Tutorial 8 – Introduction to Lambda IV: AWS Step Functions, AWS SQS

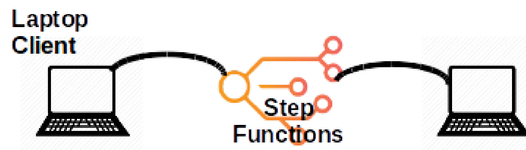
Disclaimer: Subject to updates as corrections are found
Version 0.11

The purpose of this tutorial is to introduce the use of AWS Step Functions to instrument control flow for multi-function serverless applications. The tutorial also provides a brief introduction to the Simple Queue Service (SQS).

AWS Step Functions support defining state machines to specify the serverless application control flow for a serverless application. Using AWS step functions, control flow is implemented on the cloud-provider's side. The client only needs to call the state machine to execute a workflow of functions. For this tutorial, we will connect the Encode and Decode Lambda functions from Tutorial #4 using AWS Step Functions so that a message is passed into the Encode function, shifted, and then unshifted automatically by calling Decode. The AWS step function eliminates network traffic between the client and cloud server providing a speed-up.

Tutorial #4 Caesar Cipher with a laptop client calling Lambda functions:

Tutorial #9 Caesar Cipher with Step Functions client calling Lambda functions:



1. Update Caesar Cipher Lambda Functions For Use With AWS Step Functions

When working with AWS Step Functions, data output from one Lambda function is passed to the next Lambda function as input. To prepare the Encode and Decode Caesar cipher functions for use in an AWS Step Functions state machine, it is necessary to change the output of the encode function to generate the key/value pairs needed by the decode function.

From Tutorial #4, modify the Response.java object so that the Encode function produces the follow JSON output:

Encode INPUT:

```
{
  "msg": "ServerlessComputingWithFaaS",
```

```
  "shift": 22,
}
```

Invoking Lambda function:ENCODE using AWS CLI

real 0m1.004s
user 0m0.312s
sys 0m0

Encode OUTPUT:


```
{
  "msg": "OanranhaooYkilqpejcSepdBwwO",
  "shift": 22,
  "decodeTime": 0,
  "encodeTime": 1,
  "uuid": "501064cc-1b1e-4f91-9ed3-d2d04b599db7",
  "error": "",
  "vmuptime": 1543554220,
  "newcontainer": 0
      . . . (metrics from SAAF)
}
```


Here, the response of the encoding is returned using the “msg” key/value pair to match the input of Decode. The “shift” is also returned so this can be passed directly to decode.
This JSON can now be passed directly to the decode Lambda function.

If you implemented a Response class, add to the bottom of the Encode class handleRequest method:

```
// Set return result in Response class, class is marshalled into JSON
// response object may have been declared as 'r' or 'response'
r.setMsg(msg);
r.setShift(shift);
```

Alternatively, you can use SAAF’s addAttribute method:

```
inspector.addAttribute("msg",msg);
inspector.addAttribute("shift",shift);
```

If you implemented a Response class, for the new properties, add getters and setters to the Response class:

Here we assume that Encode and Decode are combined into a single Java project to build a single deployment JAR file for both the Encode and Decode functions:

```
String msg;
public String getMsg()
{
    return msg;
}
public void setMsg(String msg)
{
```

```
        this.msg = msg;
    }

    private int shift;
    public int getShift()
    {
        return shift;
    }
    public void setShift(int shift)
    {
        this.shift = shift;
    }

    long decodetime;
    public long getDecodetime()
    {
        return decodetime;
    }
    public void setDecodetime(long decodetime)
    {
        this.decodetime = decodetime;
    }

    long encodetime;
    public long getEncodetime()
    {
        return encodetime;
    }
    public void setEncodetime(long encodetime)
    {
        this.encodetime = encodetime;
    }
```

In addition to including message and shift, let's also report the processing time for both the Encode and Decode Lambda functions. This way when the two functions are composed together we can measure execution time of the individual Lambda functions.

The Java System.currentTimeMillis() can be used to capture the system time before and after execution of code in the handleRequest() method.

