TCSS 462/562: (Software Engineering for) Cloud Computing
Fall 2024

School of Engineering and Technology
University of Washington – Tacoma

Tutorial 9 – Serverless Beyond Java, Container-Based Functions

Disclaimer: Subject to updates as corrections are found
Version 0.12

The purpose of tutorial #9 is to provide tools and resources to support working with serverless functions in Python, and Container-Based serverless functions in any language with profiling support provided by SAAF introduced in Tutorial 4. This is an extra credit tutorial. Points are available for two separate tasks. The two tasks are scored separately.

Task 1 – Deploy and Test Python-based Hello Function on AWS Lambda (10 points)

1. Download SAAF

To begin, using git, clone SAAF.
Create a new directory for python-hello

```
mkdir python-hello
cd python-hello
git clone https://github.com/wlloyduw/SAAF.git
```

2. Build Python Function

Next, inspect the python function handler's code.
This is conceptually the same as the Hello.java class's handleRequest method in the lambda package in Java.

```
cd python_template/src
cat handler.py
```

Inspect the code. No code changes are required.

Next, build a zip file to deploy the Python code as an AWS Lambda function.
Before building a zip file, copy the AWS Lambda platform specific handler wrapper:

```
cp ../platforms/aws/lambda_function.py .
zip -X -r ./index.zip *
```

3. Deploy Python Function

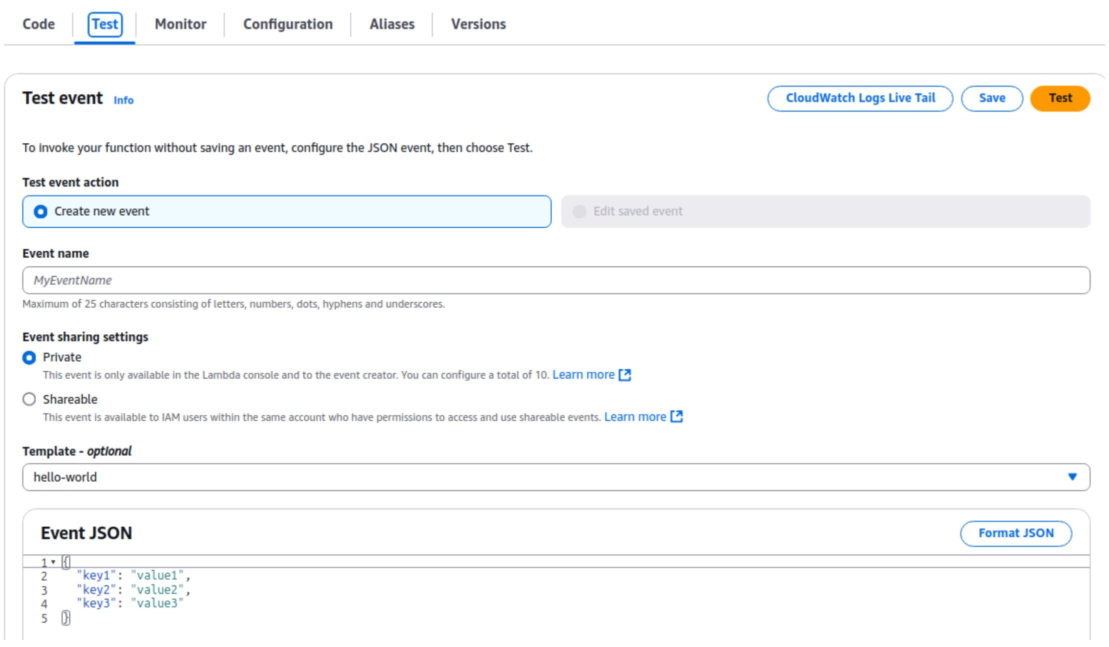
Now, refer to tutorial 4 to create a new AWS Lambda function using the management console.
Give the function a name, such as 'python-hello'.
For the runtime, select '**Python 3.13**'.

Instead of uploading a jar file, upload the **index.zip** file for the code.
The function handler uses the "lambda_function.py" SAAF wrapper that points to handler.py.

This function handler has a default file name and method name.
It is not necessary to update the handler this way in AWS Lambda for SAAF Python functions.

4. Test Python Function

Testing Python functions can be performed using the AWS Lambda GUI.
Once you've deployed your code, navigate to the "Test" tab in the AWS Lambda GUI.
This GUI tool provides a CLI alternative for fast testing of functions.



To invoke your function without saving an event, configure the JSON event, then choose Test.

