

Tutorial 10 – Introduction to FaaS Runner

Disclaimer: Subject to updates as corrections are found

Version 0.10

Scoring: 40 pts maximum

The purpose of this tutorial is to provide a comprehensive overview of many of FaaS Runners most important features. This tutorial will cover creating complex experiments, automating them, and creating pipelines of functions.

1. Download the FaaS Runner tutorial functions.

To begin, using git, clone the GitHub repository for this tutorial.

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, create a folder and clone the source repository:

```
git clone https://github.com/RCordingly/faas_runner_tutorial
```

This tutorial builds upon Tutorial 4. If you have not completed that tutorial, please review it and install any dependencies (such as Maven and the AWS CLI). This tutorial does not require code changes but does require being able to deploy functions using the built in publish scripts.

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

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

2. Deploy the Included Functions

Included in the repository are four functions that need to be deployed to AWS Lambda. To simplify this process, SAAF's built in publish scripts can be used to deploy them automatically. The repository contains three 'Hello World' functions; pello_world, jello_world, and nello_world, and the CalcsService function.

The Jello/Pello/Nello naming is because these are Hello World functions written in **Java**, **Python**, and **Node.js** respectively. SAAF supports functions written in each of these languages.

To deploy these, we must first configure each config.json file with a role ARN. You should already have an ARN created from Tutorial 4 so you can retrieve that by visiting the AWS webpage, go to IAM -> Roles and select the role you would like to use.

The screenshot shows the AWS IAM console interface. On the left is a navigation sidebar with 'Identity and Access Management (IAM)' selected. The main content area shows the 'Summary' page for the role 'simple_microservice_role'. The 'Role ARN' is highlighted with a red box: `arn:aws:iam::616835888336:role/service-role/simple_microservice_role`. Below the summary, the 'Permissions' tab is active, showing a table of attached policies.

Policy name	Policy type
AWSLambdaFullAc...	AWS managed policy
AmazonS3FullAccess	AWS managed policy

There are 6 policies applied in total. A 'Show 4 more' link is visible below the table.

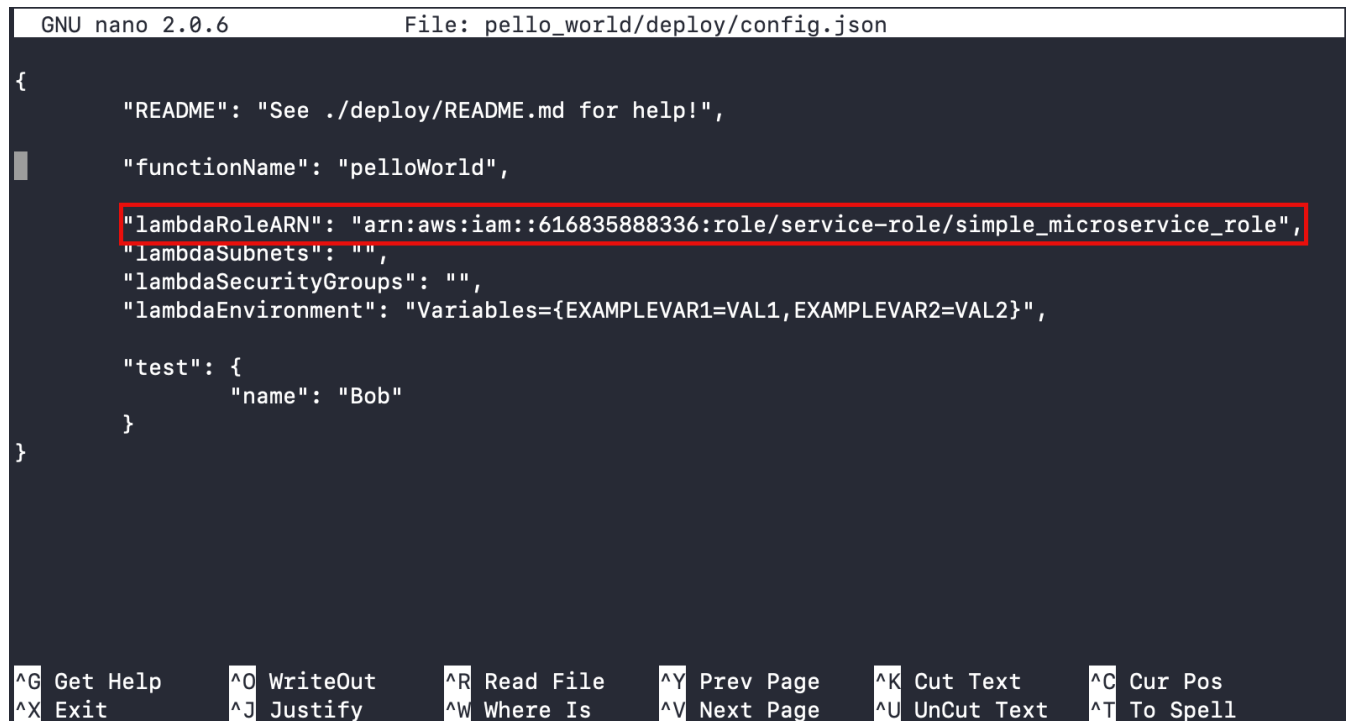
You can also easily get the role each of your functions are using by opening the terminal and running:

```
aws lambda list-functions
```

Copy your ARN shown at the top of the page. All functions can share the same ARN. Next open the **config.json** files located in each function's **deploy** folder and paste the ARN into the JSON attribute called **lambdaRoleARN**. **No other attributes in the config files need to be changed.**

You may use any text editor to enter the ARN. The example below shows opening each file in Nano.

```
cd {base directory where project was cloned}
nano pello_world/deploy/config.json
nano jello_world/deploy/config.json
nano nello_world/deploy/config.json
nano calcService/deploy/config.json
```



```
GNU nano 2.0.6      File: pello_world/deploy/config.json
{
  "README": "See ./deploy/README.md for help!",
  "functionName": "pelloWorld",
  "lambdaRoleARN": "arn:aws:iam::616835888336:role/service-role/simple_microservice_role",
  "lambdaSubnets": "",
  "lambdaSecurityGroups": "",
  "lambdaEnvironment": "Variables={EXAMPLEVAR1=VAL1,EXAMPLEVAR2=VAL2}",
  "test": {
    "name": "Bob"
  }
}
```

