-processing on input parameters provided from the function caller before they are used. Post-processing includes operations such as formating 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 HelloMain.java is identical to Hello.java except that it also contains a public static void main() 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 main() method to include required parameters for ineracting with the function. The main() method creates a mock Context() 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 Response class POJO. This has been removed in favor of simply using a HashMap. A Response POJO could be implemented alternatively to add logic to getter and setter methods to perform data formatting, transformation, or validation operations.

In Netbeans, if you see exclamation marks on the source file icons, where the exclamation mark is on every file, this indicates that the Java Platform for the IDE is not properly configured. Close the project by clicking on "java_template" and selecting "Close".

Then, under Tools | Java Platforms, you will need to add an entry to point to JDK 11 on your system. JDK 11 is typically installed at: **/usr/lib/jvm/default-jvm**. Then under Tools | Options under the Java tab, configure your newly provided Java Platform under "Java Shell" and "Maven" and save the settings. Then, reopen the project and check if the exclamation mark is resolved.

If you wish to install additional versions of Java such as Java 8, this is also possible. You can configure multiple Java version in NetBeans under Tools | Java Platforms.

On the Ubuntu command line, it is also possible to install multiple versions of Java. This enables working directly from the CLI to compile projects etc. For this, check the version of Java currently used. Make sure to

match the version of functions to be deployed on AWS Lambda. To check which Java versions are installed, use the following command, but do not select a version, just press ENTER when prompted:

```
sudo update-alternatives --config java
```

In most cases, you may have just one version installed, but it is possible to install many versions of JAVA, and then switch between them using the "sudo update-alternatives" command. **This sets the version of Java in the command-line environment.** This is different than the version of Java that is configured for the Netbeans project.

To inspect the version of Java used in your project in Netbeans, in the project explorer on the left-hand side, right-click on the project name, and select "Properties" at the bottom of the list. First, under the "Build" option, select "Compile", and in the dialog box select the proper Java Platform, such as JDK 11. After setting the Java Platform, select "Sources" and in the dialog box set the Source/Binary format to, for example "11". Note, if wanting to build Java 8, you'll select different options.

Now compile the project using maven from the IDE:

From the NetBeans IDE right click on the name of the project "java_template" in the left-hand list of Projects and click "Clean and Build".

Now try compiling directly from the command line, under the "SAAF/java_template" directory:

```
cd {base directory where project was cloned}/SAAF/java_template/
# Clean and remove old build artifacts
mvn clean -f pom.xml
```

Then to build the project jar file:

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

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, cpuUsrDelta=1, uuid=e5faf33b-154b-
4224-bb45-2904abfb9897, platform=Unknown Platform, contextSwitches=3034407068,
cpuKrn=9920122, cpuIdleDelta=7, cpuIowaitDelta=0, newcontainer=0, cpuNice=33510,
lang=java, cpuUsr=19443782, majorPageFaultsDelta=0, freeMemory=1743632,
value=Hello Susan! This is from a response object!, frameworkRuntime=61,
```

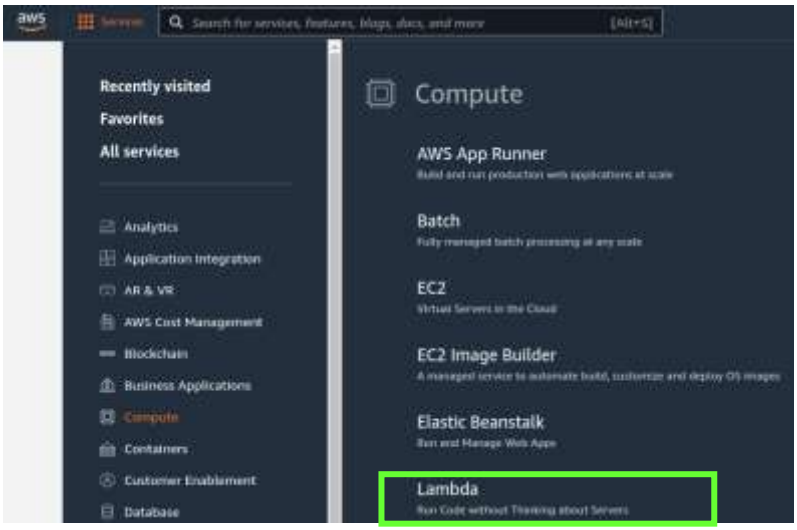
```
contextSwitchesDelta=133, vmcpusteal=0, cpuKrnDelta=0, cpuIdle=32013339,
runtime=73, message=Hello Susan! This is a custom attribute added as output from
SAAF!, version=0.31, cpuIrqDelta=0, pageFaultsDelta=324, cpuIrq=0,
totalMemory=32318976, cpuCores=4, cpuSoftIrq=60350, cpuIowait=582306,
majorPageFaults=11984, vmcpustealDelta=0, pageFaults=953729377, userRuntime=11}
```

Whoa! That's a lot of output! The actual Lambda function output is highlighted.
Other values represent data collected by the SAAF profiling framework. Of course, since you're testing locally, this data is for your local Linux environment, not the cloud.

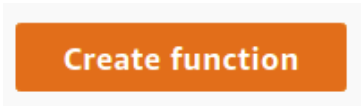
4. Deploy the function to AWS Lambda

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

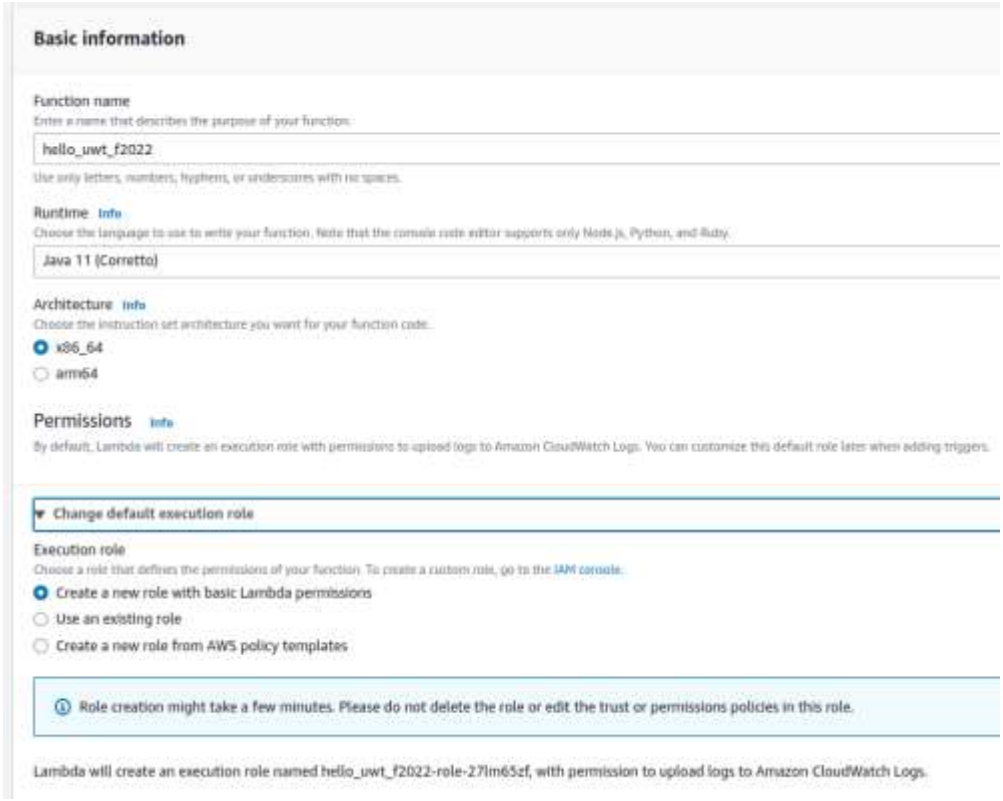
Log into your AWS account, and under "services" locate "Lambda" by searching, or selecting "Compute":



Click the button to create a new Function:

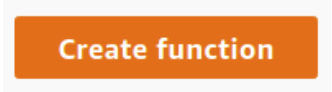


Using the wizard, use the "Author from scratch" mode.
Next provide the following values:

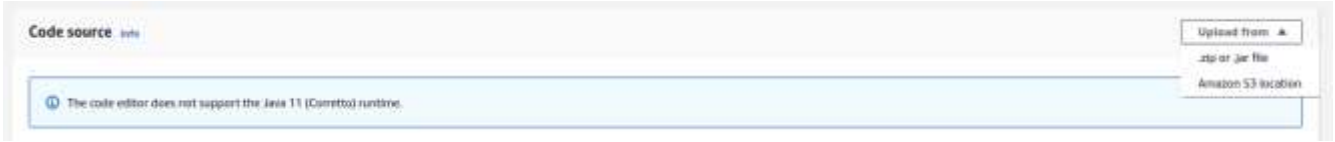


Function name: hello (choose a unique name of your choice)
Runtime: Java 11 (Corretto)
Execution Role: "Create a new role with basic Lambda permissions"