If you implemented a Response class, add the following code to the top of the handleRequest() method for both Encode and Decode:

```
long tStart = System.currentTimeMillis();
```

And then update the code at the bottom of both of the handleRequest() methods to:

```
// in both handlers:
long tEnd = System.currentTimeMillis();
// in the encode handler:
response.setEncodetime(tEnd - tStart);
// in the decode handler:
response.setDecodetime(tEnd - tStart);
```

Alternatively, use SAAF's addAttribute method:

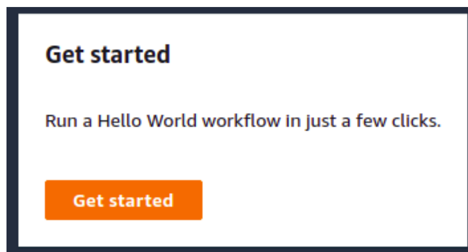
```
// in both handlers:
long tEnd = System.currentTimeMillis();
// in the encode handler:
inspector.addAttribute("encodetime",tEnd-tStart);
// in the decode handler:
inspector.addAttribute("decodetime",tEnd-tStart);
```

After making these code changes, redeploy your Encode and Decode Lambda functions.
Test that changes are applied by running callservice.sh as in Tutorial #4.

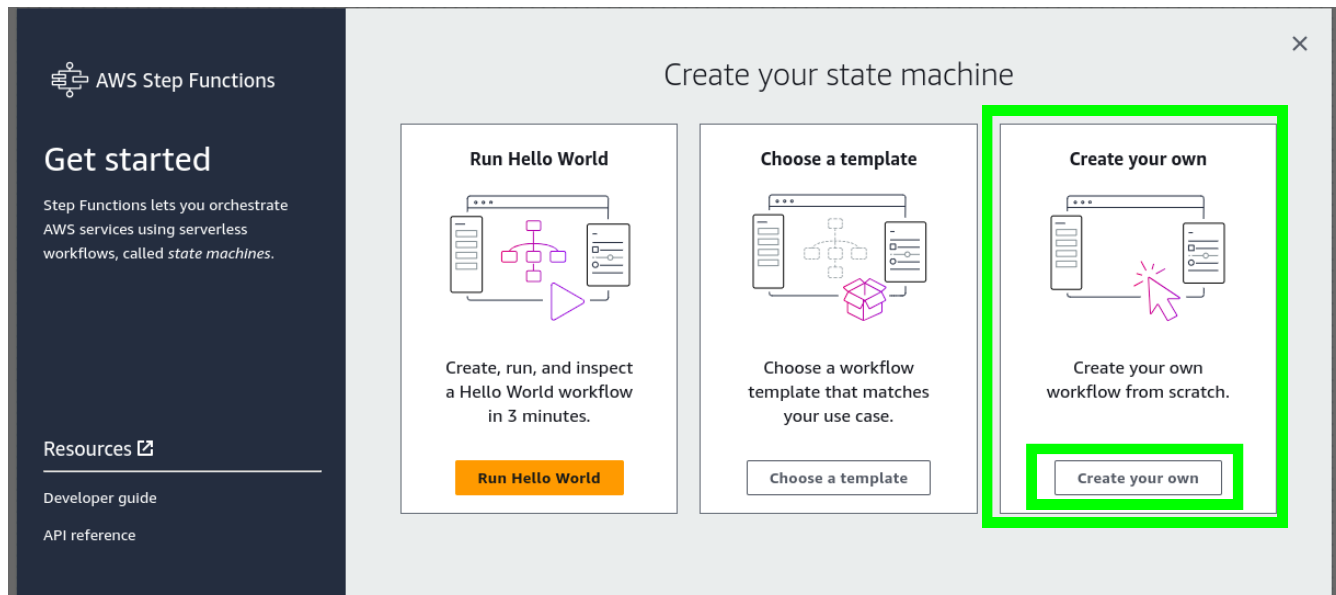
2. Create AWS Step Functions State Machine

Before creating the AWS Step Functions State Machine, be sure that you already have an Encode and Decode Lambda function defined. The Encode function should accept "msg" and "shift" and also output these parameters. The Decode function should then accept "msg" and "shift" to reverse the Caesar cipher.