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell

3. Deploy each Function.

Once each configuration file has an ARN, each function should be able to be deployed using the publish scripts.

```
cd {base directory where project was cloned}

# ./publish.sh AWS GCF IBM AZURE MEMORY
./pello_world/deploy/publish.sh 1 0 0 0 1024
./jello_world/deploy/publish.sh 1 0 0 0 1024
./nello_world/deploy/publish.sh 1 0 0 0 1024
./calcs_service/deploy/publish.sh 1 0 0 0 1024
```

The publish scripts automatically package functions and can deploy them to AWS Lambda, Google Cloud Functions, IBM Cloud Functions, and Azure Functions. Here we are just deploying to AWS Lambda with a

memory reservation setting of 1024 MBs. The publish scripts can be used to deploy new functions or update existing functions.

To verify that each deployment was successful, the publish script will automatically invoke the function with the **test** payload in the config file. Verify that each function was deployed and executed successfully. The output should look similar to the example below.

```
Testing function on AWS Lambda...
{"cpuType":"Intel(R) Xeon(R) Processor @ 2.50GHz","cpuNiceDelta":0,"vmuptime":1603561560,"cpuModel":"62","linuxVersion":"#1 SMP Fri Sep 11 23:37:26 UTC 2020","cpuSoftIrqDelta":0,"cpuUsrDelta":0,"uuid":"1f315052-6be1-453d-9a91-ea05f38ea3c7","platform":"AWS Lambda","contextSwitches":18195,"cpuKrn":80,"cpuIdleDelta":0,"cpuIowaitDelta":0,"newcontainer":1,"cpuNice":0,"startTime":1603563155979,"lang":"java","cpuUsr":87,"majorPageFaultsDelta":0,"freeMemory":"1044356","frameworkRuntime":60,"contextSwitchesDelta":0,"frameworkRuntimeDeltas":16,"vmcpusteal":15,"cpuKrnDelta":0,"functionName":"jelloWorld","vmID":"NSHoq7","cpuIdle":318888,"runtime":76,"message":"Jello Bob","version":0.5,"cpuIrqDelta":0,"pageFaultsDelta":0,"functionMemory":"1024","functionRegion":"us-east-1","cpuIrq":0,"totalMemory":"1190852","cpuCores":"2","cpuSoftIrq":0,"cpuIowait":9,"endTime":1603563156055,"containerID":"2020/10/24/[$LATEST]2cce5ba583ec44fa9bf598a8a21d01b3","majorPageFaults":918,"vmcpustealDelta":0,"pageFaults":73631,"userRuntime":0}{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

Each function should now be visible on the AWS Lambda web page:

Functions (51)

Last fetched 3 minutes ago

Actions

Create function

Filter by tags and attributes or search by keyword

1

2

	Function name	Description	Runtime	Code size	Last modified
<div></div>	calcsServiceTutorial		Java 8	262.9 kB	16 minutes ago
<div></div>	jelloWorld		Java 8	263.2 kB	8 hours ago
<div></div>	pelloWorld		Python 3.7	4.2 kB	8 hours ago
<div></div>	nelloWorld		Node.js 10.x	22.1 kB	8 hours ago

4. Running an Experiment with FaaS Runner

Now that we have all of our functions deployed, we will begin running some experiments with FaaS Runner. To work with FaaS Runner, open the **test** folder in a terminal and execute the **faas_runner.py** script.

FaaS Runner uses two types of files. Function files, which define the endpoints needed to execute a function, and experiment files that define how to process an experiment. Let's execute the built-in calcsService experiment to get an understanding of what FaaS Runner is doing and how the output is recorded.

```
cd ./test
./faas_runner.py -f ./functions/calcsService.json -e ./experiments/calcsServiceExp1.json
```

The **-f** flag defines the path to the function file and **-e** defines the path to the experiment file. After executing this function FaaS Runner should execute the entire experiment and automatically open a spreadsheet on MacOS and Linux.

FaaS Runner produces a lot of output text to show what is going on. It is broken into section that will be explained here.

```
robertcordingly@Roberts-iMac-2:~/Dropbox/Research/faas_runner_tutorial/test$ ./faas_runner.py -f ./functions/calcsService.json -e ./experiments/calcsServiceExp1.json

-----
LOADING EXPERIMENTS AND APPLYING OVERRIDES... (faas_runner.py)
-----

Overrides: {}

Loaded function: {'function': 'calcsServiceTutorial', 'platform': 'AWS Lambda', 'source': '../calcs_service', 'endpoint': '', 'sourceFile': './functions/calcsService.json'}

---Loaded function list: [{'function': 'calcsServiceTutorial', 'platform': 'AWS Lambda', 'source': '../calcs_service', 'endpoint': '', 'sourceFile': './functions/calcsService.json'}]
NOTE: parentPayload missing in experiment file! Using default option of {}
NOTE: payloadFolder missing in experiment file! Using default option of
NOTE: shufflePayloads missing in experiment file! Using default option of False
NOTE: passPayloads missing in experiment file! Using default option of False
NOTE: transitions missing in experiment file! Using default option of {}

Loaded experiment: {'callWithCLI': True, 'callAsync': False, 'memorySettings': [], 'payloads': [{'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}], 'runs': 10, 'threads': 10, 'iterations': 1, 'sleepTime': 5, 'randomSeed': 42, 'outputGroups': [], 'outputRawOfGroup': [], 'showAsList': [], 'showAsSum': ['newcontainer'], 'ignoreFromAll': ['zAll', 'lang', 'version', 'linuxVersion', 'platform', 'hostname'], 'ignoreFromGroups': ['1_run_id', '2_thread_id', 'cpuModel', 'cpuIdle', 'cpuIowait', 'cpuIrq', 'cpuKrn', 'cpuNice', 'cpuSoftIrq', 'cpuUusr'], 'ignoreByGroup': {'containerID': ['containerID'], 'cpuType': ['cpuType'], 'vmID': ['vmID']}, 'invalidators': {}, 'removeDuplicateContainers': False, 'overlapFilter': 'functionName', 'openCSV': True, 'combineSheets': False, 'warmupBuffer': 0, 'sourceFile': './experiments/calcsServiceExp1.json', 'experimentName': 'calcsServiceExp1', 'parentPayload': {}, 'payloadFolder': '', 'shufflePayloads': False, 'passPayloads': False, 'transitions': {}}

