Observations about Serverless Computing

With a few examples from AWS Lambda, Azure Functions and Open Whisk.

Dennis Gannon
School of Informatics and Computing
Indiana University

Cloud computing is going through an interesting evolution. It has gone from a platform for deploying virtual machines to planet-scale systems with extensive collections of data storage, analysis and machine learning services. Most recently we have seen the emergence of "cloud native" computing, which in its most basic form involves a design pattern of microservices where big applications are decomposed into hundreds of basic stateless components that run on clusters managed by tools like Kubernetes, Mesos and Swarm.

Serverless computing is the next step in this evolution. It addresses the following challenges. Suppose I have a small computation that I want to run against some database at the end of each month. Or suppose I want to have the equivalent of a computational daemon that wakes up and executes a specific task only when certain conditions arise. For example, when an event arrives in a particular stream or when a file in a storage container has been modified or when a timer goes off. How do I automate these tasks without paying for a continuously running server? Unfortunately, the traditional cloud computing infrastructure model would require me to allocate computing resources such as virtual machines or a microservice cluster and my daemon would be a continuously running process. While I can scale my cluster of VMs up and down, I can't scale it to zero without my daemon becoming unresponsive. I only want to pay for my computing WHEN my computation is running.

This is not a totally new idea. Paying only for the compute that we use goes back to early timesharing and persists with compute-hour "allocations" on supercomputers today. And there are cloud services such as Azure Data Lake Analytics, Amazon Kinesis or the AWS API gateway, that charge you only for the computation that you use or data that you move and they do not require you to deploy server infrastructure to use them.

However, there is something deeper going on here and it has to do with triggers and another computational paradigm called "Function-as-a-Service" (FaaS). Unlike the serverless examples above which depend upon me invoking a specific well-defined service, FaaS allows a cloud user to define their own function, and then "register" it with the cloud and specify the exact events that will cause it to wake up and execute. As mentioned above, and illustrated in Figure 1, these event triggers can be tied to changes in state of a storage account or database, events associated with queues or streams of data from IoT devices, web API invocations coming from mobile apps. Triggers can even be defined by steps in the execution of a workflow. And, of course, the user only pays when and while the function is executing.

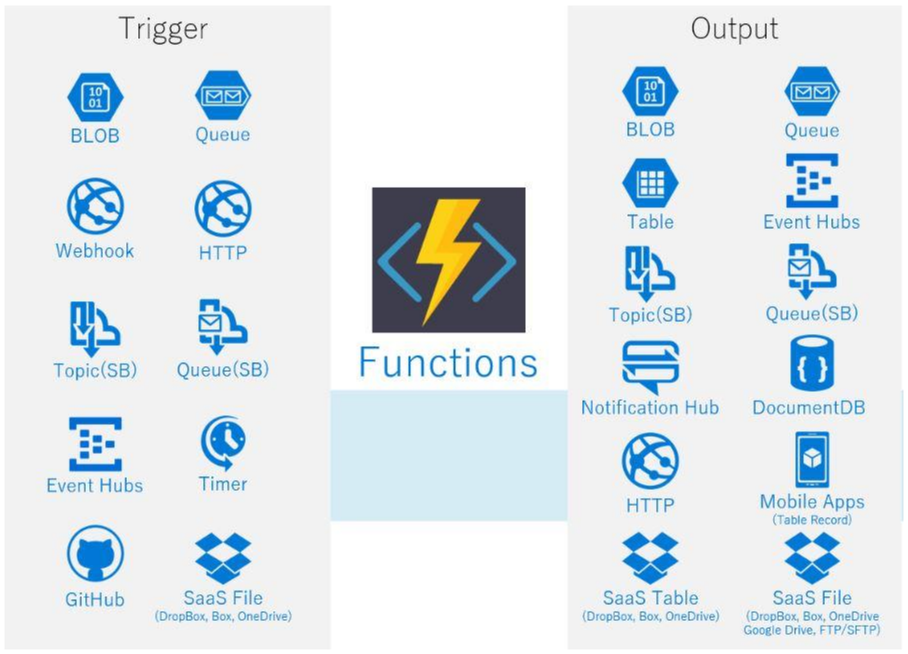


Figure 1. Function, triggers and output concept. (from markoinsights.com)

There have been two conferences that have focused on the state of serverless computing. The paper "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research" by Geoffrey Fox, Vatche Ishakian, Vinod Muthusamy and Aleksander Slominski provides an excellent overview of many of the ideas, concepts and questions that surround serverless computing that surfaced at these conferences. In that paper, they refer to an IBM tutorial that defines serverless FaaS as

1. short-running, stateless computation
2. event-driven applications
3. scales up and down instantly and automatically
4. based on charge-by-use

Notice that there is a distinction between FaaS and serverless FaaS and it has to do with item 4 above. A good example of this is Google's App Engine, which was arguably the first FaaS available from a commercial cloud. In its current form App Engine can run in one of two modes. In its standard mode, your applications run in a sandbox and you are charged only when the app is running. In the "flexible" mode you deploy a container and then specify the compute infrastructure needed in terms of CPU power, memory, disk and you are charged by the hour. You could say that App Engine running in Flexible mode is server-lite, but clearly not fully serverless, while standard mode is truly serverless.

What are the serverless FaaS choices?

There are a number of FaaS implementations. Some of these are used for research while others are commercial products. The Status report refers to many of these and the slides for the workshop are on-line. A good example of the research work is OpenLambda from the University of Wisconsin and first introduced at HotCloud '16. Based on this experience the Wisconsin team described Pipsqueak, an experiment to reduce the deployment latencies caused by Python library initializations. Ryan Chard described Ripple which is an excellent example of distributing event trigger management from the source to the cloud. Ripple has been designed and use for several significant science applications such as beamline science (ALS and APS). Another related technology is if-this-then-that IFTTT that is a service for chaining together other service.

Two other open source projects raise an interesting question about what is behind the curtain of serverless. Funktion and Fission are both implementations of FaaS on top of Kubernetes. As we discuss serverless computing we must remember that there is a "server" somewhere. The basic infrastructure for serverless computing needs to run somewhere as a persistent service and hence a microservice platform like Kubernetes is a reasonable choice. This relates to the economics of severless computing and we return to that at the end of this report.

The most commonly referenced open source FaaS service is Apache OpenWhisk which was developed by IBM and is available on their Bluemix cloud as a service. The other commercial services include Google Function, Microsoft Azure Functions and Amazon Lambda. At the end of this article we will show some very simple examples of using some of these systems.