Test event action
[✓ Create new event] Edit saved event

Event name
MyEventName
Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

Event sharing settings
[✓ Private]
This event is only available in the Lambda console and to the event creator. You can configure a total of 10. Learn more
Shareable
This event is available to IAM users within the same account who have permissions to access and use shareable events. Learn more

Template - *optional*
hello-world

Event JSON

```
{
    "key1": "value1",
    "key2": "value2",
    "key3": "value3"
}
```

Set up a new test event:
Set the name - Event name: **myTest1**
Modify the JSON object.
Provide only a "**name**" key-value pair.
Provide a name for the hello function.
Save, and the press the Test button to test the Python function.

5. Test Python Function with FaaSRunner Tool from Tutorial 4

After verifying that the function works, to receive credit for Task 1 of the tutorial, using the FaaS Runner tool from Tutorial 4, call your Python test function with using 50-parallel threads. If you have not increased your AWS Lamba Service Quota for Concurrent Executions beyond 10, it is okay to use only 10-parallel threads.

From the SAAF directory:

```
cd test
cp functions/exampleFunction.json functions/pythonHello.json
```

Update the pythonHello.json file to test this function:

function:	**`your-function-name`**
source:	**`../python_template`**

```
# update the function JSON file with your favorite text editor
nano functions/pythonHello.json
```

```
cp experiments/exampleExperiment.json experiments/pythonHelloExp.json

# update the experiment JSON file with your favorite test editor
nano experiments/pythonHelloExp.json
```

Set as follows:

```
    "runs": 50,
    "threads": 50,
    "iterations": 1,
```

Now run the experiment.

To receive extra credit for Python hello, upload the CSV, or even better a formatted .xlsx or .odf file to Canvas. Alternatively, the running function can be demonstrated and verified as correct after class or during office hours to have points immediately added in Canvas.

Task 2 – Deploy and Test a Container Based AWS Lambda Function (15 points)

1. Install the Docker Engine on your Ubuntu (Virtual) Machine

First, on your Linux machine, install the Docker container engine to support working with Docker containers. In tutorial 7, this step is completed on an ec2 instance. In this tutorial, install Docker in your local Ubuntu environment. If using the Windows sub-system for Linux, WSL2 is required.

Highlight the commands, and copy-and-paste to the VM.

USE THE MS WORD DOCUMENT NOT THE PDF FILE TO COPY AND PASTE THE COMMANDS BELOW

When copying from the PDF file, occasionally certain characters are not interpreted correctly resulting in errors. In particular, lines may be broken in two, that need to be continuous lines. Please check them.

```
sudo apt update

sudo apt install apt-transport-https ca-certificates curl software-properties-
common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker-archive-keyring.gpg
```

```
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-
archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs)
stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# refresh sources
sudo apt update

apt-cache policy docker-ce

# install packages
sudo apt install docker-ce

#verify that docker is running
sudo systemctl status docker
```

The "Docker Application Container Engine" should show as running.

The Docker daemon, by default, uses an IPC socket to support interprocess-communication between processes on the same Docker host. The Docker daemon, by default, always runs as the **root** user. Consequently, the Docker IPC socket is owned by the root user, and other users on the Linux system can only access this IPC socket using sudo. This means you will be required to preface all Docker commands with "sudo" on your system.

To allow your Ubuntu user account to work with docker, add your user to the docker group:

```
sudo usermod -aG docker $USER
```

After this update, **it is necessary to** *****REBOOT the VM*****, and log back into the VM.

This tutorial assumes your user account can run docker commands directly.
If you do not add your user to the docker group, you can run docker commands as the superuser.
To do this, preface ALL docker commands with "sudo".

If not wanting to configure the "docker" group, you can save from typing "sudo" for each command by assuming the role of superuser in your bash shell by typing: "sudo bash".

As alternative to installing the Docker Container engine, is to install Docker Desktop.
Installing Docker Desktop **may require more system resources**: ***memory, disk space, CPUs***, than only with the Docker container engine. Installing just the Docker engine is the lightest-weight approach for VMs and computers with minimum resources.