---Loaded experiment list: [{'callWithCLI': True, 'callAsync': False, 'memorySettings': [], 'payloads': [{'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}], 'runs': 10, 'threads': 10, 'iterations': 1, 'sleepTime': 5, 'randomSeed': 42, 'outputGroups': [], 'outputRawOfGroup': [], 'showAsList': [], 'showAsSum': ['newcontainer'], 'ignoreFromAll': ['zAll', 'lang', 'version', 'linuxVersion', 'platform', 'hostname'], 'ignoreFromGroups': ['1_run_id', '2_thread_id', 'cpuModel', 'cpuIdle', 'cpuIowait', 'cpuIrq', 'cpuKrn', 'cpuNice', 'cpuSoftIrq', 'cpuUusr'], 'ignoreByGroup': {'containerID': ['containerID'], 'cpuType': ['cpuType'], 'vmID': ['vmID']}, 'invalidators': {}, 'removeDuplicateContainers': False, 'overlapFilter': 'functionName', 'openCSV': True, 'combineSheets': False, 'warmupBuffer': 0, 'sourceFile': './experiments/calcsServiceExp1.json', 'experimentName': 'calcsServiceExp1', 'parentPayload': {}, 'payloadFolder': '', 'shufflePayloads': False, 'passPayloads': False, 'transitions': {}}]
```

The first section is where the function information and experiment data are loaded. Here you can see the list of loaded functions, and the list of loaded experiments. For this experiment we only have one function and one experiment. If an experiment or function file is missing attributes (such as in this example **parentPayload**, **payloadFolder**, **shufflePayloads**, **passPayloads**, and **transitions**) default values will be used instead.

```
-----
PREPARING PAYLOADS... (experiment_orchestrator.py)
-----

Not loading payloads from folder. Either folder does not exist or payloadFolder is undefined.
Skipping setting memory value.
Sleeping after setting memory value...
Running test 0:
```

```

CREATING AND RUNNING THREADS FOR EXPERIMENT

Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
Call Payload: {'threads': 2, 'calcs': 1000, 'sleep': 0, 'loops': 1000, 'arraySize': 1}
STDOUT: {"vmuptime":1603653943,"cpuModel":"62","linuxVersion":"#1 SMP Fri Sep 11 23:37:26 UTC 2020","cpuSoftIrqDelta":0,"uuid":"1c153aba-0322-47f0-bb62-1dec327ccaa7","contextSwitches":33090,"cpuNice":0,"cpuUsr":140,"majorPageFaultsDelta":0,"freeMemory":"1037120","calcs":1000,"contextSwitchesDelta":43,"vmcpusteal":113,"cpuKrnDelta":0,"vmID":"7Mg8rw","cpuIdle":543882,"runtime":116,"version":0.5,"cpuIrqDelta":0,"cpuCores":2,"cpuSoftIrq":0,"containerID":"2020/10/25/[LATEST]9cf394b6d0e4cc2a7d3223cc29aadf","vmcpustealDelta":1,"cpuType":"Intel(R) Xeon(R) Processor @ 2.50GHz","cpuNiceDelta":0,"cpuUsrDelta":8,"platform":"AWS Lambda","cpuKrn":186,"cpuIdleDelta":15,"sleep":0,"cpuIowaitDelta":0,"newContainer":0,"startTime":"1603655765466","lang":"java","finalCalc1":187966,"finalCalc0":78210,"frameworkRuntime":5,"frameworkRuntimeDeltas":1,"functionName":"calcsServiceTutorial","threads":2,"loops":1000,"arraySize":1,"pageFaultsDelta":376,"functionMemory":"1024","functionRegion":"us-east-1","cpuIrq":0,"totalMemory":"1190852","cpuIowait":15,"endTime":1603655765582,"majorPageFaults":923,"pageFaults":182656,"userRuntime":110}{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}

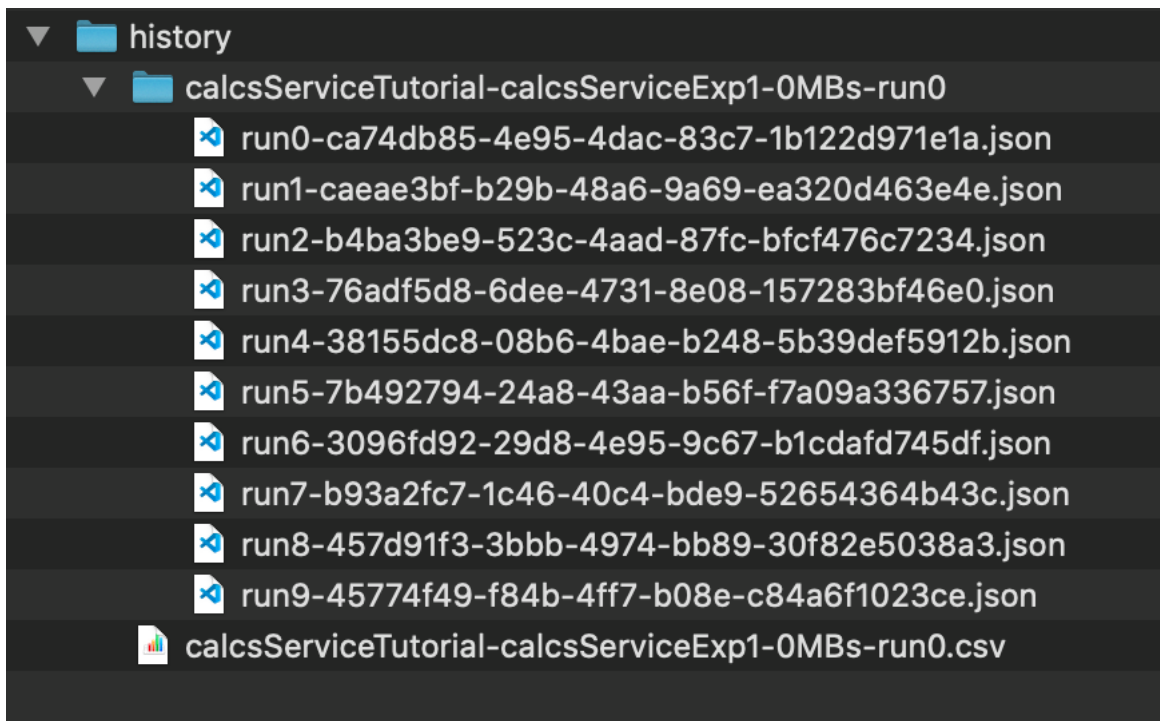
```

[illegible]