When can FaaS replace a mesh of microservices for building an app?

The Status paper also makes several other important observations. For example, they note that while serverless is great for the triggered example described above, it is not good for long-running or statefull applications like databases, deep learning training, heavy stream analytics, Spark or Hadoop analysis and video streaming. In fact, many large-scale cloud-native applications that have thousands of concurrent users require continuous use of massive networks of microservices. These apps will not be based on serverless FaaS. However, there may be many cases where a user-facing application running in the cloud could be easily implemented with serverless FaaS rather than as a big microserivce deployment. What we do not know is where the cross-over point from a serverless FaaS implementation of an app to a full kubernetes based massive microservice deployment lies. This relates to the economic of FaaS (discussed briefly at the end of this article). The Cloud Native Computing Foundation has a working group on serverless computing that is addressing this topic.

There is one interesting example of using serverless FaaS to do massively parallel computing and it called pywren. The lovely paper "Occupy the Cloud: Distributed Computing for the 99%" by Eric

Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica and Benjamin Recht describes the concepts in pywren which allow it to scale computation to thousands of concurrent function invocations achieving 40 teraflops of compute performance. Pywren uses AWS Lambda in a very clever way: it serializes the computational functions which are then passed to a lambda function to execute. We will return to Pywren in another post.

Computing at the Edge?

Perhaps the hottest topic in the general area of cloud computing is when the computing spills out of the cloud to the edge of the network. There are many reasons for this but most boil down to latency and bandwidth. A good example is the use cases that motivate the Ripple system described above. In order to generate a stream of events from sensors at the edge of the network, one needs very light weight computing that can monitor them and generate the events. In many cases it is necessary to preprocess the data in order to send the message to the cloud where a function will be invoked to respond. In some cases, the response must return to the source sooner than a remote invocation can respond because of the latencies involved. The computing at the edge may need to execute the function there and the actual communication with the cloud may be just a final log event or a trigger for some follow-up action.

Another possibility is that the functions can migrate to the places where they are needed. When you deploy the computing at the edge you also deploy a list of functions that that must be invoked. When the edge process starts up it could cache some of the function it needs locally. We expect there will be many variations on these ideas in the future.

A Tiny FaaS Tutorial

By their very nature FaaS systems are very simple to use: you write short, stateless function and tie them to triggers. We will take a quick look at three of these: AWS Lambda, Microsoft Function and IBM Bluemix OpenWhisk.

AWS Lambda Functions

Amazon was the first to introduce serverless functions as a service and it is very well integrated into their ecosystem of other services. Called Lambda functions, they can be easily created in a number of standard programming languages. Lambda functions can be associated with a variety of trigger events including changes to the state of a storage account, web service invocations, stream events and even workflow events.

We will illustrate Lambda with a simple example of a function that responds to Kinesis Stream events and for each event it adds an item to a dynamoDB table. Here is the python code for the function that accomplishes this task.

```
from __future__ import print_function
#import json
import base64
import boto3
def lambda_handler(event, context):
    dyndb = boto3.resource('dynamodb', region_name='us-west-2')
    table = dyndb.Table("lambdaTable")

    for record in event['Records']:
       #Kinesis data is base64 encoded so decode here
       payload=base64.b64decode(record["kinesis"]["data"])
      x = eval(str(payload))
      metadata_item = {'PartitionKey': x['PartitionKey'], 'RowKey':
           x['RowKey'], 'title': x['text']}
       table.put_item(Item=metadata_item)
```

There are several critical items that are not explicit here. We need to invoke the AWS **Identity and Access Management (IAM)** system to delegate some pemissions to our function and we will need to grab copies of the Amazon Resource Names (ARNs) for this and other objects. First we create a IAM role that will allow access to Kineses streams and DynamoDB. Using the IAM portal we created a role called "lambda-kinesis-execution-role" and
attached two policies:

1. 'AmazonDynamoDBFullAccess"
2. "AWSLambdaKinesisExecutionRole".

We then made a copy of the Role ARN.

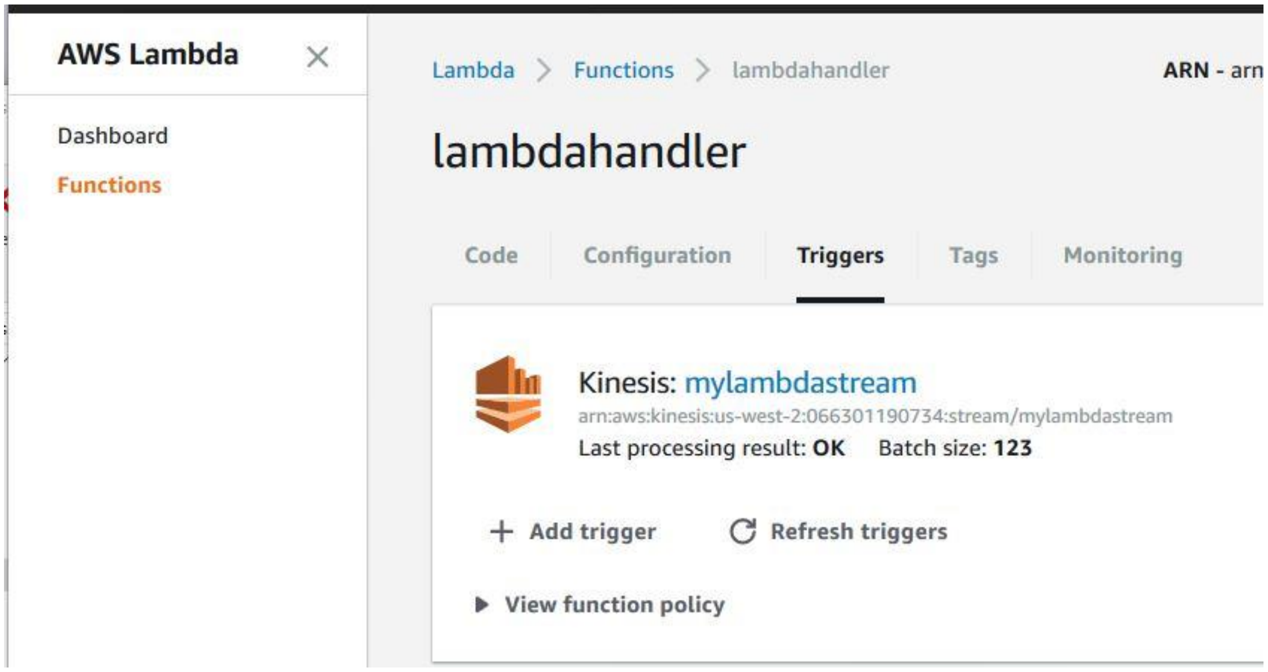
The next step is to install this function into the lambda system. To do that we put the function above into a file called "ProcessKinesisRecords.py" and then zipped it. We then uploaded the zipped file to S3 (in the us-west-2) in the bucket "dbglambda". With that we can create our function with the boto3 call from our laptop as:

```
lambdaclient = boto3.client('lambda')

response = lambdaclient.create_function(
    FunctionName='lambdahandler',
    Runtime='python2.7',
    Role='arn:aws:iam::xx:role/lambda-kinesis-execution-role',
    Handler='ProcessKinesisRecords.lambda_handler',
    Code={
        'S3Bucket': 'dbglambda',
        'S3Key': 'ProcessKinesisRecords.zip',
       }
    )
```