To begin, search for the "Step Functions" cloud service in the AWS Management Console.
On your first visit, a "splash" screen is shown. In the upper right, click on the "**Get started**" button:



On the top of the screen, select "Get started".
Next for "Create your state machine", select "**Create your own**":



First, name the state machine. Click on the **Config** icon so that it turns blue:

Assign a state machine name and use the type “**Standard**”:

Details

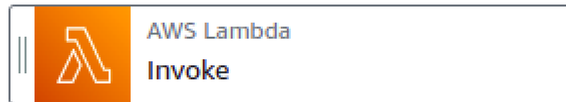
State machine name
State machine name cannot be changed after creation.
TCSS462-562-f2023-caesar-cipher
Must be 1-80 characters. Can use alphanumeric characters, dashes, and underscores.

Type Info
State machine type cannot be changed after creation.

Standard
Durable workflows for ETL, ML, e-commerce and automation. They can run for up to 1 year, and history is stored in Step Functions for auditing and playback. Supported by a feature-rich console debugger. Recommended for new users.

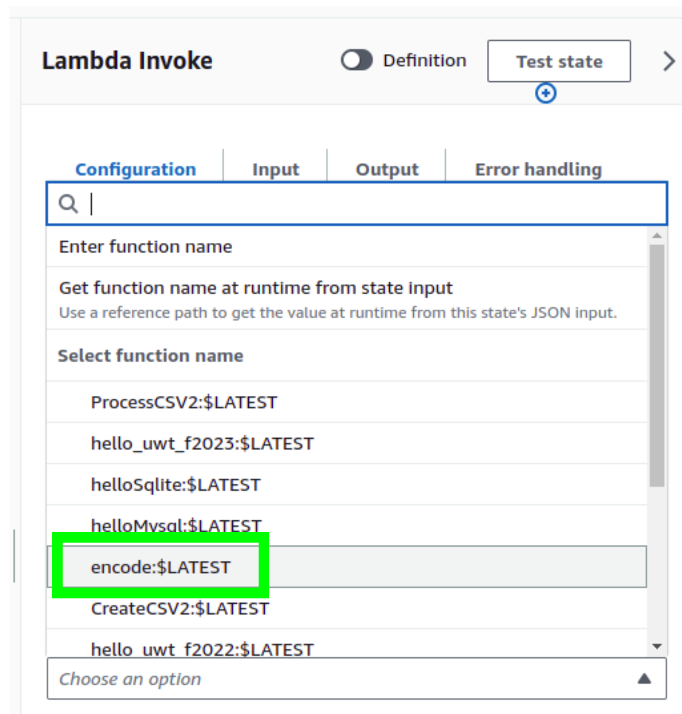
Return to the Design tab. Click on the “**Design**” button so that it turns blue:

On the left, drag the AWS Lambda Invoke widget onto the “Drag first state here” box:



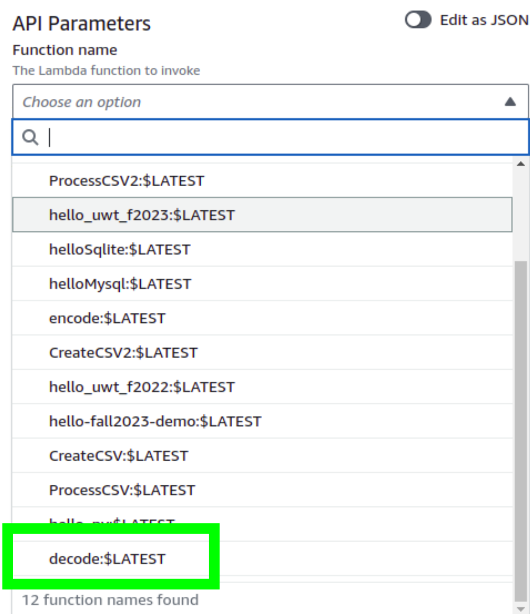
Drag the widget on top of the box:

On the right, specify the State name as “Encode”.
Next specify the “Function name”. Select “Encode” from the “Function name” dropdown list:

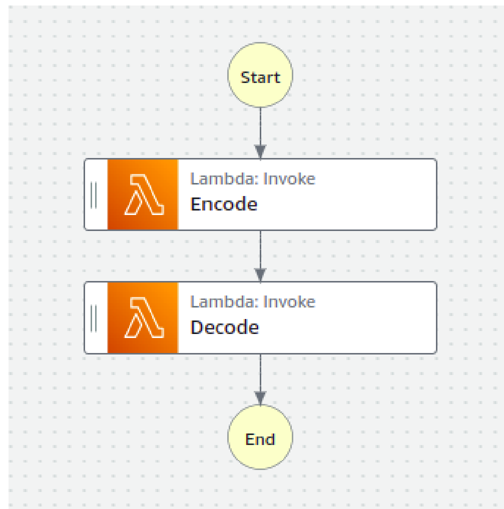


Use the default Payload setting: “Use state input as payload”.
Scroll down. For the “Next state” drop-down list, select “Add new state”:

For the new state, again drag the “AWS Lambda Invoke” widget onto “Drop state here” as before.
For the new state, on the right, specify the State name as “Decode”.
Next specify the “Function name”. Select “Decode” from the “Function name” dropdown list.



For **Payload**, keep the default of 'Use state input as payload'.
Confirm that your state machine looks as follows:



Now, in the upper right-hand corner press the "**Create**" button to create the state machine.

The GUI will ask you to confirm automatic security role creation.
Permission to execute AWS Lambda functions will be granted to your state machine.
Press "**Confirm**" to proceed.

Next, the state machine can be tested from the AWS Management Console GUI.
Press the "**Start execution**" button.

In the Input box, provide a properly formatted JSON object which defines a message and shift:

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

Press the "**Start execution"** to test the state machine.
The state machine will several seconds to run, especially if the Lambda functions are in the cold state.

Scroll down to see all of the events. It should be possible to observe each AWS Lambda function execution (output from encode is under ID 5, output from decode is under 10, and output from the state machine is under 12)

Events (12)

Filter by properties or search by keyword | Filter by a date

ID	Type	Step	Resource	Started After
1	ExecutionStarted			0
2	TaskStateEntered	encode		00:00:00.029
3	TaskScheduled	encode	Lambda | Log group	00:00:00.029
4	TaskStarted	encode		00:00:00.103
5	TaskSucceeded	encode		00:00:01.720
6	TaskStateExited	encode		00:00:01.737
7	TaskStateEntered	decode		00:00:01.737
8	TaskScheduled	decode	Lambda | Log group	00:00:01.737
9	TaskStarted	decode		00:00:01.815
10	TaskSucceeded	decode		00:00:03.392
11	TaskStateExited	decode		00:00:03.417
12	ExecutionSucceeded			00:00:03.466

Expand and inspect the JSON which is generated for key steps with IDs 5, 10, and 12.
Optionally inspect the other IDs.

For more information on Step Functions, refer to the developer guide:

https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html

Limited input and output processing is supported from within the state machine without writing (and calling) a separate Lambda function. For more information, see this article:

https://docs.aws.amazon.com/step-functions/latest/dg/concepts-input-output-filtering.html

3. Create a BASH client to invoke the AWS Step Function

AWS step function state machines can be invoked using the AWS CLI.

For step 3, customize the provided BASH script provided below to invoke your step function. Call this script "callstepfunction.sh".

The script requires installation of the awscli and jq packages.
These have been used in previous tutorials.

```
sudo apt install awscli jq
```

Next, add your state machine Amazon Resource Name (ARN) to the script:

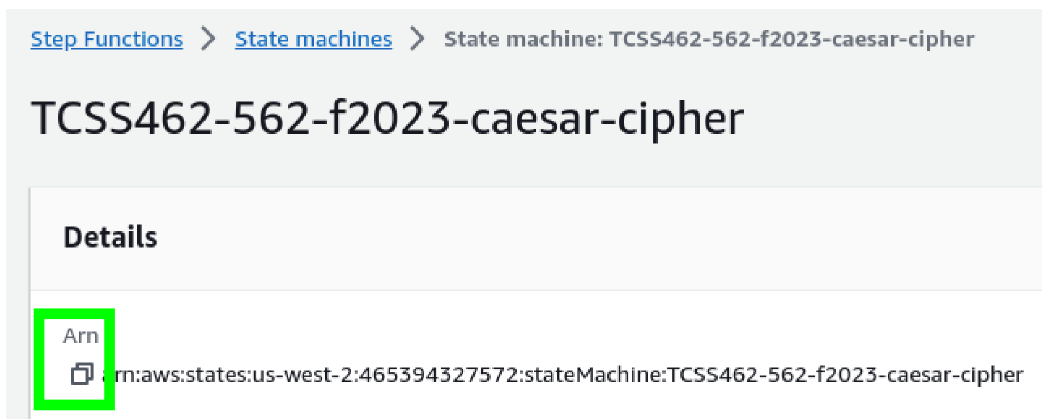
```
# JSON object to pass to Lambda Function
json={"\"msg\"":"\"ServerlessComputingWithFaaS\",\"shift\"":22}
smarn="<replace with state machine arn>"
exearn=$(aws stepfunctions start-execution --state-machine-arn $smarn --input $json
| jq -r ".executionArn")

# poll output
```

```
output="RUNNING"
while [ "$output" == "RUNNING" ]
do
  echo "Status check call..."
  alloutput=$(aws stepfunctions describe-execution --execution-arn $exearn)
  output=$(echo $alloutput | jq -r ".status")
  echo "Status check=$output"
done

echo ""
aws stepfunctions describe-execution --execution-arn $exearn | jq -r ".output" | jq
```