```
=====
WRITING REPORT TO FILE... (report_generator.py)
=====

Writing raw runs to folder ./history/calcsServiceTutorial-calcsServiceExp1-0MBs-run0-1
Opening results...
Sleeping before next test...
All tests complete!
```

6



5. Overriding Attributes with Command Line Arguments

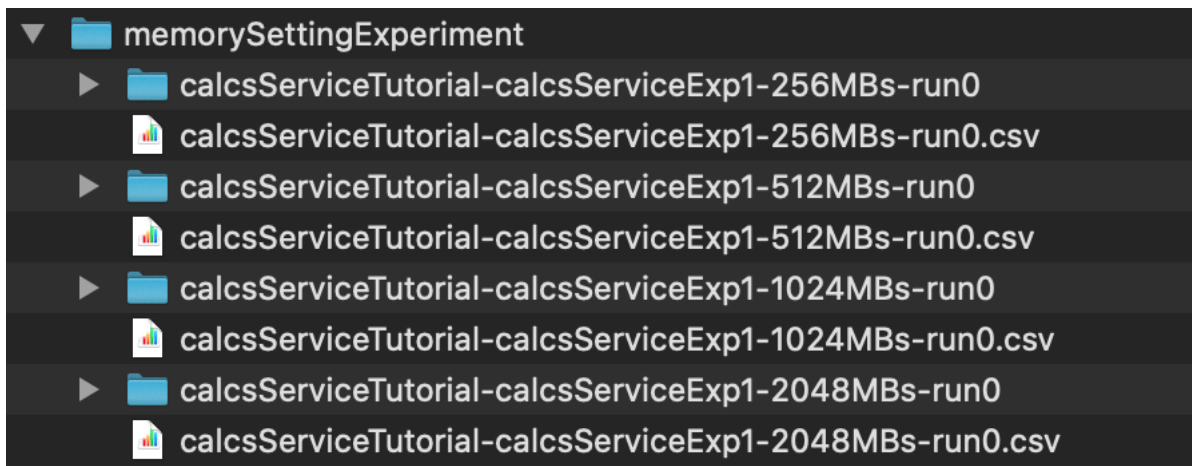
Next, let's create a more complex experiment with CalcsService. We will use the same experiment and function files but override attributes using command line arguments. Any attribute in function or experiment files can be defined through command line arguments.

For this experiment we are going to use the same workload but repeat it with different memory settings. FaaS Runner can automatically reconfigure memory settings on all supported platforms. This experiment will take a couple minutes.

```
mkdir memorySettingExperiment

./faas_runner.py -f ./functions/calcsService.json -e ./experiments/calcsServiceExp1.json
--memorySettings [256, 512, 1024, 2048] --openCSV false -o memorySettingExperiment
```

This is the most complex experiment yet so let's see what is going on. We are defining the same function and experiment files (denoted with the `-f` and `-e` flags). Then we are overriding the experiment file's `memorySettings` attribute. Overriding attributes can be done by simply using the attribute name as a flag with `--` at the start. The `memorySettings` attribute is expected to be a list of memory settings you want to use. In this case we are using 256 MBs, 512 MBs, 1024 MBs, and 2048 MBs. Next we are overriding the `openCSV` attribute to be false. For larger experiments it can be annoying having many CSV files automatically opened so we will retrieve this information later. Finally, we define the output path by using the `-o` flag to be our newly created `memorySettingExperiment` folder. The order of command line arguments does not matter.



Just like with the first experiment, if we open the output folder, we can now see CSV reports and folders of JSON files for each memory setting.

6. Creating a Unified Report

Instead of having 4 different reports for each memory setting, let's combine all the runs into one report. To do this we must first create a folder with all of the json files. This can be easily done through the command line.

```
cd memorySettingExperiment
mkdir combined
cp -r **/*.json ./combined
cd ..
```

Next we can use the **compile_results.py** script to create a single report with all 40 runs. Simply supply the path to the folder of json files (**./memorySettingExperiment/combined**) and then the path to an experiment file (**./experiments/calcsServiceExp1.json**).