The final item we need is the link between this function and the Kinesis Stream service. To do that we went to the portal for Kinesis and created a stream called "mylambdastream" and grabbed its ARN. Creating the binding is accomplished with the following.

```
response = lambdaclient.create_event_source_mapping(
    EventSourceArn=
      'arn:aws:kinesis:us-west2:0xxx34:stream/mylambdastream',
    FunctionName='lambdahandler',
    BatchSize=123,
    StartingPosition='TRIM_HORIZON',
)
```

We can verify the properties of the function we have created by looking at the AWS Lambda portal pages. As shown below, we can verify that our lambdahandler function does indeed have our stream as its trigger.



Kinesis: mylambdastream
arn:aws:kinesis:us-west-2:066301190734:stream/mylambdastream
Last processing result: **OK** Batch size: **123**

Figure 2. AWS Lambda Portal showing the trigger for our function

Finally we can invoke it by pushing an event to the kinesis stream. This is shown below.

```
client = boto3.client('kinesis')
record = "{'PartitionKey':'part2', 'RowKey': '444',
            'text':'good day'}"
resp = client.put_record(StreamName='mylambdastream',
          Data=bytearray(record), PartitionKey='a')
```

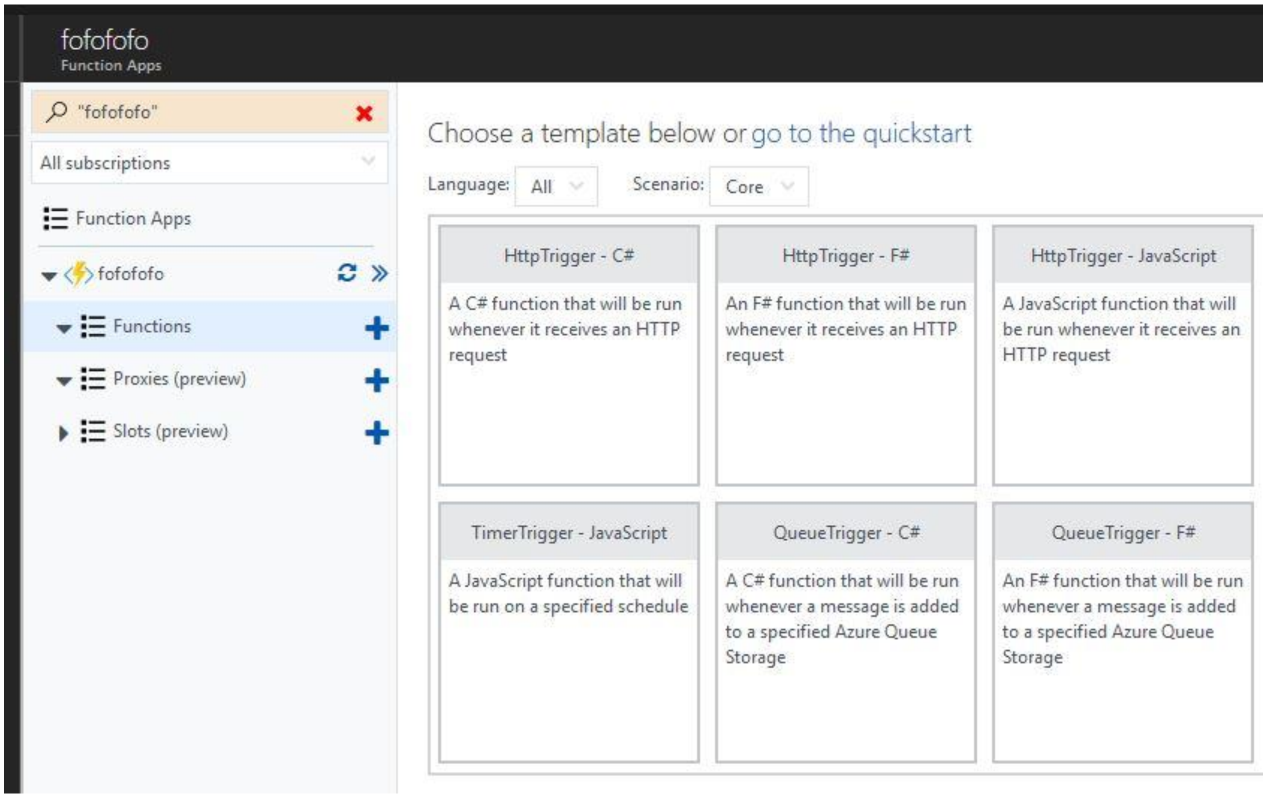
Checking the DynamoDB portal will verify that the function has picked the message from Kinesis and deposited it in the database. The full details are in the notebook "simple-kinesis-lambda-sender.ipynb".

Azure Functions

Microsoft has released Azure Functions as their entry in the serverless function domain. The concept and operation are very similar to AWS Lambda, but their support of multiple languages is not as mature as that of Lambda. In particular, the support for Python is still evolving.

The simple case of a webservice function.

The Azure function portal has a large number of basic templates we can use to build our function as shown below.



