Comparative Analysis of Cloud Computing Platforms

Serverless Computing - AWS Lambda, Azure Functions & Google Cloud Functions

Image Source: https://www.janbasktraining.com/blog/google-vs-aws-vs-azure/

PREPARED BY

Somya Sharma	827006361
Deepak Raj Komaraboina	827001735

TABLE OF CONTENTS

INTRODUCTION

Serverless Computing technology is a major paradigm shift in terms of cloud ecosystems and its adoption is expanding day-by-day. Serverless Computing came into the picture 5 years ago, and the ecosystem is still maturing. FaaS is a product of Serverless computing which offers flexibility and scalability at a low cost which attracts customers.

AWS Lambda: AWS is one of the oldest players and it started its journey in 2014. It supports Node JS, Python, Java, and .NET languages (C#, Visual Basic and F#). The lambda function concurrency rate is 1000 executions maximum with maximum execution time as 15 minutes.

Azure Functions: Azure Functions joined the race in 2016. It supports Javascript, Node.js, C#, F#, Python, PHP. Azure offers unlimited concurrency rate with maximum execution time as 10 minutes.

Google Cloud Functions: Google Cloud Functions were offered in 2017. GCP also offers unlimited invocations for HTTP trigger and execution time upgraded to 9 minutes.

There are multiple new entrants in the market which are providing competitive offerings in addition to AWS, Azure, Google Cloud Platform and IBM. The selection of the best platform depends on various factors including use case requirements, price constraints, performance requirements, throughput, and latency constraints. Hence, a comparative analysis among the FaaS services would enable us to determine an appropriate option.

Some of the key comparison points discussed in the report would be:

- Response latencies of the FaaS offerings
- Transactions per second
- Connection times that were taken by each request on the three platforms
- Total response times

This report primarily focuses on the three main FaaS competitors i.e. Amazon Web Services, Microsoft Azure and Google Cloud Platform. AWS provides FaaS offerings as AWS Lambda, Azure provides Azure Functions and GCP has Google Cloud Functions. The aim is to provide a comparative analysis of the three available solutions based on the parameters like throughput, latency, and cold start statistics.

The following sections namely motivation, methodology, results and discussion provide the above information in detail. These results would impact the decisions in selecting a cloud platform for the FaaS offering. We tried to capture as many statistics and measurements that would enable cloud users to make an insightful choice.

MOTIVATION

The main motivation for Faas can be described by comparing FaaS with the other services provided by the cloud providers namely Infrastructure as a Service (IaaS) and Platform as a Service (PaaS).

FaaS vs PaaS vs IaaS

Firstly, IaaS provides the necessary infrastructure to the developers namely server, storage, network infrastructure, hypervisor, and security support. But the caveat is that the Developer has to take care of the OS, loading the OS into the VM, application and the functions. But, in the case of Platform as a Service, the service providers take care of the OS, the framework and other development/runtime services. This leaves the users with the responsibilities of application and the data. This is a better model than IaaS in terms of the infrastructure overhead of the developers, but the developers still need to do the deployment and the configuration. Function as a service(FaaS) goes one step further and relieves the developer of not only the infrastructure overhead but also the task of configuration. The developers can just leverage the services provided by the FaaS providers and just focus on the functions which they want to implement.

The following diagram gives a pictorial representation of the differences:

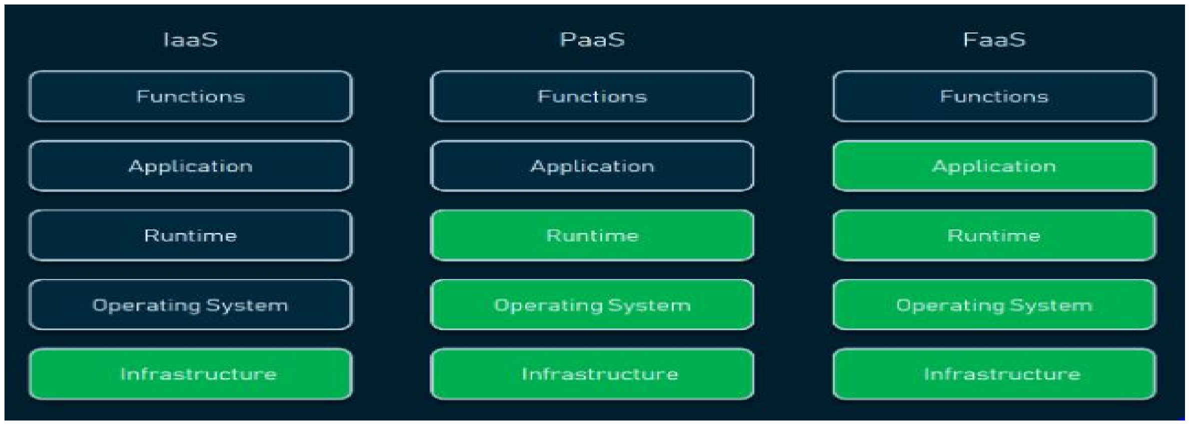


Image Source: https://www.altexsoft.com/blog/cloud/comparing-serverless-architecture-providers-aws-azure-google-ibm-and-other-faas-vendors/

Figure 1: FaaS vs PaaS vs IaaS

If we consider a situation where a Technology Startup is trying to come up with a product, the main end-goal of the company would be to design the best product with very less time to market. In this case, the company cannot afford to spend time either on setting up the server/networking infrastructure nor on creating VM's and deployment configuration. The wise thing to do would be to utilize Faas for some of the tasks and maximize their profits.