```
# ./compile_results.py {FOLDER PATH} {PATH TO EXPERIMENT JSON}
./compile_results.py ./memorySettingExperiment/combined ./experiments/calcsServiceExp1.json
```

This should generate a report such as the one shown below.

2020-10-25 15:29:51.240063 - Python Partest Version 0.5																
Setting up test: runsperthead=1 threads=10 totalruns=10 payload=[{"threads": 2 "calcs": 1000 "sleep": 0 "loops": 1000 "arraySize": 1}]																
Raw results of each run:																
1_run_id	2_thread_id	arraySize	calcs	containerID	contextSwitches	contextSwitchesDelta	cpuCores	cpuIdle	cpuIdleDelta	cpuWait	cpuWaitDelta	cpuPq	cpuPqDelta	cpuKm	cpuKmDelta	cpuModel
0	8	1	1000	2020/10/25/[\$LATEST]7ee7a6c8b2be441b1835f524725d25b50	21417	79	2	491128	6	7	0	0	0	107	0	62
0	1	1	1000	2020/10/25/[\$LATEST]d8564c54d9a47daad7e70c709ad8bc	30621	76	2	944228	8	10	0	0	0	129	0	62
0	4	1	1000	2020/10/25/[\$LATEST]93ccbad03624c88ae978e93d123156	18299	820	2	287045	76	7	0	0	0	70	6	62
0	9	1	1000	2020/10/25/[\$LATEST]712080313b74fde8f5eac5404f8dd	24890	688	2	667337	134	9	0	0	0	116	7	62
0	8	1	1000	2020/10/25/[\$LATEST]297cc4d551dd4b998c946a1dba3558f8	24753	814	2	620808	147	7	0	0	0	112	7	62
0	6	1	1000	2020/10/25/[\$LATEST]Hed1a0b9f114cdca06652054ebede37	18801	89	2	344820	29	5	0	0	0	96	0	62
0	5	1	1000	2020/10/25/[\$LATEST]3d14a1f86b544b5b0402188a02b119d	18380	95	2	300838	34	7	0	0	0	87	0	62
0	2	1	1000	2020/10/25/[\$LATEST]3365e746391d47e09f7bbad2cd309a6	17451	855	2	227433	145	5	0	0	0	66	5	62
0	0	1	1000	2020/10/25/[\$LATEST]964fb496f3a24ac6bf5ef9f110f672c	17633	582	2	221853	61	7	0	0	0	93	3	62
0	8	1	1000	2020/10/25/[\$LATEST]3b0113f129c74c1db4ae323f7e6ef6d9c	17481	808	2	278458	64	7	0	0	0	88	8	62
0	3	1	1000	2020/10/25/[\$LATEST]1147ab05d7a448beb068f6a02a71f2	27236	782	2	762451	147	8	0	0	0	112	9	62
0	1	1	1000	2020/10/25/[\$LATEST]0e0a06d3b6f04d08b0b05f7a9f9cb72	21036	261	2	469084	54	7	0	0	0	86	1	62
0	0	1	1000	2020/10/25/[\$LATEST]23e89e5365c647c9b3a888d5f11c201	25304	778	2	696118	121	8	0	0	0	120	5	62
0	1	1	1000	2020/10/25/[\$LATEST]96a846e5f343a0abb042ef4dcb1a85	19425	768	2	414889	157	6	0	0	0	76	6	62
0	4	1	1000	2020/10/25/[\$LATEST]7f0ea277b2204df998fe1681393a8f3c	17021	114	2	240843	34	6	0	0	0	71	0	62
0	5	1	1000	2020/10/25/[\$LATEST]7a87d3a89a0a433bed09c60793ec12	18519	476	2	274596	64	7	0	0	0	101	3	62
0	9	1	1000	2020/10/25/[\$LATEST]e8496554451a42fb087aabdeb725568	17172	567	2	267614	65	5	0	0	0	76	2	62
0	7	1	1000	2020/10/25/[\$LATEST]8077c61601cf421cb119bb880af987b0	22818	67	2	606000	8	7	0	0	0	94	0	62
0	5	1	1000	2020/10/25/[\$LATEST]0a1af92f3aa441b18b90127adb97295	25682	932	2	786264	155	8	0	0	0	124	7	62
0	0	1	1000	2020/10/25/[\$LATEST]992dd7a71b1e4387aeeed5444b8cc2a0	17987	75	2	249841	26	9	0	0	0	92	0	62
0	6	1	1000	2020/10/25/[\$LATEST]203bc37c47734421a8f77cc0f0e5385	18849	444	2	279069	53	11	0	0	0	84	4	62
0	1	1	1000	2020/10/25/[\$LATEST]f87937a3624d01add1106a97da7d55	15430	79	2	124586	26	4	0	0	0	82	0	62
0	9	1	1000	2020/10/25/[\$LATEST]57a15eefc680465d966a3382b6db6bea	18467	86	2	305805	26	5	0	0	0	91	1	62
0	3	1	1000	2020/10/25/[\$LATEST]d0be397bb5954b7297b46d145cfe65d9	19078	579	2	316590	65	4	0	0	0	92	2	62
0	2	1	1000	2020/10/25/[\$LATEST]cecf0ddc33949f0b9eb505bf38b06fe	24169	66	2	639570	7	7	0	0	0	107	0	62
0	4	1	1000	2020/10/25/[\$LATEST]b89874dc325b46d9984da01177d38198	25709	681	2	737277	135	10	0	0	0	117	5	62
0	7	1	1000	2020/10/25/[\$LATEST]a3f00b6c89884fdabfad406eac698348	25749	813	2	722416	146	7	0	0	0	91	6	62
0	6	1	1000	2020/10/25/[\$LATEST]565db740d26b4c0fad13fa1e0ca2b423	25408	765	2	663583	154	13	0	0	0	112	5	62
0	7	1	1000	2020/10/25/[\$LATEST]ba5b9cc7ed4d46f6815dbdcddc59b668	19850	81	2	379082	30	6	0	0	0	88	0	62
0	3	1	1000	2020/10/25/[\$LATEST]bb8e53c73cea40d98f7d2772b6cc5d5de	22377	87	2	542461	11	5	0	0	0	88	0	62
0	7	1	1000	2020/10/25/[\$LATEST]399406d20e794375b630c39c490c2a	18200	342	2	328416	55	10	0	0	0	97	1	62
0	2	1	1000	2020/10/25/[\$LATEST]jacba41332ea44eeac05de93749e34a7	18053	100	2	267552	32	7	0	0	0	82	0	62
0	6	1	1000	2020/10/25/[\$LATEST]e2502bc422014b4b92e271dd0f41175f	20370	68	2	438534	7	8	0	0	0	103	0	62
0	2	1	1000	2020/10/25/[\$LATEST]72d7cd09bae433da97095316d2fc424	14959	473	2	129353	59	5	0	0	0	72	2	62
0	9	1	1000	2020/10/25/[\$LATEST]a61954c8a9a44e498ede6736801f86ed	20926	68	2	456841	8	8	0	0	0	100	0	62
0	4	1	1000	2020/10/25/[\$LATEST]2cbecc482314b47d8a255c9f9acb1d2ae3	22165	71	2	482063	8	8	0	0	0	100	0	62
0	3	1	1000	2020/10/25/[\$LATEST]a617a11dbf3345eadd334287f321e7f1	20288	90	2	418077	35	5	0	0	0	83	1	62
0	0	1	1000	2020/10/25/[\$LATEST]0aa42248ac9eac8896d3674ebd3737f	23794	73	2	570963	7	8	0	0	0	90	0	62
0	5	1	1000	2020/10/25/[\$LATEST]H421274f5d45a48e9037303692c1f1	26543	78	2	746690	9	8	0	0	0	124	0	62
0	8	1	1000	2020/10/25/[\$LATEST]731dc81a650d425fa2e03a106771651	17344	84	2	231921	24	6	0	0	0	88	0	62
Successful Runs: 40																

Now that we can regenerate reports, this gives us the ability to create experiment files dedicated to formatting a report. Let's create a new experiment file to categorize this data.