In the bash script, replace the state machine ARN (amazon resource name) with the value from the state machine GUI. There is a copy-icon which makes it easy to copy the ARN to the clipboard for pasting into the script:



The “aws stepfunctions start-execution” command launches an asynchronous execution of the state machine. The execution ARN is captured by the script.

Then, to determine when the state machine has completed, successive calls are made to “aws stepfunctions describe-execution” using the execution ARN to check the status. (polling!!)

When the state machine is no longer running, a call is made to describe-execution to capture the JSON result.

4. Create a Simple Queue Service Queue for messages

Using the AWS Management Console, navigate to the “SQS” cloud service. On the first visit the splash screen will appear. Press the “Create queue” button:

Get started

Learn how to use Amazon SQS by creating a queue, sending a message to the queue, and receiving and processing the message.

Create queue

The Simple Queueing Service supports FIFO Queues which ensure first-in, first-out processing if needed:

https://aws.amazon.com/about-aws/whats-new/2019/02/amazon-sqs-fifo-qeues-now-available-in-15-aws-regions/

Assign the Queue name as “CaesarQ”, and select a “Standard” Queue type:

Standard Info
At-least-once delivery, message ordering isn't preserved
- At-least once delivery
- Best-effort ordering

Name
CaesarQ
A queue name is case-sensitive and can have up to 80 characters. You

Under “**Configuration**”, set the “**Receive Message Wait Time**” to be “20” seconds:

Receive message wait time Info
20 Seconds
Should be between 0 and 20 seconds.

Then click [Create Queue]:

Create Queue

5. Add a message to your SQS Queue from a Lambda function

Now, modify the decode Lambda function to write the decoded message to your SQS queue.

First, in your maven build file (pom.xml) add the dependency to include the SQS API.
Adding the dependency directly using the search feature in the GUI may not work.
Modifying the pom.xml file directly should work:

```
        <dependency>
           <groupId>com.amazonaws</groupId>
```

```
        <artifactId>aws-java-sdk-sqs</artifactId>
        <version>1.12.599</version>
    </dependency>
```

Next, modify the tail end of the **Decode** Lambda function's handleRequest() method to submit a message to your newly created SQS queue. Post the decoded message to the SQS queue for consumption.

The following import statements should be added near the top:

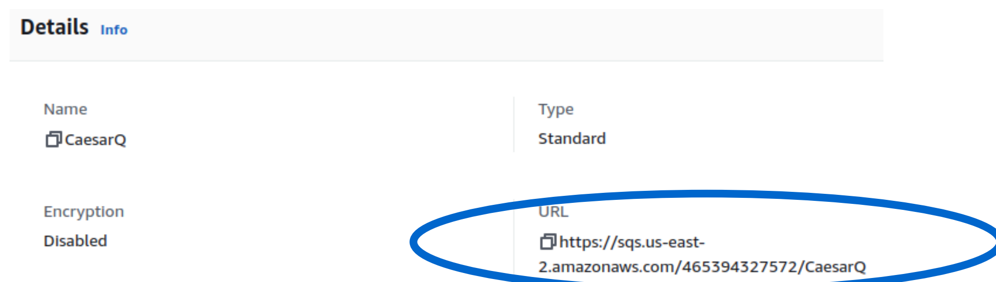
```
import com.amazonaws.services.sqs.AmazonSQS;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import com.amazonaws.services.sqs.model.SendMessageRequest;
```