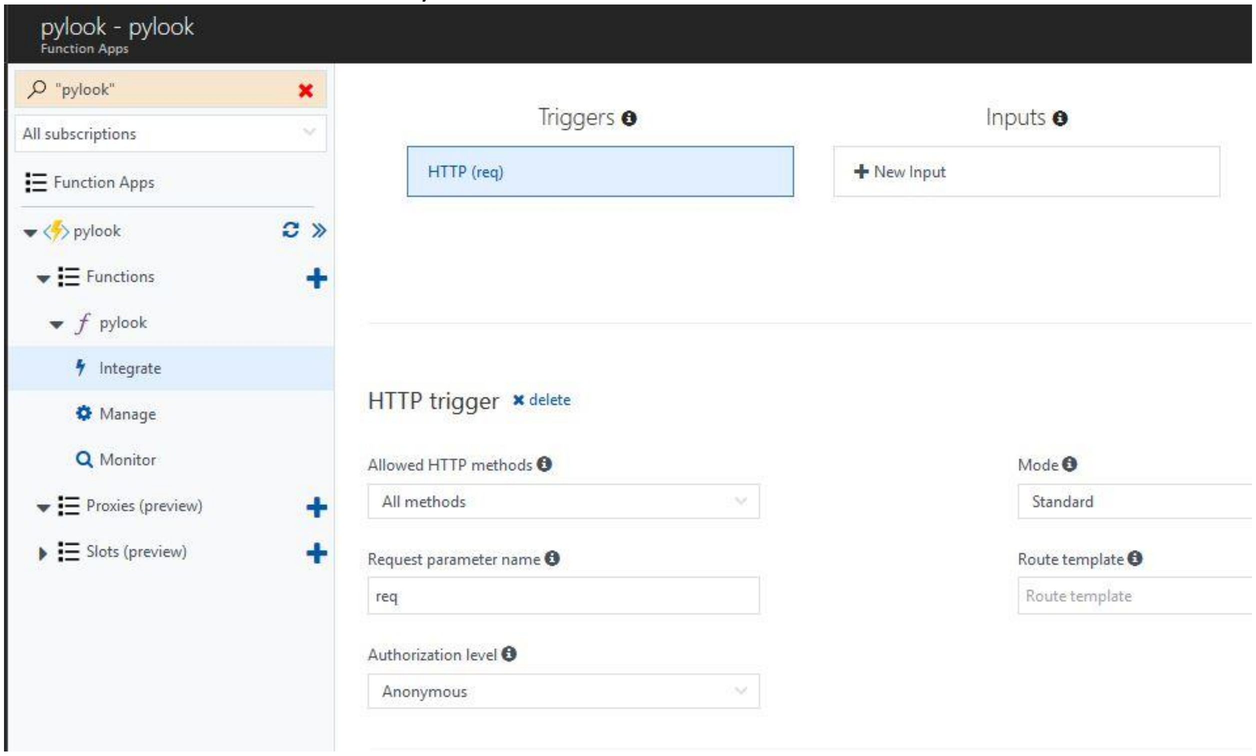
HttpTrigger - C#	HttpTrigger - F#	HttpTrigger - JavaScript
A C# function that will be run whenever it receives an HTTP request	An F# function that will be run whenever it receives an HTTP request	A JavaScript function that will be run whenever it receives an HTTP request
TimerTrigger - JavaScript	QueueTrigger - C#	QueueTrigger - F#
A JavaScript function that will be run on a specified schedule	A C# function that will be run whenever a message is added to a specified Azure Queue Storage	An F# function that will be run whenever a message is added to a specified Azure Queue Storage

Figure 3. Azure Fuction template selection page.

We have selected one of the Python examples that creates a simple web service. Bringing this up on the portal we see the code below.

```
import os
import json
postreqdata = json.loads(open(os.environ['req']).read())
response = open(os.environ['res'], 'w')
response.write("hello world to "+postreqdata['name'])
response.close()
```

To get this running, we now must go to the "integrate" page to provide one more detail: we set the authorization level to "anonymous" as shown below.



HTTP trigger

Allowed HTTP methods: All methods

Mode: Standard

Request parameter name: req

Route template: Route template

Authorization level: Anonymous

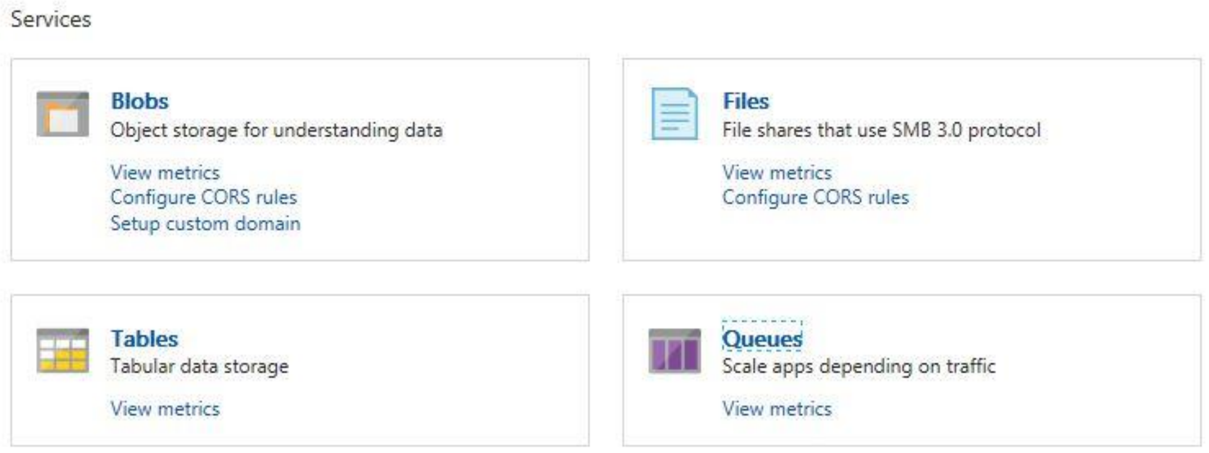
Figure 4. Setting the authorization level for the trigger to “anonymous”.

The client is also simple and shown below.

```
import os
import json
import requests
import json

data = {}
data['name'] = 'Dr. Jones'
json_data = json.dumps(data)
r = requests.post("https://pylook.azurewebsites.net/api/pylook",
                    data=json_data)
print(r.status_code, r.reason)
print r.text
```

Using Azure Functions to pull data from a queue.