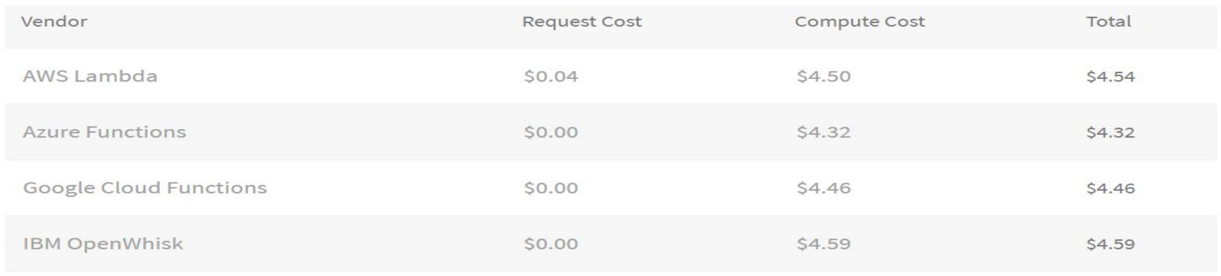
The following are some key driving factors for the major paradigm shift towards FaaS.

1. **Pay-per-usage model:** The users need not pay in advance for the services. The resources are allocated the FaaS providers according to the user's requirements and they are scaled accordingly. Finally, the users pay as they use on the fly. This is a huge advantage for microservices which are short-lived and they end up paying less as they are not paying for unused VM or container server time.

2. **Scalability:** The resources allocated based on the event and the load on the server and is managed efficiently by the FaaS providers. This relieves the developers of the constant monitoring and optimizing the server usage by themselves.

3. **Security:** The service providers provide security benefits and the developers need not worry about the server security issues and the fear of OS attacks, malicious software attacks and denial of service attacks. This even removes the burden of maintaining the security infrastructure on the part of the developers.

Cloud users need to find the basis of comparison between the FaaS offerings to select the best platform. The available offerings provide very similar and competitive pricing. Refer below price structure for serverless calculated using: "Serverless Cost Calculator" http://serverlesscalc.com/ for 10,000 executions, execution time 18000 milliseconds and memory size as 1536MB. It clearly shows that each platform offers more or less very similar pricing. Hence, pricing cannot be the only attribute to decide the platform.



Vendor	Request Cost	Compute Cost	Total
AWS Lambda	$0.04	$4.50	$4.54
Azure Functions	$0.00	$4.32	$4.32
Google Cloud Functions	$0.00	$4.46	$4.46
IBM OpenWhisk	$0.00	$4.59	$4.59

Our experiments try to focus on some of the aforementioned advantages of Function as a Service which would help the user take the right decision. We have used similar configurations across all platforms such as server location, the complexity of the code and the number of transactions. This would give us an opportunity for accurate comparison using various parameters. Our main goal is to compare the transactions per second, latencies and connection times respectively. These can be used to compare the key qualities of FaaS such as pay-per-usage and scalability across different platforms. Since pay-per-usage and scalability are the deciding factors for shifting to FaaS, the cloud provider with the best results would the clear winner.

METHODOLOGY

We implemented a function on all the three platforms to test the performance, efficiency, and throughput for the FaaS offerings of AWS, Microsoft Azure and Google Cloud Platform. The function was written in Python on all the three platforms and was invoked using the API gateway or HTTP Trigger. The function converts the input JSON request body to XML response. Postman was used for testing the exposed APIs and JMeter was used to perform the load and performance testing.

All the deployments were made in Europe region:

- AWS - Frankfurt (eu-central-1)
- GCP - Belgium (europe-west1)
- Azure - Netherlands (westeurope)

For AWS and GCP, the converter function was written in Python 3.7. As Azure does not support Python 3.7 yet, Python 3.6 was used.

Note: *All the tests were performed using the free tier offered by the 3 platforms. Hence certain configuration and settings are based on availability.*

As mentioned earlier we believe our methodology of comparing the transactions per second, latencies and connection times would help cloud users make a wise decision. Hence, we have designed an experiment wherein we deployed a JSON to XML converter in Python on all three platforms. Using the API exposed, these functions can easily be tested based on the above mentioned parameters.

1. **Azure Function**

   For Implementing the Azure functions in python we had to adopt the following approach:

   a. Installed the Azure Functions Core Tools to build and test locally
      i. Created and activated a virtual environment
      ii. Created a local functions project with worker runtime as Python
      iii. Developed an HTTP trigger based Function which converts JSON to XML and then tested the code locally
   b. Deployed the code using Azure CLI
      i. Created a resource group and an Azure Storage account

         - *az group create --name deepakResourceGroup8 --location westeurope*

- *az storage account create --name memx --location westeurope --resource-group deepakResourceGroup8 --sku Standard_LRS*

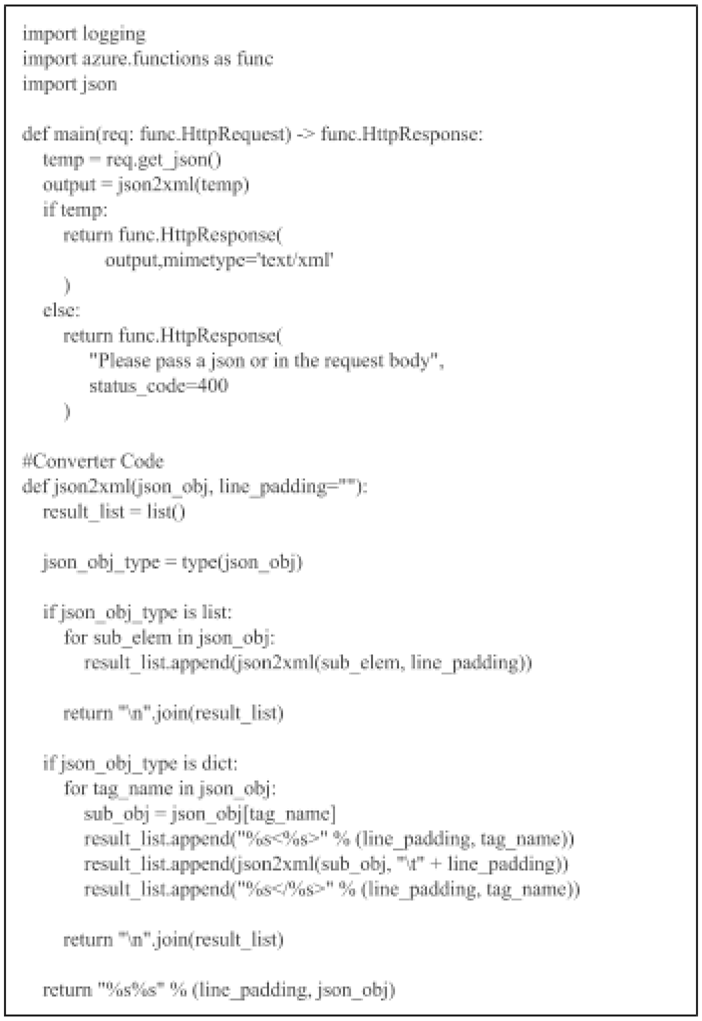
ii. This was followed by creating a Linux function App