Then the code to add a message to an SQS queue is as follows:

```
        AmazonSQS sqs = AmazonSQSClientBuilder.defaultClient();
        SendMessageRequest send_msg_request = new SendMessageRequest()
        .withQueueUrl("<INSERT SQS URL here>")
        .withMessageBody(msg)
        .withDelaySeconds(0);
        sqs.sendMessage(send_msg_request);
```

SQS queues use public http URLs for communication.
The public SQS URL for your queue can be found in the Details window pane once selecting the queue "**CaesarQ**" from the list of queues to inspect the queue's configuration.

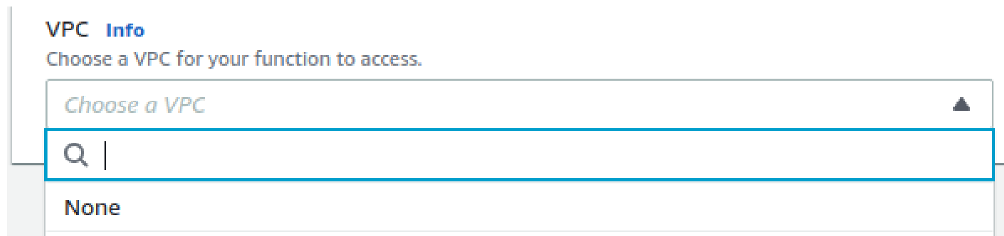
The URL property is listed under Details in the SQS GUI as below:



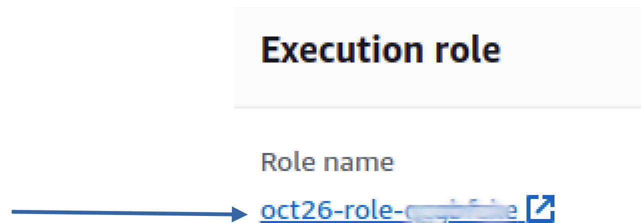
Next, compile, build, and redeploy your Lambda Decode function.

**** IMPORTANT CHANGES to DECODE LAMBDA FUNCTION ****

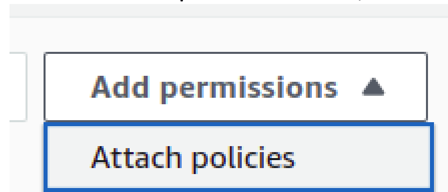
Two important changes are required for Decode to use SQS. First, Decode can no longer run in a VPC to use SQS unless the VPC has a NAT Gateway configured to enable public internet access. Because the NAT gateway is very expensive, it is easier to simply delete the VPC, and set the Lambda function to use "None".
If you configured a VPC for your Lambda function, remove it now by selecting None:



Next, modify the security role for the Decode Lambda function to have permission to work with SQS queues. In AWS Lambda, select your function, then select the Configuration Tab, and select "Permissions" on the left. Use the BLUE link to modify the Decode function's security role:



In the role editor, select the "Add Permissions" drop-down button, and select "Attach policies":



Search for and attach the policy "**AmazonSQSFullAccess**". Click on the "**Add permissions"** button.

6. Modify BASH client to retrieve AWS Step Function result from SQS queue

Previously we polled the AWS Step Function by calling "aws stepfunctions describe-execution" repeatedly until a result was available. Now that Decode posts the message result to a queue, the result can be fetched from the queue instead.

If programming a Java or Python client to interact with a message queue, it would be possible to "subscribe" to the queue to receive messages as events. Note, this is the classic publish-subscribe message queue model common for distributed systems. AT UW-Tacoma, these queues are discussed in TCSS 558 Applied Distributed Computing.

The AWS CLI does not support a callback mechanism.

Instead, the CLI offers a blocking call that will call SQS and wait up to 20 seconds for a message to return. If no message appears, the call exits.

Copy your AWS Step Functions BASH client (**callstepfunction.sh**) and make a new version called (**callstepfunctionQ.sh**) to receive the Decode result back from SQS instead of from AWS Step Functions.