We now give another version of the function that reads messages from a queue and puts them in a table. There is no Python template for this one yet, so we will use JavaScript. To use template this we must first create a new storage account or use an existing one. Go to the storage account page and you will see:


Services

Blobs
Object storage for understanding data
View metrics
Configure CORS rules
Setup custom domain

Files
File shares that use SMB 3.0 protocol
View metrics
Configure CORS rules

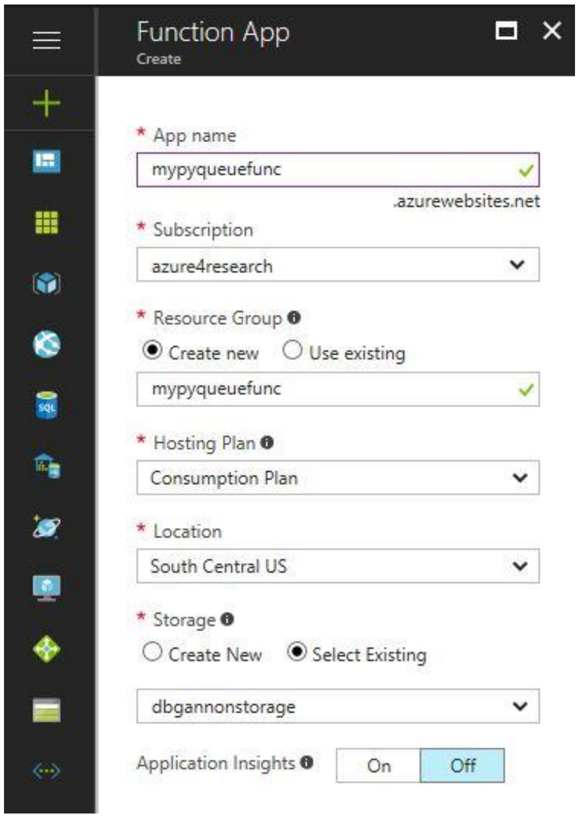
Tables
Tabular data storage
View metrics

Queues
Scale apps depending on traffic
View metrics

Figure 5. Creating a new table and queue in a storage account

We click on Table and create a new table and remember its name. Then we go to Queues and create a new one and remember its name. Looking in the storage explorer should show for these items. Clicking on the table name should give us a picture of an empty table.

Go back to the portal main page and click + and look for "Function App" and click create. There is a table to fill in like the one below.


Function App
Create

* App name: mypyqueuefunc .azurewebsites.net

* Subscription: azure4research

* Resource Group: (●) Create new ( ) Use existing
mypyqueuefunc

* Hosting Plan: Consumption Plan

* Location: South Central US

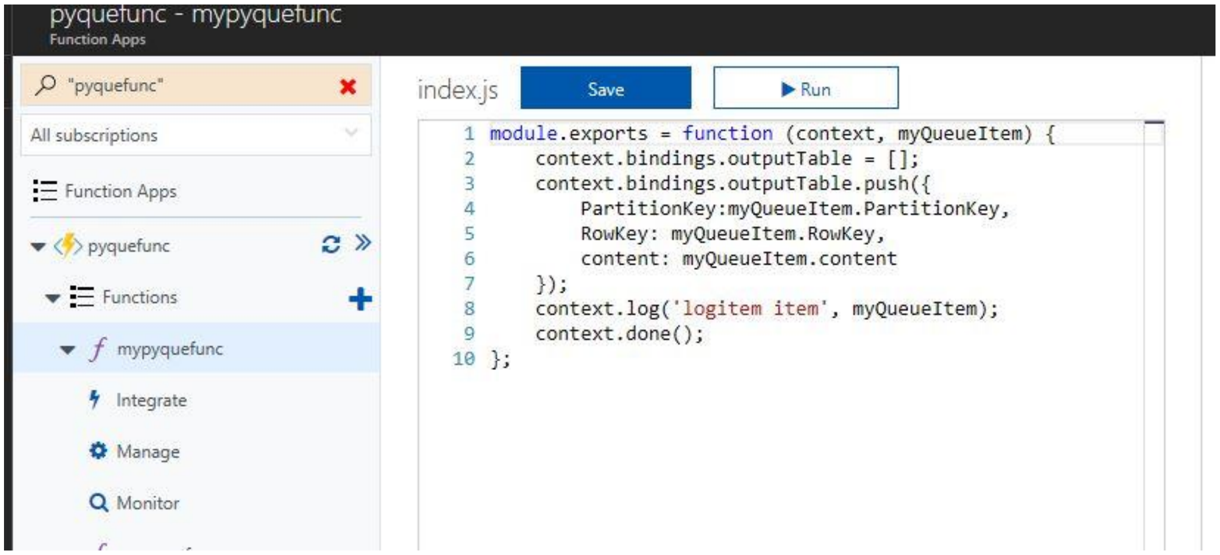
* Storage: ( ) Create New (●) Select Existing
dbgannonstorage

Application Insights: On [✓ Off]

Figure 6. Creating a new function.

We give it a name and allow it to create a new resource group. For the storage we want to use the dropdown and look for the storage account name. (It is important that the storage account is in the same location as the function.) We click "create" and wait for the function to appear on the function portal page. You should see it in your resource groups. Follow that link and you will see that it is running.

Go to the functions tab and hit "+". It will ask you to pick one of the templates. At the top where it says "language" select Javascript and pick the one that is called QueueTrigger. This will load a basic template for the function. Now edit the template so that it looks like the following example.



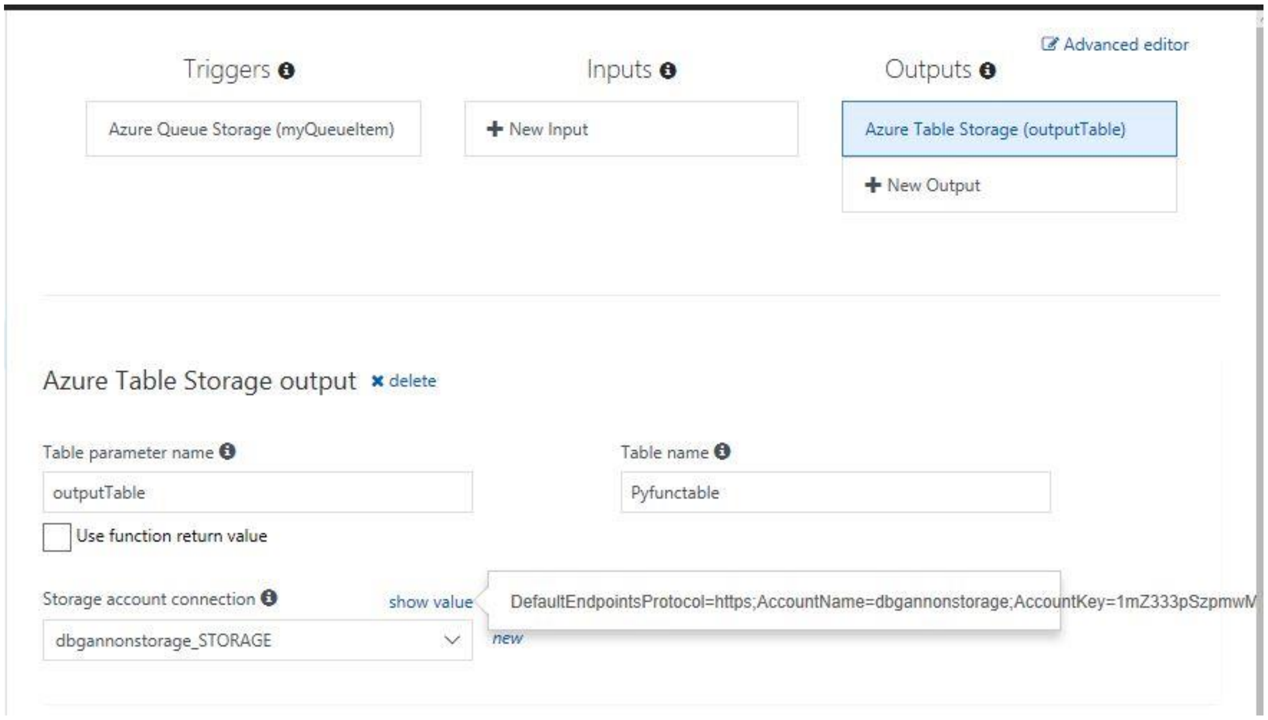
```
module.exports = function (context, myQueueItem) {
    context.bindings.outputTable = [];
    context.bindings.outputTable.push({
        PartitionKey:myQueueItem.PartitionKey,
        RowKey: myQueueItem.RowKey,
        content: myQueueItem.content
    });
    context.log('logitem item', myQueueItem);
    context.done();
};
```