If interested in installing Docker Desktop, please refer to the instructions here:
https://docs.docker.com/desktop/setup/install/linux/ubuntu/

2. Set up a new SAAF folder for working with the container-based serverless function

Create a new directory for 'container-function', and then clone SAAF.

```
mkdir container-function
cd container-function
git clone https://github.com/wlloyduw/SAAF.git
```

Next, navigate to the aws_docker_debian bash_container folder.

```
cd container_template/functions/bash_container/aws_docker_debian
```

3. Create a reusable role to provide basic security policies for creating new AWS Lambda functions from the CLI/publish script

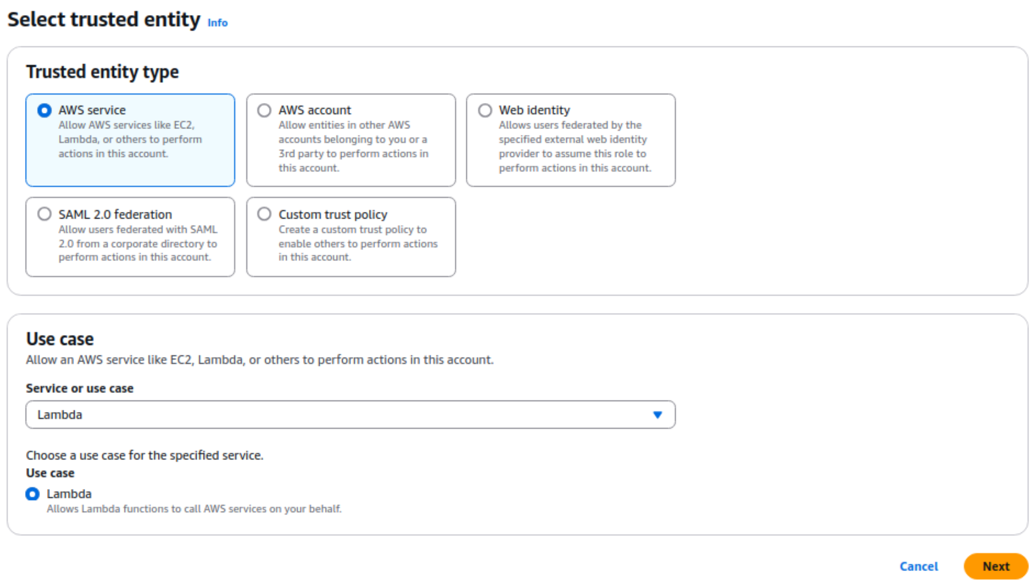
To use container based functions, we will need to create a reusable role which captures common security policies that you will grant to container-based functions. You can create different roles for different container function use cases. For example, you could have a basic-role that only has CloudWatch log permission, and an S3-role that provides both CloudWatch log permission and S3 Full Access if the serverless function is to work with S3 buckets. When working with container-based functions, to simplify their creation, we will use a deploy script which requires us to provide the role. This saves many steps in creating container-based functions using the AWS Management Console.

In the upper right-hand corner of the AWS management console, click on your username, and select '**Security credentials**'.

On the left, select '**Roles**', and select the button to create a new role:

Create role

Select the following settings as in the screen image below:
Trusted entity type: **AWS service**
Use case – Service or use case: **Lambda**
Use case – Use case: **Lambda**



Select trusted entity Info

Trusted entity type

[✓ AWS service] Allow AWS services like EC2, Lambda, or others to perform actions in this account.

AWS account — Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

Web identity — Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

SAML 2.0 federation — Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

Custom trust policy — Create a custom trust policy to enable others to perform actions in this account.

Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case: Lambda

Choose a use case for the specified service.
Use case
[✓ Lambda] Allows Lambda functions to call AWS services on your behalf.

On the next screen, search for '**AWSLambdaBasicExecutionRole**', and add this policy.

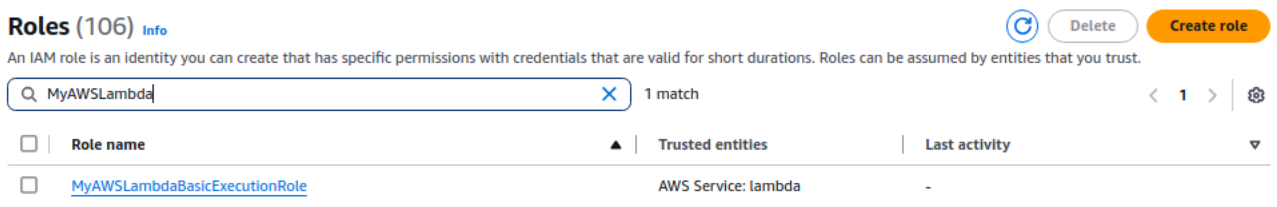
The basic execution role grants permission to write CloudWatch log files.

If doing a custom function for a project – at this step, you would add any other required policies, such as: 'AmazonS3FullAccess', 'AmazonRDSFullAccess', or 'AWSLambdaVPCAccessExecutionRole'.

After selecting policies, click the **next** button.
Assign a role name such as: "**MyAWSLambdaBasicExecutionRole**"

Finally, create the role by pressing the "**Create role**" button.