Directly replace all code above the "# poll output" comment in your callstepfunction.sh script to include the following code:

```
smarn="<Your State Machine ARN>"

# a file-based counter to generate unique messages for encode/decode
count=0
if [ -e .uniqcount ]
then
  count=$(cat .uniqcount)
fi
count=$(expr $count + 1)
echo $count > .uniqcount

# JSON object to pass to Lambda Function, uses the unique $count
json={"\"msg\"":"\"NEW-SQS-$count-
ServerlessComputingWithFaaS\",\"shift\"":22}

# Call the state machine
exearn=$(aws stepfunctions start-execution --state-machine-arn $smarn --
input $json | jq -r ".executionArn")
echo $exearn

# get output from SQS
msgs=$(aws sqs receive-message --queue-url <Your SQS Queue URL>)

# show result from SQS queue
echo $msgs | jq

# delete the message from the queue using the receipt handle
receipthandle=$(echo $msgs | jq -r .Messages[0].ReceiptHandle)
aws sqs delete-message --queue-url <Your SQS Queue URL> --receipt-handle
$receipthandle

exit
```

Messages in SQS queues are not deleted when read. Since we only want to consume this message once, use the receipthandle when calling the "aws sqs delete-message" CLI API to delete the message once it is read.

If messages are not deleted they will pile up in the queue, and requests to "aws sqs receive-message" may return a message from a previous execution of the state-machine.

Where SQS FIFO queues guarantee the order that messages will be delivered to clients, standard SQS queues do not guarantee ordering of messages. When calling "sqs receive-message" on a standard queue, ***its possible the latest message is not returned!***

By deleting messages once consumed, we shouldn't accidentally see them again.

The approach here will not scale, however! With multiple users executing the state machine concurrently, calls to "aws sqs receive-message" are not client specific. All of the results from the state machine are posted to the

same queue. It is possible to tag messages with a “GroupId” or a “DeduplicationId” for this purpose. This allows filtering of messages. If sharing a queue with many users, it may be necessary to pull batches of messages, filter them, and only consume and delete the client’s specific message. For these reasons, S3 may be preferable method for returning a single state machine result to a client as it can be tagged by the client through the workflow. Message queues are more ideally suited for distributed systems to orchestrate multiple nodes consuming and operating on shared data.

Now, test your **`callstepfunctionQ.sh`** BASH client and check out how well SQS works as an alternative to polling the aws stepfunctions describe-execution API.

Note, for the SQS bash script to work correctly, your encode and decode functions need to support upper and lower case letters and also the dash “-“ character. If you do not support these characters, please update your cipher so that it DOES support encoding and decoding these characters.

Tutorial 8 is optional and offered as extra credit in fall 2024.

To submit this tutorial

To submit tutorial #8, a Zoom recording should be made demonstrating that Tutorial 8 is working properly. Mp4 recordings can be made using Zoom. Create a personal Zoom session. Share the screen. Record your video to the cloud. While demonstrating operation of the script, feel free to speak and describe what is happening. After recording the video, upload the mp4 to Google Drive. Set the file so the course grader and instructor have read access. Grant read access to wlloyd@uw.edu and kirito20@uw.edu. Upload the video link on Canvas as your submission for tutorial 8.

To submit your tutorial, upload each of the following to Canvas:

1. **VIDEO LINK:** IF YOU WERE UNABLE TO PRESENT A WORKING SCRIPT OVER A LIVE ZOOM SESSION: **please provide an mp4 video recording**, or a PDF file with a URL to the recording made using Zoom. Videos can alternatively be hosted on Google Drive, a personal web server, or on YouTube. Please include a publicly accessible URL for the video to receive credit.
2. Your Bash scripts: **callstepfunction.sh** and **callstepfunctionQ.sh**
3. <OPTIONAL> screen captures can optionally be provided as a BACKUP in addition to the video recording

For the demonstration, be sure to start with an empty queue.
SQS messages can be purged using the GUI.

Then demonstrate calling the state machine by calling it twice with two different JSON messages.

Message 1: encode-→ decode “this is a unique message”
Message 2: encode → decode “second message of the demo”

The strings must be different.

The demo must show where in the **`callstepfunctionQ.sh`** BASH script the JSON object is created.
Alternatively, the BASH script can echo to the screen the message text before processing with Encode.

Your queue messages must be unique.
This should work because in callservice.sh a local counter has been added to ensure that each message is tagged with a unique ID.