(Additional policies and permissions can be added to this role if needed.)
(Roles can be inspected under IAM | Roles in the AWS Management Console)

Once filling the form, click the button:



Next, upload your compiled Java JAR file to AWS Lambda. Under "Code Source", select "Upload from" and ".zip or .jar file".



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, scroll down to "Runtime settings". Click **Edit**, and in the dialog box change the "Handler" to:

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

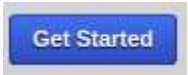
***** IF THE HANDLER IS NOT UPDATED, LAMBDA WILL NOT BE ABLE TO LOCATE THE ENTRY POINT TO YOUR CODE. THE LAMBDA FUNCTION WILL FAIL TO RUN *****

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, but using the search bar may be the fastest way:

The very first time you visit the API Gateway console screen, there ***may*** be a "splash" screen. IF so, select the button:

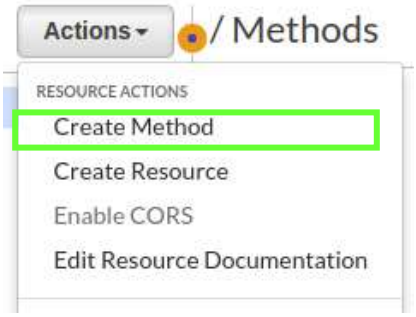


Next choose an API type. We will select **"REST API"** and click **Build:**

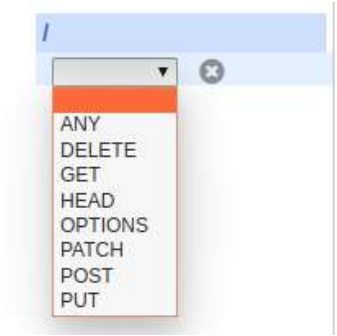
Select "Build", keep defaults, and specify these settings:
<*> REST
<*> NEW API
API Name: hello_uwt
Description: <can leave blank>
Endpoint Type: choose Regional

Press the BLUE "Create API" button.

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

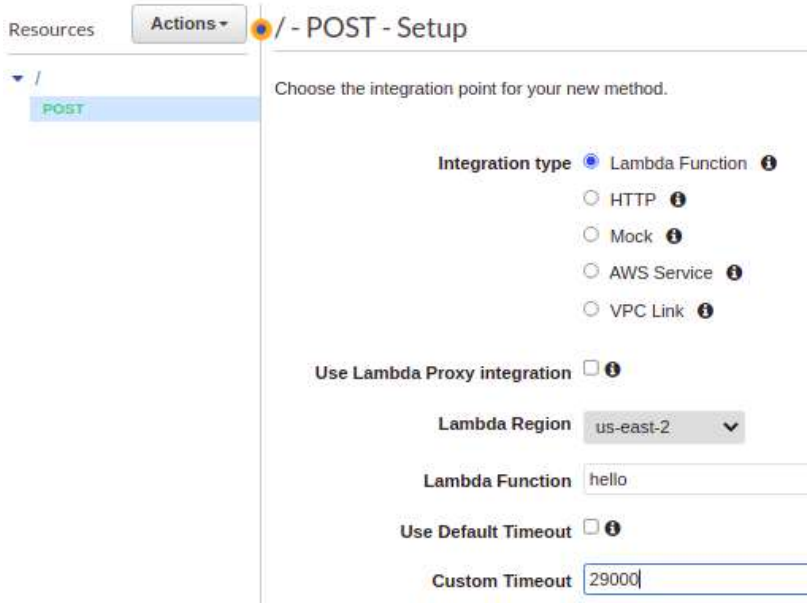


Select, drop down the menu and select “Post”:



Next, press the “checkmark” icon so the POST text turns green.

Then complete the form:



Fill in “Lambda Function” to match your function name “hello” or whatever name you have used.

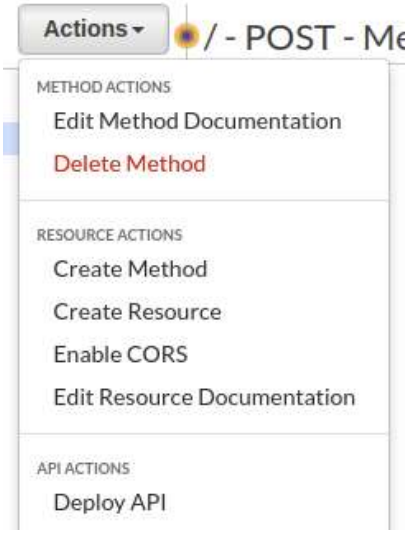
Uncheck the “Use Default Timeout”.