Now we need to copy the Amazon Resource Name (ARN) that uniquely identifies the role you just created.
In the "**Roles**" list, search for, and locate the role you just created:



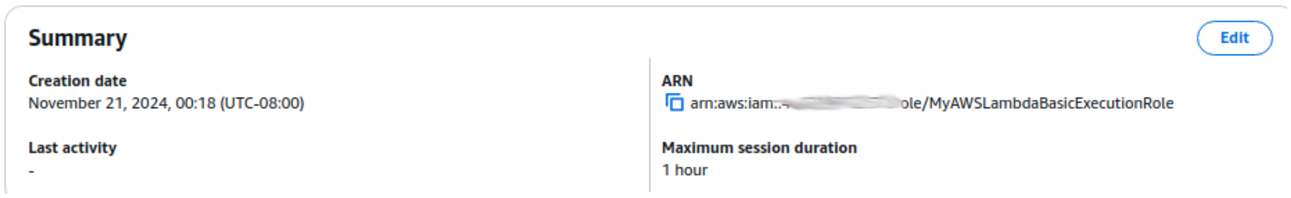
Roles (106) Info

An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.

MyAWSLambda — 1 match

Role name	Trusted entities	Last activity
MyAWSLambdaBasicExecutionRole	AWS Service: lambda	-

Click on the Role Name.



Summary

Creation date
November 21, 2024, 00:18 (UTC-08:00)

Last activity
-

ARN
arn:aws:iam:[redacted]ole/MyAWSLambdaBasicExecutionRole

Maximum session duration
1 hour

Click on the paste icon directly under **ARN** as shown above.

4. Update the config.json file to specify details regarding your container based function

Now edit the **config.json** file using a preferred text editor, and replace the ARN with the ARN you just copied.

The **config.json** file enables you to customize details of your container based function for deploying using automated scripts. No other items need to be configured for this tutorial, but here is a description of the options:

Parameter	Description
faaset_parent_platform	set to 'aws_docker'. Do not change.
version	has no effect currently
role	Amazon resource name for the role that defines access policies for the function
subnets	a list of comma delimited subnets to deploy the container-based function
security_groups	the security group(s) for the function – required when deploying to a VPC
timeout	the function timeout 1 to 900 seconds
storage	the size of the tmp filesystem from 512 to 10240 MB.
profile	normally set this to default, this specifies which set of AWS CLI credentials to use from

	the ~/.aws/credentials file.
architectures	the CPU architecture, either ‘x86_64’ or ‘arm64’
memory	function memory from 128 to 10240 MB. This also scales the cpu timeshare and vCPUs.
handler	the Python function handler in the container - should not be necessary to change this.
env	system environment variables to be made available to the function
runtime	version of Python to use on AWS Lambda
region	region to deploy the function
function_name	name of the container-based function

This container based function uses the Python SAAF function template to shell out and run a bash script.
Inside the bash script, you could run any program, in any language.
The first and only argument passed to the bash script is the request JSON object from AWS Lambda.

5. Update the bash script which is used to deploy custom code/programs written in any language using a container-based AWS Lambda function

In the bash script, let’s now modify the code to parse the JSON object, and capture the ‘name’ attribute from JSON.

Using your preferred editor, edit the script called **your_script.sh**.

Replace the primitive script, with a new bash script which parses the name attribute from the json object, to say hello:

```
json=$1

# parse the name attribute
name=`echo -e $json | jq -r .name`

# say hello world if there is no name
if [ -z "${name}" ]; then
        name="World"
fi

echo -n "Hello $name!"
```

5.b. Special Items for M1/M2 Mac and ARM64 platforms

To build, publish, and run the container-based AWS Lambda function using an ARM64-based client computer make the following updates:

In **config.json**, set the architectures to:

```
"architectures": "arm64",
```

In the **Dockerfile**, update the first line with the ‘FROM’ statement. Add “arm64v8/” in front of the base image:

```
FROM arm64v8/python:3.12-slim-bookworm
```

As a workaround, use AWS ec2 instances such as x86_64 (c6i.large) or arm64 (c8g.large) to build a container-based platform with the desired platform, x86_64 vs. arm64.

6. Build the Docker Container for the container-based function

Next, build the docker container for your container-based function.
You must have added your Ubuntu user to the docker group, or this will fail.
If this is the case, the **build.sh** script must be edited and '**sudo**' must be added in front of every docker command.

Run the build script:

```
./build.sh
```

7. Publish the Container Based AWS Lambda function to the Elastic Container Registry, and Create the new AWS Lambda Function

Using the publish script we can perform all steps to publish the container based AWS Lambda function.