Figure 7. The Function text from the Azure portal.

The main difference between this and the template is that we have added an output table and instructions to push three items into the table. The function is assuming that the items in the queue are of the form

```
{'PartitionKey':'part1', 'RowKey':'73','content':'some data '}
```

Next we need to tie the queue to our queue and the table to our table. So click "Integrate" on the left and fill in the form so that it ties it to your resources as illustrated below.



Azure Queue Storage (myQueueItem)

Azure Table Storage (outputTable)

Azure Table Storage output

Table parameter name: outputTable

Table name: Pyfunctable

Use function return value

Storage account connection: dbgannonstorage_STORAGE

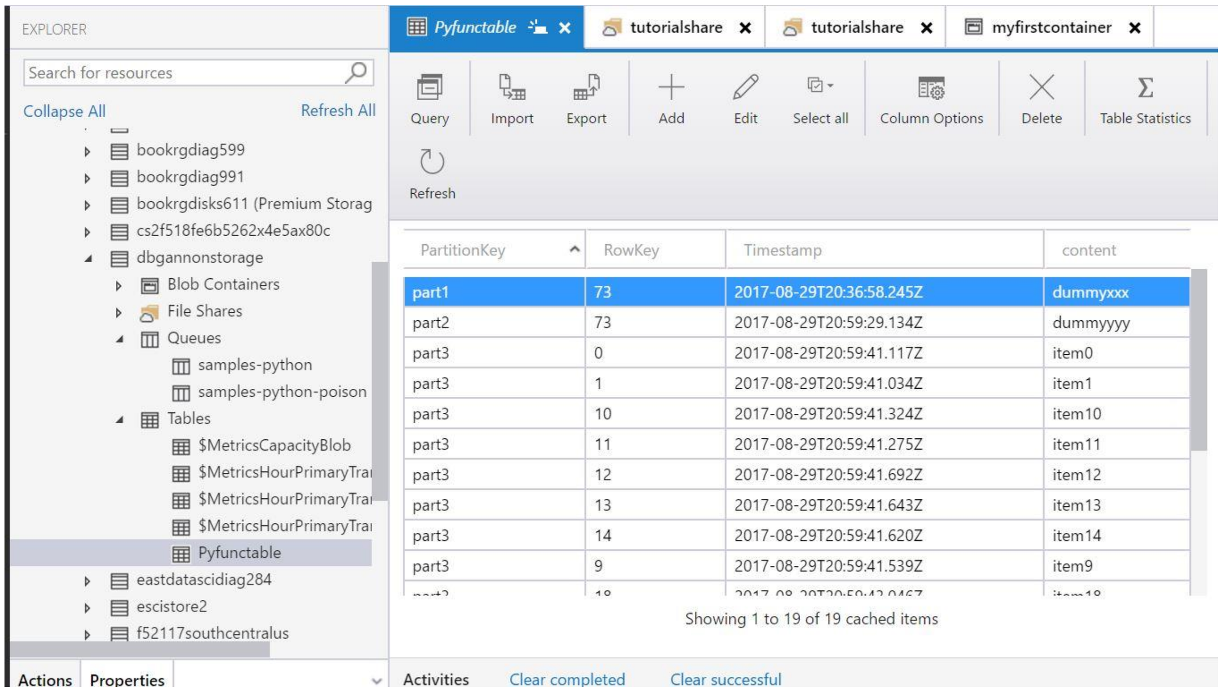
DefaultEndpointsProtocol=https;AccountName=dbgannonstorage;AccountKey=1mZ333pSzpmwM

Figure 8. The association between the storage account queue and table service with the variable in our function. Here we have highlighted the “show value” to verify that it has the right storage account.

You should see your storage account in the dropdown menu. Select it. And they add the Table name. You need to do the same for the AzureQueueStorage.

Once this is done and your function is saved and the system is running your function should be instantiated and invoked as soon as you send it queue items. For that we have a simple python script in a Jupyter notebook. You can get it from https://SciEngCloud.github.io/py-functions-queue-driver.ipynb . You will need to fill in your account key for the storage account, but then you should be able to step through the rest.

The notebook runs a few tests and then sends 20 items to the queue. Using the Azure storage explorer we see the results in the table as shown below.