- az functionapp create --resource-group deepakResourceGroup8 --os-type Linux \ --consumption-plan-location westeurope --runtime python \ --name jxconverter --storage-account memx

iii. Finally, deployed the function app project to Azure

- func azure functionapp publish jxconverter

Code Snippet:



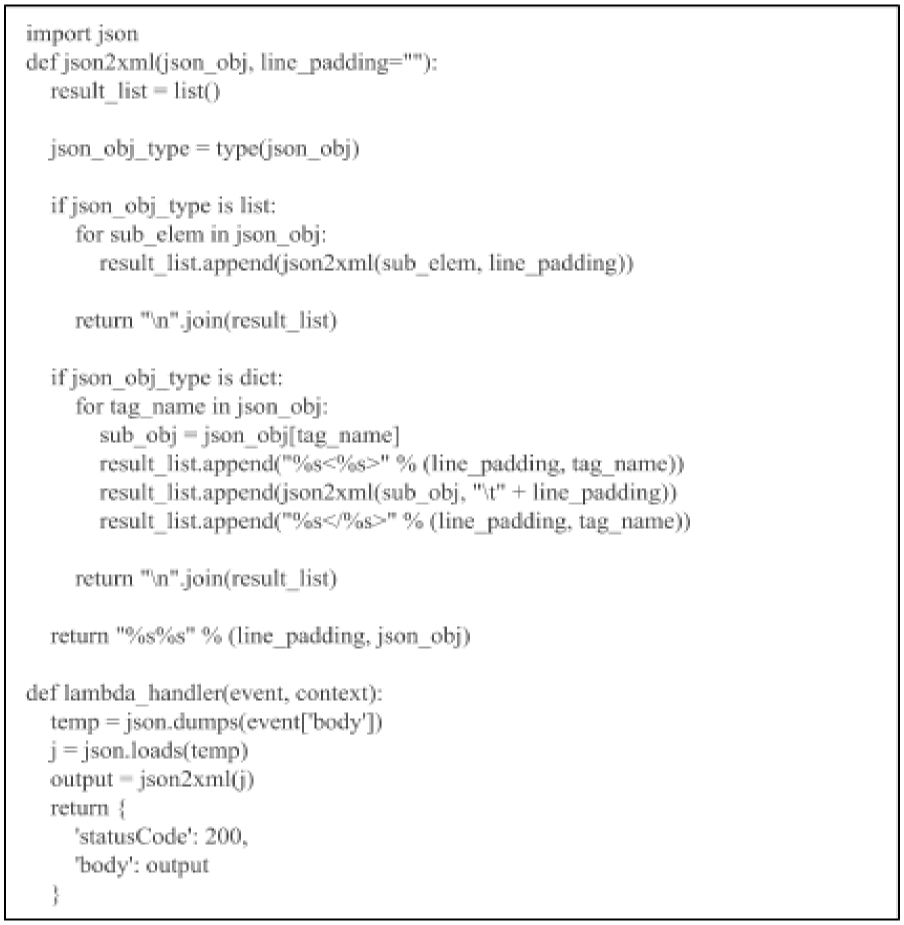
```
import logging
import azure.functions as func
import json

def main(req: func.HttpRequest) -> func.HttpResponse:
    temp = req.get_json()
    output = json2xml(temp)
    if temp:
        return func.HttpResponse(
                output,mimetype='text/xml'
        )
    else:
        return func.HttpResponse(
             "Please pass a json or in the request body",
             status_code=400
        )

#Converter Code
def json2xml(json_obj, line_padding=""):
    result_list = list()

    json_obj_type = type(json_obj)

    if json_obj_type is list:
        for sub_elem in json_obj:
            result_list.append(json2xml(sub_elem, line_padding))

        return "\n".join(result_list)

    if json_obj_type is dict:
        for tag_name in json_obj:
            sub_obj = json_obj[tag_name]
            result_list.append("%s<%s>" % (line_padding, tag_name))
            result_list.append(json2xml(sub_obj, "\t" + line_padding))
            result_list.append("%s</%s>" % (line_padding, tag_name))

        return "\n".join(result_list)

    return "%s%s" % (line_padding, json_obj)
```

2. AWS Lambda

For writing Lambda functions following steps were followed:

a. After logging in to AWS account, navigate to Lambda console
b. Select creating a function from scratch option, i.e. Author from scratch
c. Ensure the correct availability region is selected from the top-right corner
d. Enter information such as function name, Runtime (Python 3.7), and select the existing role
e. Then the below code snippet is written.
f. API gateway is added as a trigger and a resource with POST action is created
g. Configure Method Request, Integration Request, Method Response, and Integration Response
h. Once ready the API can be tested and deployed which generates an API endpoint

Code Snippet:



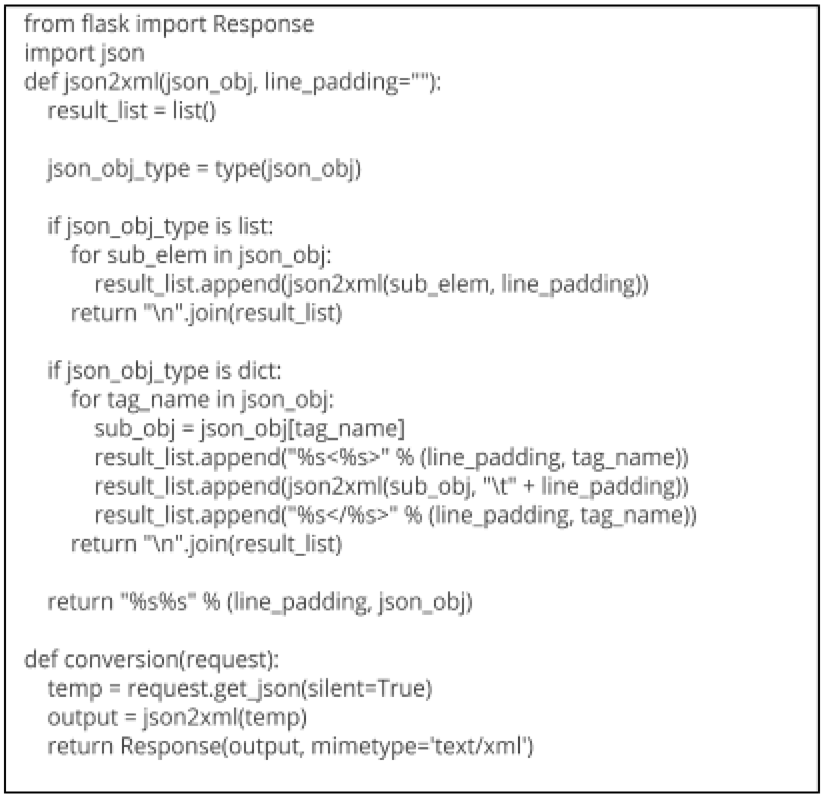
```
import json
def json2xml(json_obj, line_padding=""):
    result_list = list()

    json_obj_type = type(json_obj)

    if json_obj_type is list:
        for sub_elem in json_obj:
            result_list.append(json2xml(sub_elem, line_padding))

        return "\n".join(result_list)

    if json_obj_type is dict:
        for tag_name in json_obj:
            sub_obj = json_obj[tag_name]
            result_list.append("%s<%s>" % (line_padding, tag_name))
            result_list.append(json2xml(sub_obj, "\t" + line_padding))
            result_list.append("%s</%s>" % (line_padding, tag_name))

        return "\n".join(result_list)

    return "%s%s" % (line_padding, json_obj)

def lambda_handler(event, context):
    temp = json.dumps(event['body'])
    j = json.loads(temp)
    output = json2xml(j)
    return {
        'statusCode': 200,
        'body': output
    }
```

3. Google Cloud Function

For writing Google Cloud functions following steps were followed:

a. After logging in to GCP account, navigate to Cloud functions
b. Select “Create Function” option and enter the information such as: function name, trigger as HTTP, Runtime as Python 3.7 and region
c. Once the function is written, it generates an HTTP URL which acts as a trigger for invoking the function

Code Snippet:

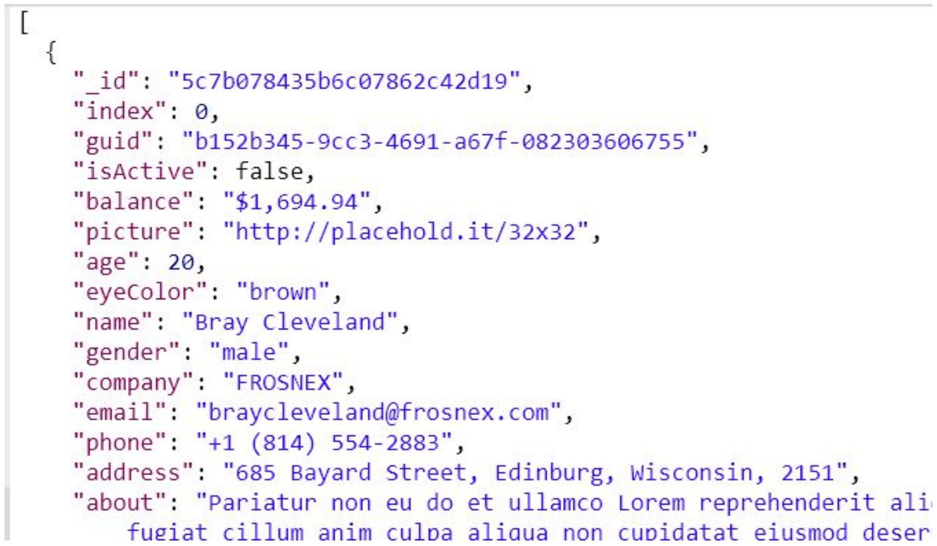


```
from flask import Response
import json
def json2xml(json_obj, line_padding=""):
    result_list = list()

    json_obj_type = type(json_obj)

    if json_obj_type is list:
        for sub_elem in json_obj:
            result_list.append(json2xml(sub_elem, line_padding))
        return "\n".join(result_list)

    if json_obj_type is dict:
        for tag_name in json_obj:
            sub_obj = json_obj[tag_name]
            result_list.append("%s<%s>" % (line_padding, tag_name))
            result_list.append(json2xml(sub_obj, "\t" + line_padding))
            result_list.append("%s</%s>" % (line_padding, tag_name))
        return "\n".join(result_list)

    return "%s%s" % (line_padding, json_obj)

def conversion(request):
    temp = request.get_json(silent=True)
    output = json2xml(temp)
    return Response(output, mimetype='text/xml')
```

RESULTS

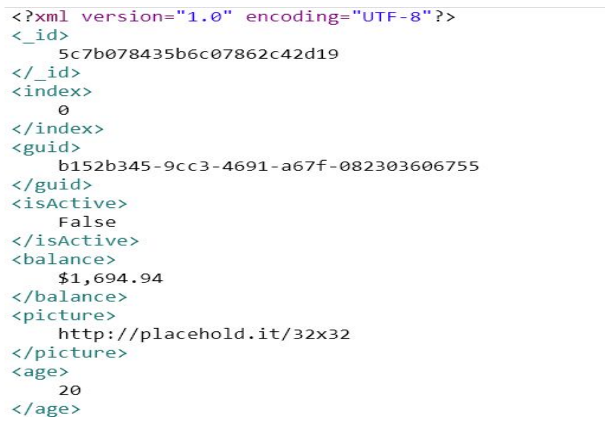
To test the Serverless deployments we used JMeter to trigger the APIs exposed on AWS, Azure, and GCP. To observe the comparison of the response time, latency and throughput we plotted using comparative graphs based on the results. Three different tests were performed changing the number of request configurations and the results were captured. The following metrics are captured: throughput per second, connection time, response latency and total response time.