The publish script does the following. Alternatively, without a script, these steps can be performed using the AWS Management Console GUI and docker commands.

1. Create an elastic container registry (ECR) in your AWS region called 'saaf-functions' to store docker container images. This is like a container image registry like Docker Hub in tutorial 7, except it is hosted on AWS.

   ****NOTE: It costs 10c/GB/month to store Docker images in an ECR registry**
   In the first year, 50 GB/month of ECR storage is provided for free.
   After the first year, there is no free ECR storage.
   The docker image for this example uses 102.23 MB. (~$0.01 storage cost per month)

   A password is created and saved for your ECR registry at:
   `/home/(your-username)/.docker/config.json`

2. The docker image you built is tagged (scripts a docker command)
3. The tagged docker image is pushed to the ECR registry (scripts a docker command)
4. The script checks if the container-based function already exists, if it exists the function is updated, if the function does not exist, it is created from scratch.
5. When the function is updated or created, in addition to creating the function, the script waits for the process to be finished, and then automatically creates an AWS Lambda Function URL for calling the function.

Now run the publish script, and watch for errors.
(* * * CROSS FINGERS * * *)

```
./publish.sh
```

8. Test your container-based AWS Lambda function

If everything worked, it should now be possible to test the container-based function.

First, inspect the script file called run.sh by opening it using a preferred text editor.
This script is similar to callservice.sh. At the top, a name is provided inside a json object.
Feel free to change the name, but replace (escape) any spaces ‘ ‘ with ‘\u0020’.

Now trying running the container based function.
(* * * DOUBLY CROSS FINGERS * * *)

```
./run.sh
```

If your function worked a long JSON object should appear.
If you scroll up, the standard output for the container based function is captured in the standard_output parameter.

Next, for extra credit, modify your_script.sh to run the Linux sysbench benchmarking tool to execute a CPU-bound (compute-bound) task to generate millions of prime numbers. By calling sysbench in your bash script it will be possible to profile resource utilization of prime number generation on AWS Lambda using SAAF.

Please note, this is a newer version of SAAF than what is used in Tutorial 4. This version includes additional profiling data.

9. Adapt the bash script to run the Linux sysbench prime number generation benchmark on AWS Lambda

First, modify the Dockerfile to install the sysbench tool.

Modify the line as follows adding the word ‘**sysbench**’ to the end:

```
RUN apt-get install -y curl jq sysbench
```

Now replace the code in your_script.sh with a call to sysbench. This should be the only thing that your function does. This command uses double-dashes for the command line arguments. If the wrong dash character is used, the command will FAIL. It may be better to type this command out using the keyboard to ensure the right dash character is used. This should be the key to the right of the ZERO on the keyboard.

```
sysbench cpu --cpu-max-prime=2000000 --threads=2 --events=10 run
```

After you’ve updated your function, rerun the build, publish, and run scripts to rebuild, redeploy, and test the change.

10. Test the sysbench Lambda function using FaaSRunner. Investigate relationship between function memory and runtime. Investigate which CPU type is fastest.

Now, using FaaSRunner, create a function file (container.json), and an experiment file (containerExp.json).

In the function file, specify the function name and source:

```
"function": "bash_container",
"source": "../python_template",
```

In the experiment file, specify:

```
"memorySettings": [512,1024,2048,3200],
"runs": 50,
"threads": 50,
"iterations": 1,
```

This experiment configuration, will repeat the experiment 4 times by making 50 parallel calls.
Each experiment will test a different AWS Lambda function memory setting: 512 MB, 1024 MB, 2048 MB, and 3200 MB.

On AWS Lambda, vCPUs and CPU time share are linked to function memory.
As we increase the memory, your code will receive more CPU time to complete its work.

Since prime number generation is a compute-bound task, and because we are using a benchmark that performs a fixed amount of work for every call, we can observe how the runtime changes as memory is increased.

Submit a PDF file. Include the following

1. Report the average function runtime at 512 MB, 1024 MB, 2048 MB, and 3200 MB for 50 parallel calls when running sysbench.
2. Describe what happens to your observed average function runtime when increasing function memory from 512 MB to 3200 MB.
3. For the results for the 3200 MB run, if you received more than one type of CPU, identify which CPU type had the fastest function runtime at 3200 MB.
4. Identify the average function runtime for each CPU type you received. This is CPU heterogeneity on a cloud platform.

Submit to Canvas, the PDF file, and your CSV, XLSX, or ODF FaaSRunner report output.

Optionally, your results can be graded in-person or over Zoom during/after the class, or during office hours for an immediate credit update to Canvas.