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” milliseconds.

Then click Save:

Next, acknowledge the permission popup:

Then, select the Actions drop-down and select “**Deploy API**” at the bottom:



Next complete the form. Select the “Deployment stage”, and assign the Stage name:

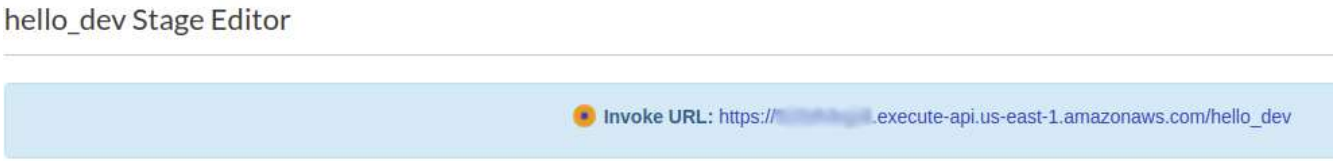
When complete, press the blue [Deploy] button.

The API-Gateway allows the configuration of RESTful web service Uniform Resource Identifiers (URIs) to associate RESTful endpoints with function backends that clients can invoke just as done in Tutorial #2. The API-Gateway is not limited to AWS Lambda functions. It can also point to other services hosted by AWS. When configuring a URI for a RESTful webservice, the AWS hosted URL acts as a proxy that routes incoming traffic to your configured backend, in this case the Hello Lambda function. This enables traffic to be routed using a URL through the API-Gateway, where the API-Gateway as an intermediary can introduce logging and/or additional processing.

Using the API-Gateway it is also possible to host multiple versions of a function to support Agile software development processes. An organization may want to maintain multiple LIVE versions of a function in various stages of development such as: (dev)evelopment, test, staging, and (prod)uction

A **Stage Editor** should appear with a REST URL to your AWS Lambda function.
The Stage name you provided is appended to the end of the Invoke URL.

COPY THE INVOKE URL TO THE CLIPBOARD:
Mouse over the URL, -right-click- and select “Copy link address”:



Use this URL in the callservice.sh test script below.

6. Install package dependencies and configure your client to call Lambda

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” as follows:

```
#!/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 ""
echo "JSON RESULT:"
echo $output | jq
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-2 --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.
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-2 --payload $json /dev/stdout |
head -n 1 | head -c -2 ; echo`
```

Replace {INSERT AWS FUNCTION NAME HERE} with your Lambda function name “hello” (or whatever name you have used for your function).

Save the script and then provide execute permissions:

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

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 (CLI) (***this should have been completed previously for Tutorial 0, but if not, do it now)***:

```
sudo apt install awscli
```

Please refer to Tutorial 0 for detailed instructions for configuring the AWS CLI.

Note that running '**aws configure**' in Tutorial 0 will create 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,
  "cpuUsrDelta": 0,
  "uuid": "7b40cab1-5389-4667-8db9-3d703a982b18",
  "platform": "AWS Lambda",
  "contextSwitches": 20319,
  "cpuKrn": 65,
  "cpuIdleDelta": 1,
```

```
  "cpuIowaitDelta": 0,
  "newcontainer": 0,
  "cpuNice": 0,
  "lang": "java",
  "cpuUsr": 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.**

7B. (NEW) AWS Lambda Function URLs - BONUS

As of April 2022, AWS introduced Function URLs as an alternative to creating http REST endpoints using the API Gateway. This provides a faster way to create a REST URL that is associated with a Lambda function with one caveat (issue). For Function URLs, the Request object that is provided by curl is nested under the **body** tag. This means that the function source code must first read the body tag/value pair, and then convert the value of the body tag into a JSON object or Python dictionary. This adds an extra step, and makes code deployed using the API Gateway (or invoked using the AWS CLI) not directly compatible with code using Function URLs.

You can read about this feature in the April 2022 press release:
https://aws.amazon.com/blogs/aws/announcing-aws-lambda-function-urls-built-in-https-endpoints-for-single-function-microservices/

To create a Function URL for your Lambda function, from the AWS Lambda GUI, click on the **Configuration** tab. On the left-hand side select **Function URL**. Creating a Function URL is then a simple matter of clicking the **Create function URL** button.

For the 'Auth type' it will be easiest to use NONE. Function permissions can be restricted using advanced security capabilities by configuring specific IAM users and roles to have access. Additionally cross-origin resource sharing (CORS) can be used to restrict access further. Press **Save** to then create the Function URL.

As a bonus activity, convert your hello function to use a Function URL. To extract the name parameter, you will need to parse the content of the "body" tag/value pair. This is where the JSON object will appear. It is no longer possible to simply extract the "name" tag/value pair as it is embedded under "body". You can parse the

string directly, or you can try to convert it to JSON to extract the value of the 'name' attribute using a JSON library in Java. This task is easier in Python as a Python dictionary can be used.

As an interesting benchmarking task, try comparing the function roundtrip time when invoking a function using an API Gateway HTTP REST endpoint, a function URL, and the AWS CLI from **callservice.sh**. Is there any difference in the round-trip time? Is one of the invocation methods more efficient?

Optional: 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:

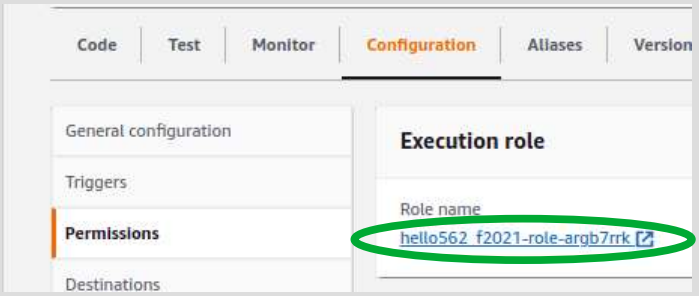
```
{
        "README": "See ./tools/README.md for help!",

        "functionName":     "hello",

        "lambdaHandler": "lambda.Hello::handleRequest",
        "lambdaRoleARN": "arn:aws:iam::465394327572:role/service-role/simple_microservice_rolef19",
        "lambdaSubnets": "",
        "lambdaSecurityGroups": "",
        "lambdaEnvironment": "Variables={EXAMPLEVAR1=VAL1,EXAMPLEVAR2=VAL2}",
                "ibmHandler": "ibm.Hello",

        "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. Under the **Configuration** tab, select **Permissions** on the left. Where it says **Role name**, click on the link and open the role under Identity Access Management.



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** on the RIGHT 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.

<OPTIONAL - VPC Setup – will be reviewed again in Tutorial #6> 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:

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

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

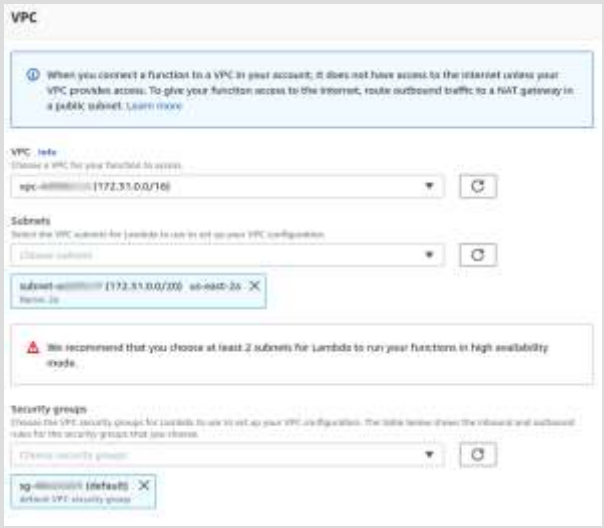
In the AWS Lambda function GUI, explore these options under **VPC**. Scroll down to "VPC" and press EDIT. To deploy a function to a VPC, first select the "Default VPC". This enables the Subnets drop-down list. By default AWS has provided a subnet for each availability zone in the Region. These subnet 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 may increase runtime variation.