```
cd ./experiments
cp calcsServiceExp1.json report.json
nano report.json
cd ..
```

Edit the report.json file so that the ReportGenerator will create groups based on the **functionMemory** attribute.

FaaS Runner has the ability to automatically aggregate data returned by functions. By adding an attribute to the **outputGroups** attribute in an experiment file the ReportGenerator will automatically group runs with shared values together. For example if you run an experiment with 256MBs and 512MBs of memory, grouping by functionMemory will automatically calculate the average of all runs at each memory setting. By adding an attribute to **outputRawOfGroup** the ReportGenerator will simply print out the raw data of an entire group together in one block of CSV. These two attributes can be incredibly useful to get quick experiment results without having to use other tools like Excel.

```
GNU nano 2.0.6                                File: report.json                                Modified
{
  "callWithCLI": true,
  "callAsync": false,
  "memorySettings": [],
  "payloads": [{
    "threads": 2,
    "calcs": 1000,
    "sleep": 0,
    "loops": 1000,
    "arraySize": 1
  }],
  "runs": 10,
  "threads": 10,
  "iterations": 1,
  "sleepTime": 5,
  "randomSeed": 42,
  "outputGroups": ["functionMemory"],
  "outputRawOfGroup": ["functionMemory"],
  "showAsList": [],
  "showAsSum": ["newcontainer"],
  "ignoreFromAll": ["zAll", "lang", "version", "linuxVersion", "platform", "hostname"],
  "ignoreFromGroups": ["1_run_id", "2_thread_id", "cpuModel", "cpuIdle", "cpuIowait", "cpuIrq", "cpuKrn", "cpuNice", "cpuSoftIrq", "cpuUsr"],
  "ignoreByGroup": {
    "containerID": ["containerID"],
    "cpuType": ["cpuType"],
    "vmID": ["vmID"]
  },
  "invalidators": {},
  "removeDuplicateContainers": false,
  "overlapFilter": "functionName",
  "openCSV": true,
  "combineSheets": false,
  "warmupBuffer": 0
}
```

Once edited and saved, run the **report_compiler.py** script again with the newly created **report.json** file.

```
./compile_results.py ./memorySettingExperiment/combined ./experiments/report.json
```

In the report you should now see aggregated categories for functionMemory. Alongside that, the results of each run should also be consolidated together in the report.

48	Category functionMemory:									
49	functionMemory	uses	avg_arraySize	avg_calcs	avg_contextSwitches	avg_contextSwitchesDelta	avg_cpuCores	avg_cpudleDelta	avg_cpulowaitDelta	avg_cpulrqDelta
50	1024	10	1.00	1000.00	18162.10	89.30	2.00	29.60	0.00	0.00
51	2048	10	1.00	1000.00	23520.00	73.30	2.00	7.90	0.00	0.00
52	256	10	1.00	1000.00	24160.70	787.60	2.00	144.10	0.00	0.00
53	512	10	1.00	1000.00	18122.60	535.20	2.00	61.60	0.00	0.00
54	Total number of unique functionMemorys: 4									
55										
56	--- Runs of Group functionMemory ---									
57										
58	Category functionMemory with 1024:									
59	arraySize	calcs	containerID	contextSwitches	contextSwitchesDelta	cpuCores	cpudleDelta	cpulowaitDelta	cpulrqDelta	cpuKrnDelta
60	1	1000	2020/10/25/\$LATE:	18801	89	2	29	0	0	0
61	1	1000	2020/10/25/\$LATE:	18380	95	2	34	0	0	0
62	1	1000	2020/10/25/\$LATE:	17021	114	2	34	0	0	0
63	1	1000	2020/10/25/\$LATE:	17987	75	2	26	0	0	0
64	1	1000	2020/10/25/\$LATE:	15430	79	2	26	0	0	0
65	1	1000	2020/10/25/\$LATE:	18467	86	2	26	0	0	1
66	1	1000	2020/10/25/\$LATE:	19850	81	2	30	0	0	0
67	1	1000	2020/10/25/\$LATE:	18053	100	2	32	0	0	0
68	1	1000	2020/10/25/\$LATE:	20288	90	2	35	0	0	1
69	1	1000	2020/10/25/\$LATE:	17344	84	2	24	0	0	0
70										
71	Category functionMemory with 2048:									
72	arraySize	calcs	containerID	contextSwitches	contextSwitchesDelta	cpuCores	cpudleDelta	cpulowaitDelta	cpulrqDelta	cpuKrnDelta
73	1	1000	2020/10/25/\$LATE:	21417	79	2	6	0	0	0
74	1	1000	2020/10/25/\$LATE:	30621	76	2	8	0	0	0
75	1	1000	2020/10/25/\$LATE:	22818	67	2	8	0	0	0
76	1	1000	2020/10/25/\$LATE:	24169	66	2	7	0	0	0
77	1	1000	2020/10/25/\$LATE:	22377	87	2	11	0	0	0
78	1	1000	2020/10/25/\$LATE:	20370	68	2	7	0	0	0

7. Creating Complex Experiments with Scripts

Now that you have the ability to run multiple experiments and combine the results together into one report, we can create even more complex experiments with FaaS Runner. For the most complex experiments it is best to create a script that then invokes FaaS Runner. We can leverage many features of FaaS Runner to improve this process.

For this experiment we will use all the features of our calcsService application. CalcsService is a CPU bound workload that does random math ($a * b / c$). The amount of calculations can be defined using the calcs attribute. For our next experiment we want to add variability to our runtime and measure how runtime changes as the number of calculations increases or decreases. To do this we can add many payloads to the list of **payloads** attribute and FaaS Runner will distribute them between function invocations. Here we will use payload inheritance to define a single **parentPayload** that will contain attributes that all function invocations will use. Then the values in the **payloads** list will override the values in the **parentPayload** if there are conflicts.

Next, we want to measure the impact of the FaaS freeze/thaw lifecycle. After a memory value is changed all infrastructure allocated to the function will be destroyed, entering the function into a luke-warm state where the application code is cached but infrastructure must be reallocated. To fully achieve the “cold” state we must wait around 45 minutes. To measure the impact of the “luke-warm” state we simply need to run an experiment a second time after getting to the luke-warm state. So one experiment run is in the luke-warm state and the next will be warm. This same methodology can apply when comparing cold to warm states. To run an experiment twice in succession we can use FaaS Runner’s **iterations** attribute.

On AWS Lambda, the CPU allocated to a function varies in performance depending on the memory setting assigned to a function. At low settings (<256MBs) a function may have allocated 1/10 of a single CPU core up to over 2 CPU cores after 1536MBs. We can measure this performance variability by executing an experiment across multiple memory settings. Like we did in the previous experiment, we can define multiple memory settings using the **memorySettings** attribute in FaaS Runner.

Finally, the calcsService application has the feature to produce memory stress by setting the **arraySize** attribute to a large number. When doing random math ($a * b / c$), calcsService does not use primitive variables (e.g. int/double) but instead creates arrays and accesses the random numbers from those arrays. By setting arraySize to be a large number (e.g. 1,000,000) we create memory stress in two ways. First, creating large arrays requires allocating and freeing large amounts of memory. Second, the numbers to do math with are assigned and read from random indices in the arrays. Reading and writing to random positions in memory can greatly impact performance as it can cause something called page faults. Pieces of memory are frequently cached in different levels of memory (e.g. L1/L2/L3) so when an application reads something that is not cached a slowdown occurs. We can measure the memory performance of AWS Lambda by running the experiment once without memory stress (arraySize = 1) and with memory stress (arraySize = 1000000).

Here is the summarized process of what we want the experiment to do:

1. Vary the number of calculations (calcs) calcsService does between 1,000 and 100,000 in steps of 1,000 in each experiment run.
2. Repeat the experiment a second time to measure cold/warm performance.
3. Change memory setting between 256 MBs, 1024 MBs and 2048 MBs.
4. Repeat all the steps once again with memory stress (arraySize = 1,000,000).

We can create a bash script to easily create FaaS Runner arguments and execute this experiment. The script is included below, see highlighted sections for the implementation of each step. Review comments to see what arguments are being defined. Save and execute this script as **complexTest.sh** in the **test** directory. This experiment will take a few minutes to complete.