The testing was done in 3 different stages. The number of users and loops was changed in the experiments to deduce the comparative results. JSON request with approximately 25K lines was used as input to test the performance.



```
[
  {
    "_id": "5c7b078435b6c07862c42d19",
    "index": 0,
    "guid": "b152b345-9cc3-4691-a67f-082303606755",
    "isActive": false,
    "balance": "$1,694.94",
    "picture": "http://placehold.it/32x32",
    "age": 20,
    "eyeColor": "brown",
    "name": "Bray Cleveland",
    "gender": "male",
    "company": "FROSNEX",
    "email": "braycleveland@frosnex.com",
    "phone": "+1 (814) 554-2883",
    "address": "685 Bayard Street, Edinburg, Wisconsin, 2151",
    "about": "Pariatur non eu do et ullamco Lorem reprehenderit ali
        fugiat cillum anim culpa aliqua non cupidatat eiusmod deser
```

Figure 2: JSON Request



```
<?xml version="1.0" encoding="UTF-8"?>
<_id>
    5c7b078435b6c07862c42d19
</_id>
<index>
    0
</index>
<guid>
    b152b345-9cc3-4691-a67f-082303606755
</guid>
<isActive>
    False
</isActive>
<balance>
    $1,694.94
</balance>
<picture>
    http://placehold.it/32x32
</picture>
<age>
    20
</age>
```

Figure 3: XML Response

<u>Test 1:</u> For the Test 1, we considered 5 parallel users triggering the API. This is repeated twice to understand the behaviour of consecutive parallel requests.

- Concurrent Requests = 5
- Repetition of Loop = 2
- Total number of Requests = 5*2 = 10

Graphs captured for the following parameters:

a. Transactions per second

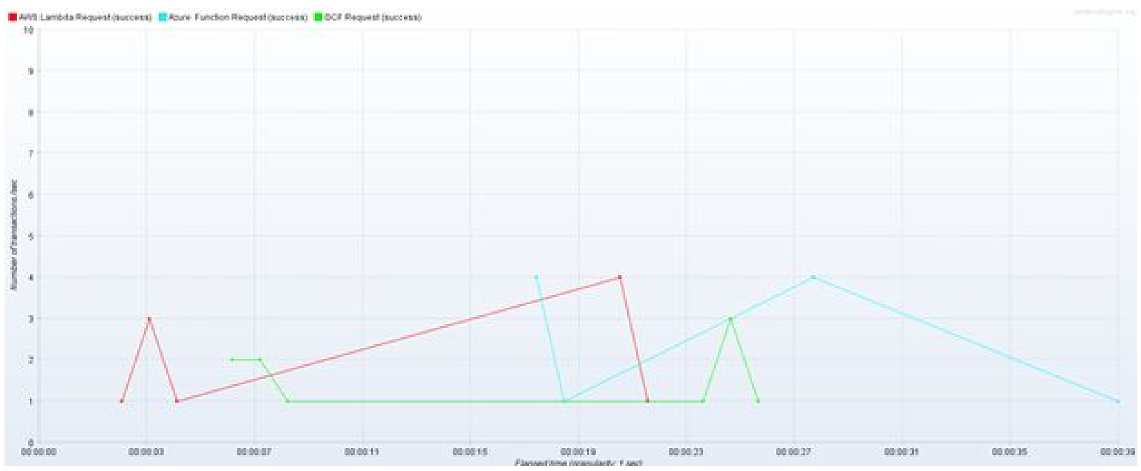


Figure 4: Transactions per Second - Test 1

b. Response latencies over time

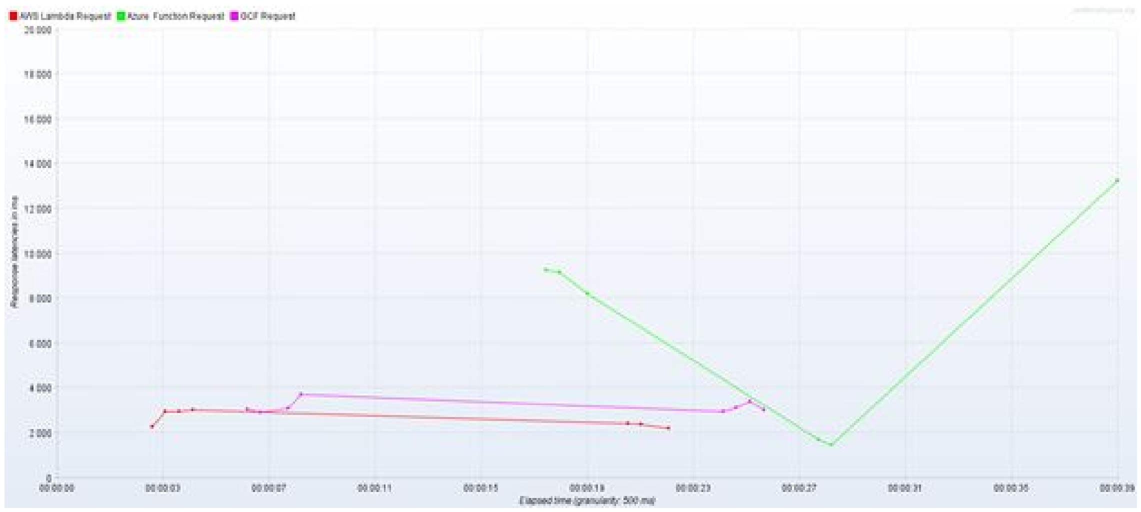


Figure 5: Response Latencies - Test 1

c. Connection time

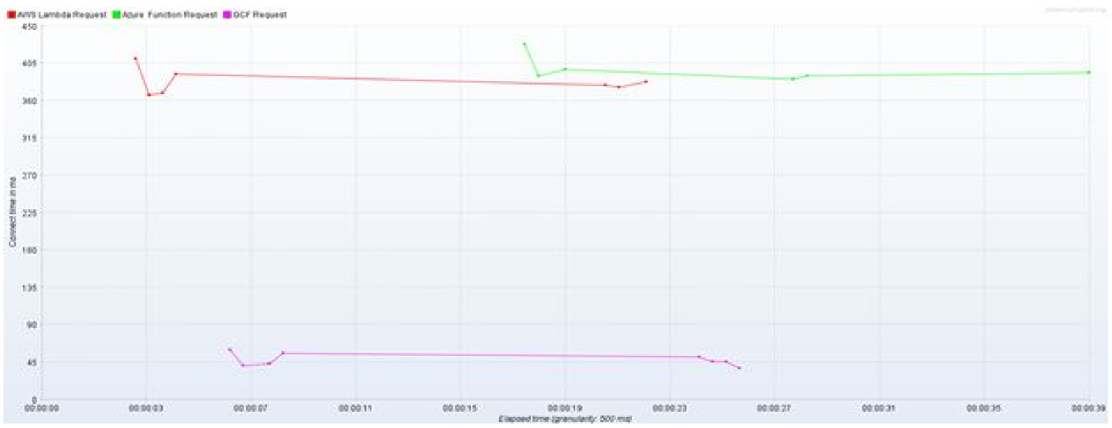


Figure 6: Connection Time: Test 1

d. Response time

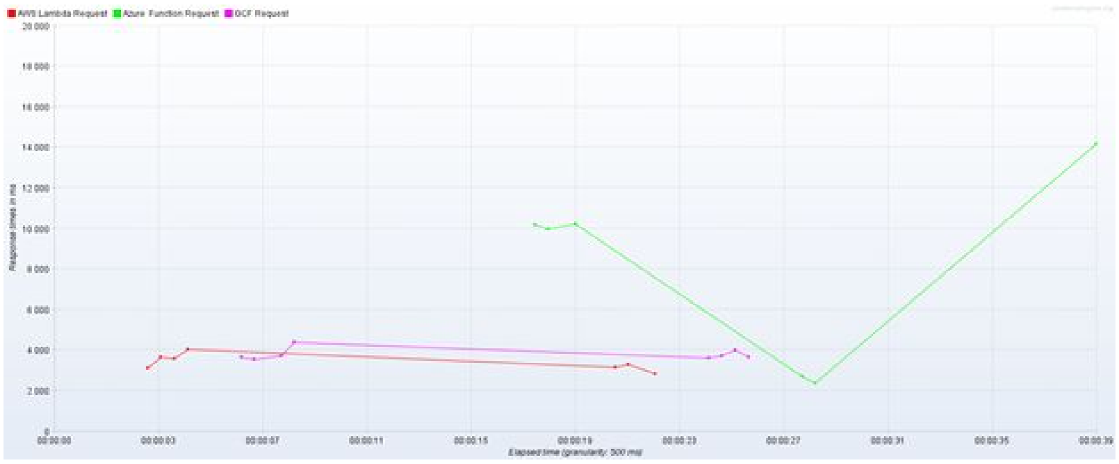


Figure 7: Response Time - Test 1

Table 1: Summary Report - Test 1

Label	#Samples	Average	Min	Max	Std. Dev.	95% Line	99% Line	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
AWS Lambda Request	10	3400	2862	4041	317.52	3654	4041	0.4555	327.05	347.56	735239
GCF Request	10	3815	3562	4388	243.8	4026	4388	0.44462	319.22	339.26	735191.8
Azure Function Request	10	7547	2371	14179	4207.31	10934	14179	0.31444	225.79	239.93	735298.4

From the above table, it is evident that the throughput is considerably greater for AWS and GCP (in the order of 0.4) when compared to Azure (0.31).

Test 2: For this test, the number of parallel users 5 is kept same. However, the repetition of triggers is increased to 20. This is done to understand the warm start performance of the FaaS.

- Concurrent Requests = 5
- Repetition of Loop = 20
- Total number of Requests = 5*20 = 100

Graphs captured for the following parameters:

a. Transactions per second

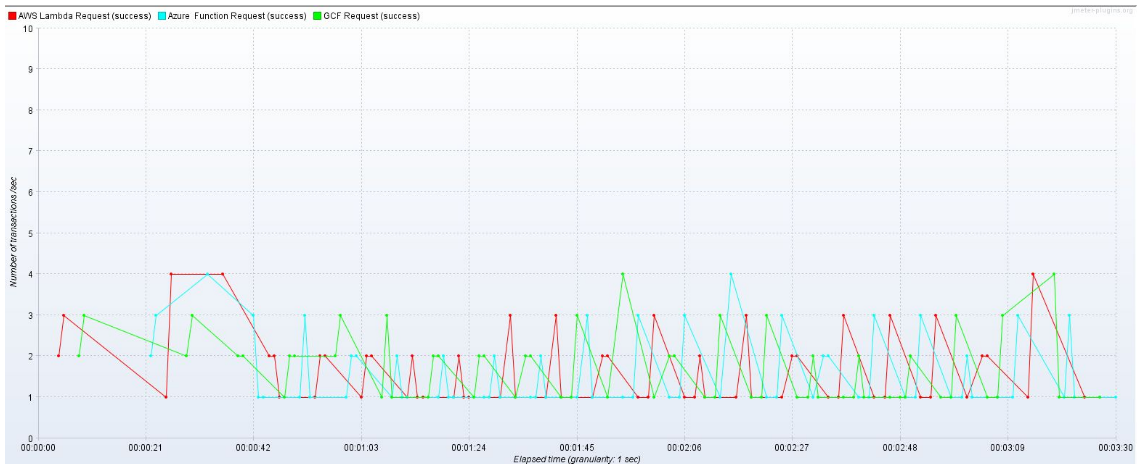


Figure 8: Transactions per Second - Test 2

b. Response latencies over time

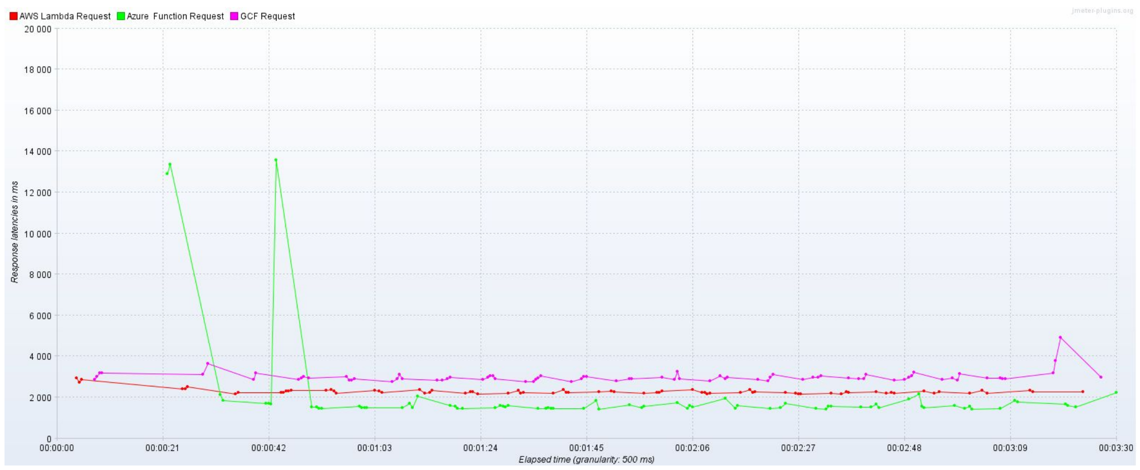


Figure 9: Response Latency - Test 2

c. Connection time

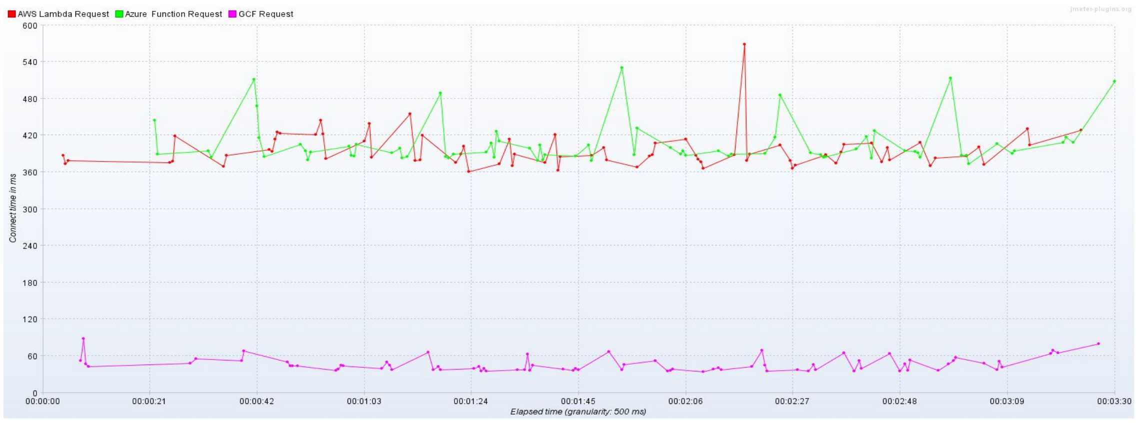


Figure 10: Connection Time - Test 2

d. Response time

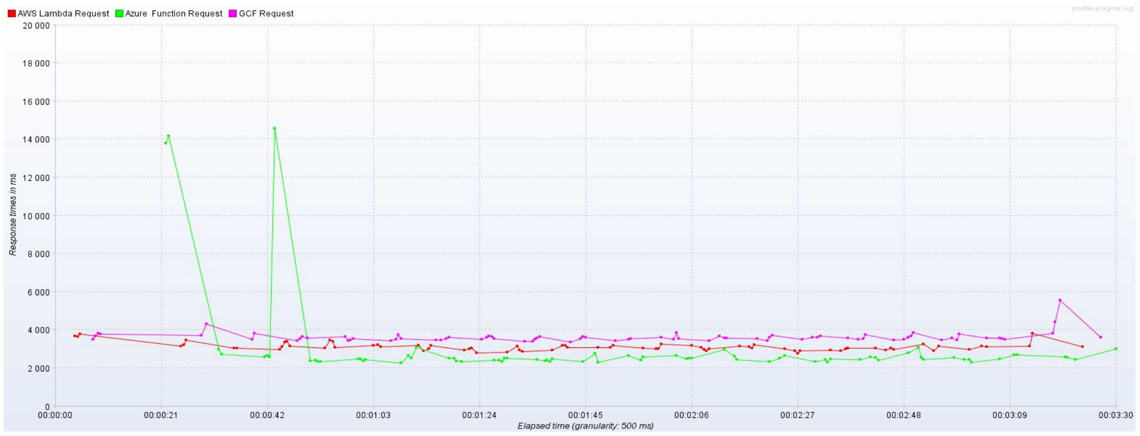


Figure 11: Response time - Test 1

Table 2: Summary Report - Test 2

Label	#Samples	Average	Min	Max	Std. Dev.	95% Line	99% Line	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
AWS Lambda Request	100	3157	2783	6061	350.88	3564	3810	0.49217	353.38	375.54	735239
GCF Request	100	3652	3372	5555	281.28	4210	4438	0.49248	353.58	375.77	735191.7
Azure Function Request	100	3250	2288	14676	2756.93	13246	14587	0.49378	354.57	376.78	735318.4

In this case, 5 concurrent requests were taken and the throughput is almost of same order with Azure having slightly greater throughput.