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. The 0s are for deployment to other FaaS platforms which are not used in this tutorial: Google Cloud Functions, IBM Cloud Functions, and Azure Functions.

Note: if you're deploying w/ Java 11, it is necessary to search and replace in the "publish.sh" script and update "java8" with "java11". There are two locations in the publish script to update. This will be added as a JSON parameter soon.

```
# 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. TO DO: 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
```

We I did this, it broke my aws cli. Test your aws cli version to be sure the aws cli is still working:

```
aws --version
```

If the CLI, stops working, reinstall the following:

```
pip3 install awscli==1.19.53
pip3 install botocore==1.22.0
```

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 hello.json
```

Edit the file hello.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 hello.json
```

Next edit the hello.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. Keep this set to 50. (this is the default concurrency limit)
Threads: This is the total number of threads used to invoke the **Runs**. Keep this set this to 50. 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 **false**.

To obtain 50 distinct execution environments on AWS Lambda (think sandboxes), on remote network connections It is necessary to add a sleep call in the function so that the client computer can concurrently invoke 50 functions to run in parallel. Without adding a sleep function, AWS Lambda is so fast that many of the functions will complete preventing the client computer from successfully invoking 50 functions in parallel. Instead some function environments (instances) will be reused. When function executions do not overlap in time existing function environments (think sandboxes) will be reused resulting in (**newcontainer=0**). When all

function invocations overlap this forces AWS Lambda to create and run 50 distinct sandboxes at the same time. This create resource contention in the public cloud because the function instances will compete for resources across a set of cloud servers. Given that HelloWorld is not a computationally complex function, overlapping calls requires adding the sleep statement to extend the duration of execution to force an overlap. These functions, however, don't compete for resources because they just "sleep".

Try adding a sleep statement to force the cloud provider to create 50 distinct execution environments (i.e. sandboxes) for running your HelloWorld function at the same time. Success will be indicated by obtaining 50 functions with newcontainer=1. After sandboxes are created, they are reused on subsequent calls, so they report newcontainer=0. Function instances (e.g. sandboxes) are deprovisioned randomly by AWS Lambda starting approximately 5 minutes after the last function call. In experiments, previously deprovisioning 100 sandboxes has been shown to take from 10 to 40 minutes as the sandboxes are slowly retired by AWS.

Add a sleep function to overlap the execution of your functions on AWS Lambda to obtain 50 new containers:

```
// Sleep for 10 seconds
try
{
  Thread.sleep(10000);
}
catch (InterruptedException ie)
{
  System.out.println("Interruption occurred while sleeping...");
}
```

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/hello.json -e experiments/hello.json
```

Note that faas_runner will automatically use the default AWS region that has been configured using the '**aws configure**' command. If for some reason you need to change your default region, please rerun '**aws configure**'.

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.

>>> FOR SUBMISSION <<<

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. Report the total number of "Successful Runs"

2. Report the total number of unique container IDs
3. Report the total number of unique VM IDs
4. Report the number of runs with newcontainer=0 (these are recycled runtime environments)
5. Report 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, report the:
- **avg_runtime** for your function calls (measured on the server side in milliseconds)
- **avg_roundTripTime** for your function calls (measured from the client side in milliseconds)
- **avg_cpuIdleDelta** for your function calls (units are in centiseconds)
- **avg_latency** for your function calls (in milliseconds)

The difference between the avg_roundTripTime and the avg_runtime should be the avg_latency.

cpuIdle time is measured in centiseconds. Multiply this by 10 to obtain milliseconds.
Linux CPU time accounting is provided in SAAF to report the state of the processor when executing Lambda functions. The wall clock (or watch time) can be derived by adding up the available CPU metric deltas and dividing by the number of CPU cores (2 for AWS Lambda @ 3GB RAM) to obtain an estimate of the wall clock time (function runtime).

Once adding "Thread.sleep(10000)" to your hello function check the delta value for CPU IDLE time. By including Thread.sleep(10000) this value should be close to 10,000. Sleep essentially makes the CPU idle for most of the duration of the function's runtime.

Difference Between AWS Lambda VPC and NO VPC function deployments:
Lambda functions that run in a Virtual Private Cloud (VPC) suffer from additional cold start overhead because when function instances are first called, there is a higher initialization cost to setup the VPC network connection for the function compared to standard non-VPC Lambda functions.

AWS Lambda Abstracts CPU type through the use of Micro VMs:
When executing Lambda functions, for each concurrent client request arriving at the same time, Lambda creates distinct virtual infrastructure known as "function instances". One function instance is created for each user request received in parallel (at the same time). Function instances can be reused on subsequent calls. After 5 minutes of inactivity, function instances are gradually deleted. When function instances are continually used, they can stay alive for up to ~4.5 hours. Amazon will automatically replace function instances to continually refresh infrastructure at random. Function instances are implemented using micro-VMs which are a lighter-weight form of a full virtual machine. AWS has created the "Firecracker" MicroVM specifically for serverless (FaaS and CaaS) workloads. MicroVMs provide better isolation from a resource accounting point of view. Using these micro-VMs, however, has led to further abstraction of the underlying hardware. For example the CPU type on Firecracker VMs 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 Firecracker which is based on KVM.

To read more about the Firecracker MicroVM, see:
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 - hello - 0 - 0.csv" tcss462-562_ex1.csv
```