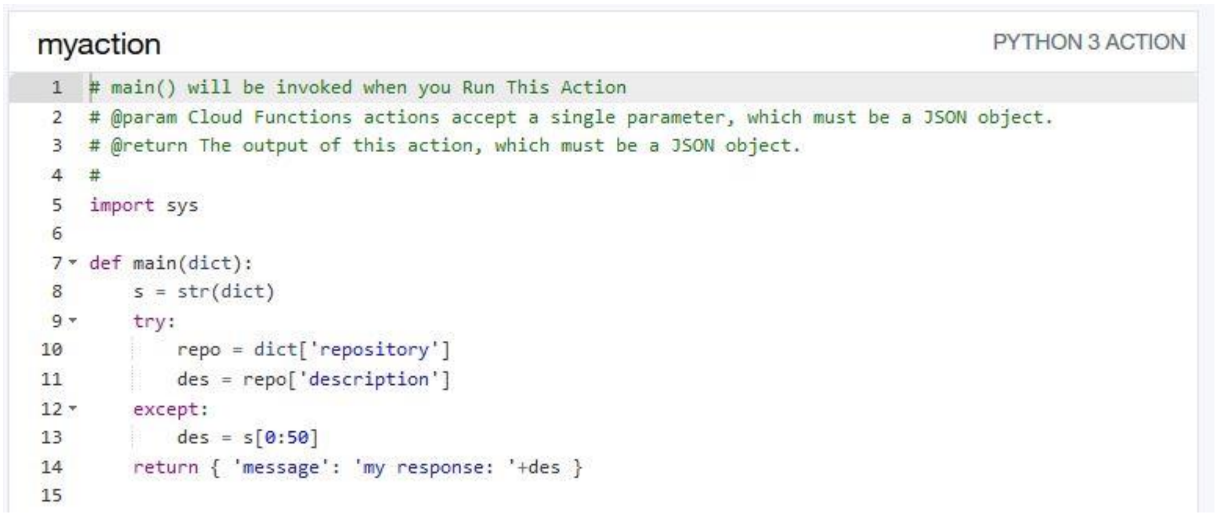
PartitionKey	RowKey	Timestamp	content
part1	73	2017-08-29T20:36:58.245Z	dummyxxx
part2	73	2017-08-29T20:59:29.134Z	dummyyyy
part3	0	2017-08-29T20:59:41.117Z	item0
part3	1	2017-08-29T20:59:41.034Z	item1
part3	10	2017-08-29T20:59:41.324Z	item10
part3	11	2017-08-29T20:59:41.275Z	item11
part3	12	2017-08-29T20:59:41.692Z	item12
part3	13	2017-08-29T20:59:41.643Z	item13
part3	14	2017-08-29T20:59:41.620Z	item14
part3	9	2017-08-29T20:59:41.539Z	item9

Showing 1 to 19 of 19 cached items

Figure 9. The view of the table after running the Jupyter Notebook.

OpenWhisk and IBM Bluemix

OpenWhisk is the open source serverless function system developed by IBM that is also supported as a commercial service on their Bluemix cloud. Like the others it is reasonably easy to use from their command-line tools, but the Bluemix portal provides an even easier solution. Creating a function is as easy as selecting the runtime and downloading a template. Figure 10 below illustrates a simple function derived from template for a Python3 function.



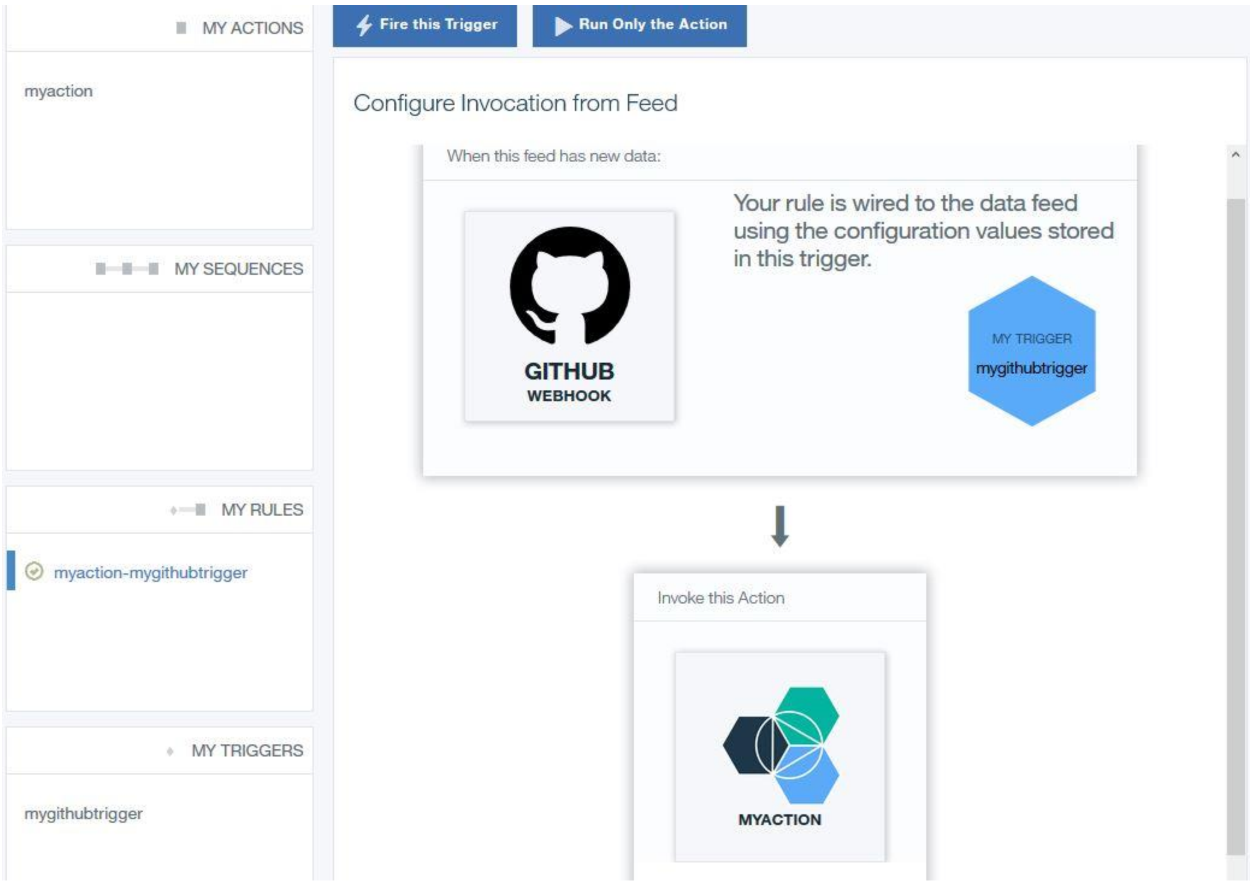
```python
# main() will be invoked when you Run This Action
# @param Cloud Functions actions accept a single parameter, which must be a JSON object.
# @return The output of this action, which must be a JSON object.
#
import sys

def main(dict):
    s = str(dict)
    try:
        repo = dict['repository']
        des = repo['description']
    except:
        des = s[0:50]
    return { 'message': 'my response: '+des }

```

Figure 10. An OpenWhisk function that decodes a dictionary input and returns a dictionary

Notice that the parameter to the function is a python dictionary (all Openwhisk messages are actually Json objects which, for Python, are rendered as dictionaries). While this template can be run as a web service in its present form, it is also possible to connect it to an external trigger. To illustrate this we connected it to a trigger that monitors activity in a Github account.