```
#!/bin/bash

# FaaS Runner Complex Experiment Example
# @author Robert Cordingly

# Define Experiment Arguments
args="--function calcsServiceTutorial --runs 100 --threads 100 --warmupBuffer 0 --combineSheets 0 --sleepTime 0
--openCSV 0 --iterations 2 --memorySettings [256, 1024, 2048]"

# Create parent payload.
parentPayloadNoMemory="{\"threads\":2,\"sleep\":0,\"loops\":1000,\"arraySize\":1}"
parentPayloadMemory="{\"threads\":2,\"sleep\":0,\"loops\":1000,\"arraySize\":1000000}"

# Generate scaling number of calcs payloads.
#
# This creates a list of payloads like this:
# [{"calcs":1000}, {"calcs":2000}, ..., {"calcs":99000}, {"calcs":100000}]
start=1000
step=1000
end=100000
payloads "["
for calcs in $(seq $start $step $end)
do
    payloads="$payloads{\"calcs\":$calcs}"
    if [ "$calcs" -lt "$end" ]
    then
        payloads="$payloads,"
    else
        payloads="$payloads]"
    fi
done

# Created Payloads List:
echo "Created Payloads List:"
echo $payloads

# Create Output Folders
mkdir complexExperiment
mkdir complexExperiment/NoMemory
mkdir complexExperiment/Memory

# Run Experiments with and without Memory Stress
./faas_runner.py -o ./complexExperiment/NoMemory --payloads $payloads --parentPayload $parentPayloadNoMemory $args
./faas_runner.py -o ./complexExperiment/Memory --payloads $payloads --parentPayload $parentPayloadMemory $args

echo "Experiments Done!"
```

This script leverages FaaS Runner's payload inheritance. We first create a **parentPayload** that contains attributes that all function invocations in an experiment will use. In this case we create two parents, one with memory stress and one without. Then we create the **payloads** attribute to vary the number of calculations. This list of payloads will be distributed randomly between the threads. Finally, we define all other attributes in the **args** variable. This script also creates a few folders to keep our output organized. Unlike previous experiments, this experiment does not use any experiment or function files. Everything is defined through command line arguments and makes use of FaaS Runner's default parameters. For example, by default FaaS Runner assumes you are using AWS Lambda. Save and execute this script as **complexTest.sh** in the **test** directory. This experiment will take a few minutes to complete.

```
# Run the Experiment
cd ./test
./complexTest.sh
```

Task 1: Create a single report with all data from the complex experiment. In your **report.json** file add "newcontainer" and "arraySize" to the **outputGroups** list just like you did in section 6 for **functionMemory**. Copy all json files from both the NoMemory and Memory folders into one combined folder. Run the **report_compiler.py** script on the folder to generate the report.

FaaS Runner can aggregate data for any attribute returned by a function. For all data returned by SAAF and their definitions see: https://github.com/wlloyduw/SAAF/tree/master/java_template

For all FaaS Runner experiment execution, data aggregation, and report generation options see: <https://github.com/wlloyduw/SAAF/tree/master/test>

```
# Create a single report.
cd ./complexExperiment
mkdir combined
find . | grep json | xargs -I{} -n1 cp '{}' ./combined/
cd ..
./compile_results.py ./complexExperiment/combined ./experiments/report.json
```

Task Questions:

1. Read the report and scroll down to the aggregated results for **newcontainer**, what was the impact on the avg_userRuntime column of the cold (newcontainer = 1) versus warm (newcontainer = 0)?
2. What was the impact of memory stress on average runtime? Look at the aggregated results for **arraySize**.
3. What was the impact of different memory settings on average runtime? Look at the aggregated results for **functionMemory**.

8. Using FaaS Runner with Function Pipelines

Alongside running individual functions, FaaS Runner can execute complex pipelines of functions. To begin we must first explain the syntax. To execute a pipeline, you must define lists of **functions** and **experiments**. Like with single function calls, both functions and experiments can be defined through either files or command line arguments. For these examples we will use both.

Using the included function and experiment files. Try executing this experiment:

```
./faas_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json  
-e ./experiments/jello.json ./experiments/pello.json ./experiments/nello.json
```

Now let's explain what happened. The first experiment, in this case **jello.json**, is considered our parent experiment. This experiment file defines how many runs are going to be executed, the number of threads, and will be used to generate the report. In this case, this experiment file says that there will be 3 runs with 1 thread. In our output we saw a total of 9 function calls. For pipelines, the number of runs are runs of the entire pipeline. 1 Threads means that the pipeline was called sequentially so we saw responses come back in the expected order of Jello, Pello, Nello, Jello, Pello, Nello, etc. If we chose 3 threads, then 3 instances of the pipeline would run concurrently.

payload	message
{"name": "Jello"}	Jello Jello
{"name": "Pello"}	Pello Pello
{"name": "Nello"}	Nello Nello
{"name": "Jello"}	Jello Jello
{"name": "Pello"}	Pello Pello
{"name": "Nello"}	Nello Nello
{"name": "Jello"}	Jello Jello
{"name": "Pello"}	Pello Pello
{"name": "Nello"}	Nello Nello

Now take a look at the message and payload column of each function:

Since each experiment file defined payloads for the function, those payloads were used in the function invocation.

Instead of supplying a specific set of payloads to each function in the pipeline it may be necessary to pass the results from one function invocation to another.

Let's try and pass the response message from each function to the next, resulting in a final message of "Nello Pello Jello Jello"

9. Command Line Arguments and Passing Attributes in a Pipeline

FaaS Runner has the built-in attribute **passPayloads** that does just that! By default, this attribute is false so we can override that with command line arguments just like with single function experiments. Run the same experiment again but add the "**--passPayloads true**" flag.

```
./faas_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json -e
./experiments/jello.json ./experiments/pello.json ./experiments/nello.json --passPayloads
true
```

payload	message
{"name": "Jello"}	Jello Jello
{"cpuType": "Intel(R) Xeon(R) Proce	Pello Pello
{"version": "0.5"; "lang": "python";	Nello Nello
{"name": "Jello"}	Jello Jello
{"cpuType": "Intel(R) Xeon(R) Proce	Pello Pello
{"version": "0.5"; "lang": "python";	Nello Nello
{"name": "Jello"}	Jello Jello
{"cpuType": "Intel(R) Xeon(R) Proce	Pello Pello
{"version": "0.5"; "lang": "python";	Nello Nello

As we can see now ALL attributes returned by previous functions are passed onto the payload of the next function invocation. But our message response is still unchanged.

This is because the Hello World functions expects an attribute called "name" as the input and returns the response in the "message" attribute. Between function invocations we need to rename "message" to "name" to get the desired output we want.

To do this, we can use FaaS Runner's **transitions** attribute. This attribute expects a JSON object of key value pairs that will rename one attribute to another between function invocations. Like we did with **passPayloads** we can define this through command line arguments:

```
./faas_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json -e
./experiments/jello.json ./experiments/pello.json ./experiments/nello.json --passPayloads
true --transitions {"message": "name"}
```

FaaS Runner is passing all attributes from the response of one function into the request of the next. While it does that, it renames the "message" attribute to "name" as defined by the transition attribute.

By default, when a command line argument is used to override something it applies it to ALL experiment/function files. If you want to only apply an argument to one specific function in the pipeline you can add array-style indexes to the argument (starting at 0). For example, if we want to do the same experiment but only pass arguments from the first function to the second, we can apply **passPayloads** only to the second function:

```
./faas_runner.py -f ./functions/jello.json ./functions/pello.json ./functions/nello.json -e
./experiments/jello.json ./experiments/pello.json ./experiments/nello.json
--passPayloads[1] true --transitions {"message": "name"}
```

This syntax can be applied to any attribute. If you want to have specific transitions between functions in a pipeline you can define that this way. This also allows complete pipelines to be entirely defined through command line arguments. For example, the same pipeline can be executed without using function or experiment files:

```
./faas_runner.py --function[0] jelloWorld --function[1] pelloWorld --function[2] nelloWorld
--runs 3 --threads 1 --payloads [{\"name\": \"Jello\"}] --passPayloads true --transitions
{\"message\": \"name\"}
```

10. Dynamic Pipelines and State Machines

For the most complex pipelines, FaaS Runner can be used to orchestrate function execution by modifying `test/tools/pipeline_transition.py`.

```
def transition_function(index, functions, experiments, payloads, lastPayload):
    return (index + 1, functions, experiments, payloads, lastPayload)
```

This is the default transition function, after each execution, increment the index to go to the next function; leaving the functions, experiments, and payloads unchanged. To better understand what data is being passed through here, add a few comments to `pipeline_transition.py` and rerun the previous pipeline:

```
def transition_function(index, functions, experiments, payloads, lastPayload):

    print("----- INDEX -----")
    print(str(index))
    print("----- FUNCTIONS -----")
    print(str(functions))
    print("----- EXPERIMENTS -----")
    print(str(experiments))
    print("----- PAYLOADS -----")
    print(str(payloads))
    print("----- LAST PAYLOAD -----")
    print(str(lastPayload))
    print("-----")

    return (index + 1, functions, experiments, payloads, lastPayload)
```

Task 2: Using the experiment defined below, create a transition function that skips the 2nd function (pelloWorld) if the first function returns a message of "Jello End" otherwise execute the pipeline normally.

```
./faas_runner.py --function[0] jelloWorld --function[1] pelloWorld --function[2] nelloWorld
--runs 10 --threads 1 --payloads [{\"name\": \"Jello\"}, {\"name\": \"End\"}] --passPayloads
true --transitions {\"message\": \"name\"} --shufflePayloads true
```

Spoiler: Solution. Copy and Paste text from box below to reveal.