9. TO DO: Two-Function Serverless Application: Caesar Cipher

To complete tutorial #4, use the resources provided to construct a two-function serverless application that implements a Caesar Cipher. The Caesar cipher shifts an ASCII string forward to encode the message, and shifts the string backwards to decode.

To get started, create a new directory under /home/ubuntu
Then clone the SAAF repository twice to have two separate empty Lambdas.
Alternatively, a single project can be used where there are separate encode and decode class files. The function handler can be adjusted to point to the specific class and/or method that serves as the Lambda function entry point to your Java code.

```
$ 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 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/wlloyduw/SAAF/tree/master/java_template
For the Caesar Cipher, pass a message as JSON to your "encode" function as follows:

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

The encode function should shift the letters of an ASCII string forward to disguise the contents as shown in the example JSON below (SAAF metrics mostly removed):

```
{
      "msg": "OanranhaooYkilqpejcSepdBwwO",
      "uuid": "036c9df1-4a1d-4993-bb69-f9fd0ab29816",
      "vmuptime": 1539943078,
      "newcontainer": 0
      . . .  output from SAAF truncated for brevity...
}
```

The second service, decrypt, should shift the letters back to decode the contents as shown in the JSON output:

```
{
      "msg": "ServerlessComputingWithFaaS",
      "uuid": "f047b513-e611-4cac-8370-713fb2771db4",
      "vmuptime": 1539943078,
      "newcontainer": 0
      . . .  output from SAAF truncated for brevity...
}
```

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 if two functions share the same VMs. Note: *This behavior is no longer observable as AWS Lambda now uses the Firecracker MicroVM for hosting function which abstracts this information about shared hosts from users.*

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

You'll notice that SAAF provides a lot of attributes in the JSON output. This verbosity may be optionally reduced to simplify the output. Instead of calling **inspectAll()** the code can be reworked to call a few functions that will then only provide a subset of the information. For example, this set would offer fewer attributes while retaining some helpful metrics:

```
inspector.inspectCPUDelta();
inspector.inspectContainer();
inspector.inspectPlatform();
```

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.

When possible please create and submit a Linux tar.gz file to capture all of your project's source files. From the command line, navigate to the SAAF directory for your encode/decode project. You may combine functions into a single project (*by modifying the function handler when deploying the Lambdas- recommended)* or submit separate tar.gz files for a separate encode and decode project to Canvas.

To create the tar.gz archive file, from the SAAF directory, use the command:

```
tar czf encode.tar.gz .
```

Once having the archive, the contents can be inspected as follows:

```
tar ztf encode.tar.gz | less
```

Use the 'f' key to go forward, 'b' key to go backward, and 'q' key to quit

For the submission, submit a working bash client script **(Cipher_client.sh)** that invokes both functions.

Be sure to include in the Canvas submission the tar.gz file that includes all source code for your Lambda functions. Alternatively a zip file can be submitted.

In addition, 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 Cipher_client.sh that instruments the two-functions Lambda app using REST URLs from the API Gateway.