Using the portal it was easy to create the trigger (called "mygithubtrigger") and bind it to push and delete actions on a repository called dbgannon/whisk. All this required was an access token which was easy available by logging in to the GitHub portal. In the case of GitHub triggers the event returns a massive Json object that is a complete description of the GitHub repository. In the action "myaction" we go two levels down in the dictionary and extract the official repository description and make that the response of the action to the event.

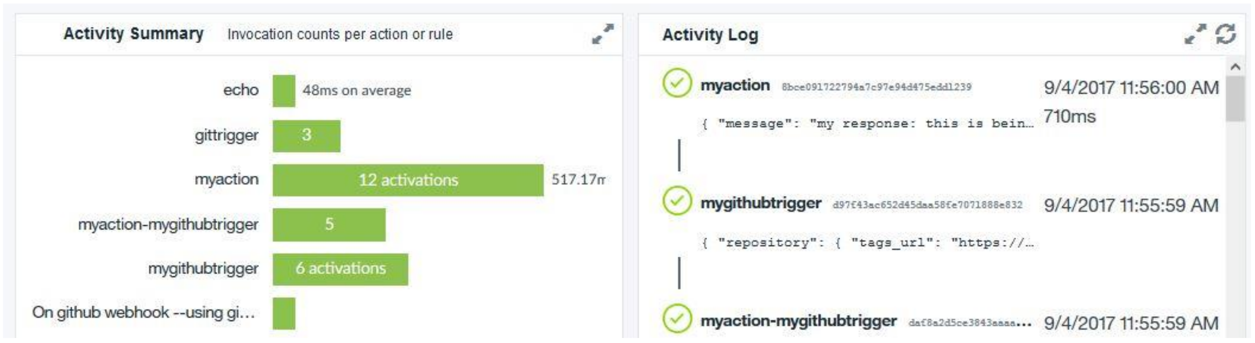
When a trigger fires you need a rule which bind the firing to an action. We bound it to the trivial "myaction" example above. The view from the portal of the rule is below



Your rule is wired to the data feed using the configuration values stored in this trigger.

Figure 11. The rule view of our association of the trigger to the function.

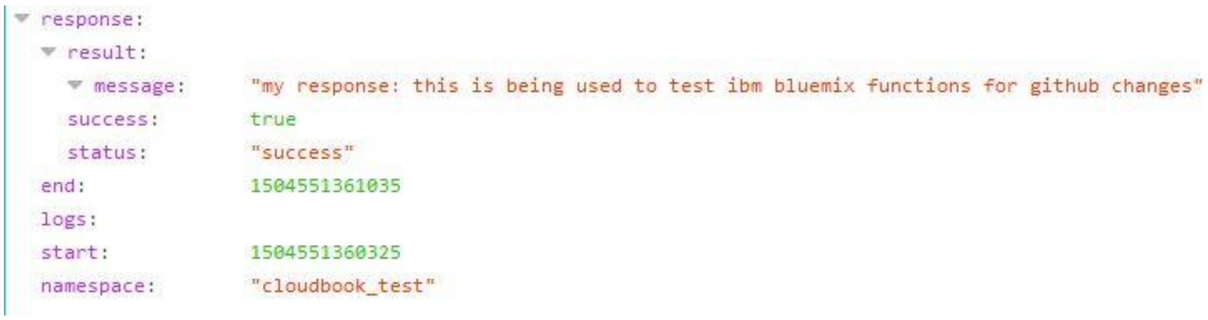
We next added a new file to the repository. This activated the trigger and the rule invoked the action. The portal has a nice monitor facility and the image in Figure 12 below.



```
myaction 8bce091722794a7c97e94d475edd1239                9/4/2017 11:56:00 AM
  { "message": "my response: this is bein…               710ms

mygithubtrigger d97f43ac652d45daa58fe7071888e832         9/4/2017 11:55:59 AM
  { "repository": { "tags_url": "https://…

myaction-mygithubtrigger daf8a2d5ce3843aaaa...           9/4/2017 11:55:59 AM
```

Figure 12. This is the "Monitor" view after the tirgger has fired.

Finally, drilling down on the "myaction" log event we see the description of the GitHub repository we created.



```
response:
  result:
    message:      "my response: this is being used to test ibm bluemix functions for github changes"
    success:      true
    status:       "success"
  end:            1504551361035
  logs:
  start:          1504551360325
  namespace:      "cloudbook_test"
```

Figure 13. The output of the "myaction" function after a file was added to the GitHub repository.

Finally

These examples above are all very trivial. The next thing to explore is how functions can be composed into workflows. Each of the three systems has its own way of doing that and if we have time later we will show some examples of this capability. We have also not disussed performance or cost which is greatly dependent on the rate at which your triggers are firing and the amount of work in each function execution.

The economics of serverless computing are also very interesting. As we pointed out earlier, for a cloud provider to offer a FaaS serverless capability it must be supported by actual infrastructure. How can the provider afford this if it is only charging by the second? There are two possible answers. First if your FaaS is serving very large numbers of function per second then it will certainly pay for itself. But there is another consideration. Many of the big cloud providers are running vast microservice frameworks that support most, if not all of their big internal and publiclly available applications. Running a FaaS system on top of that is as easy as running any other microservice based application. One only needs to set the cost per second of function evaluation high enough to cover the cost of that share of the underlying platform the FaaS system is using. For Amazon Lambda "The price depends on the amount of memory you allocate to your function. You are charged $0.00001667 for every GB-second used." This amounts to $0.06 per GB hour. So for a function that take 8 GB of memory to execute that is $.48 per hour, which is 4.8 times greater than the cost of an 8GB m4.large ec2 instance. Consequently a heavily used FaaS system is a financial win and a lightly used one may have little impact on your infrastructure.

Of course, single executions of a FaaS function are limited to five minutes, so you would need 12 5 minute concurrent executions to reach an hour of execution time. Furthermore AWS give you 1million invocations (at 100ms) each month for free or 400,000 GB-seconds per month free. That is 111 GB hours per month for free.   This is not a bad deal and it would seem to indicate that Lambda is not a big drain on their infrastructure yet.

Acknowledgement: Thanks go to Ryan Chard for helpful comments and suggestions on a draft of this article.