<u>Test 3:</u> For test 3, the repetition of request was increased to 100 with a single user at a time.This test is performed to understand how each platform responds when the transactions are evenly distributed.

- Concurrent Requests = 1
- Repetition of Loop = 100
- Total number of Requests = 1*100 = 100

Graphs captured for the following parameters:

a. Transactions per second

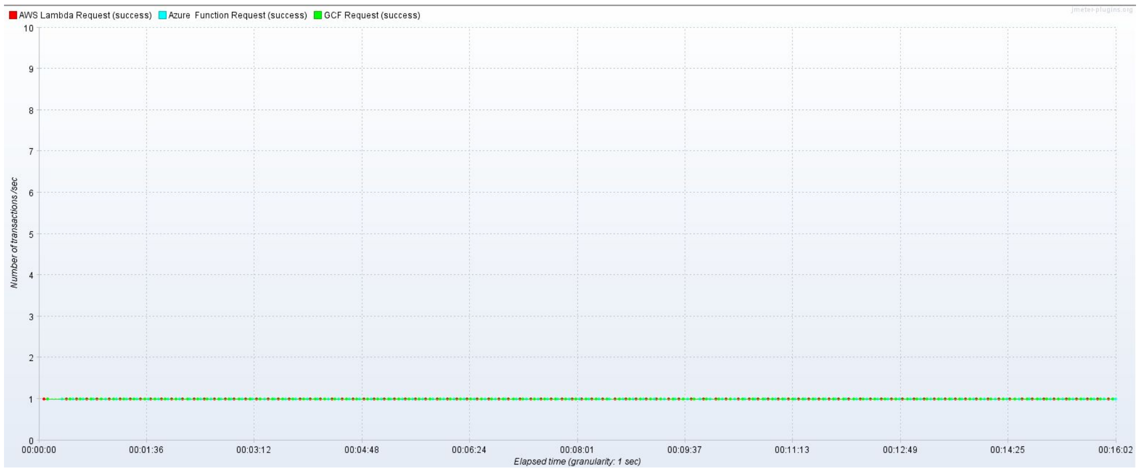


Figure 12: Transaction per Second - Test 3

b. Response latencies over time

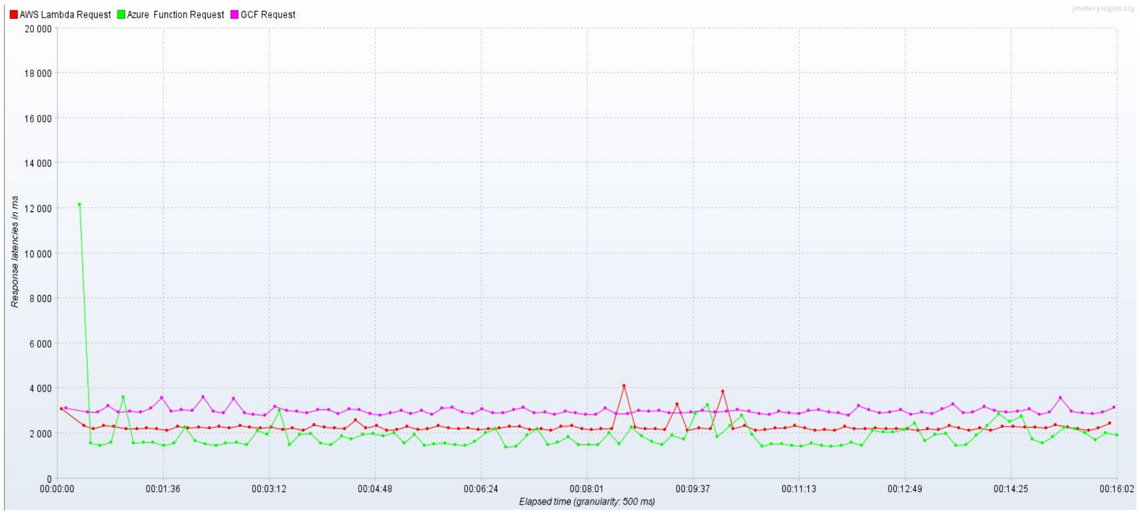


Figure 13: Response Latency - Test 3

c. Connection time

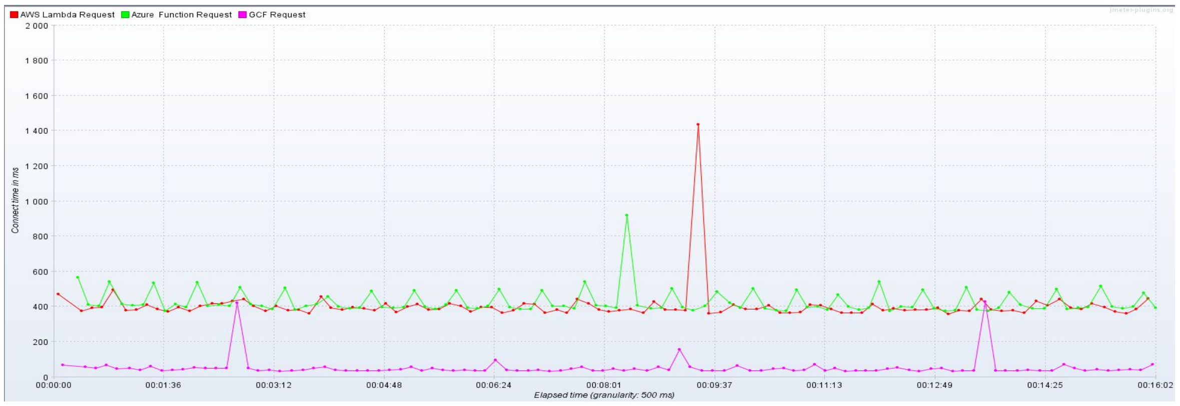


Figure 14: Connection time - Test 3

d. Response time

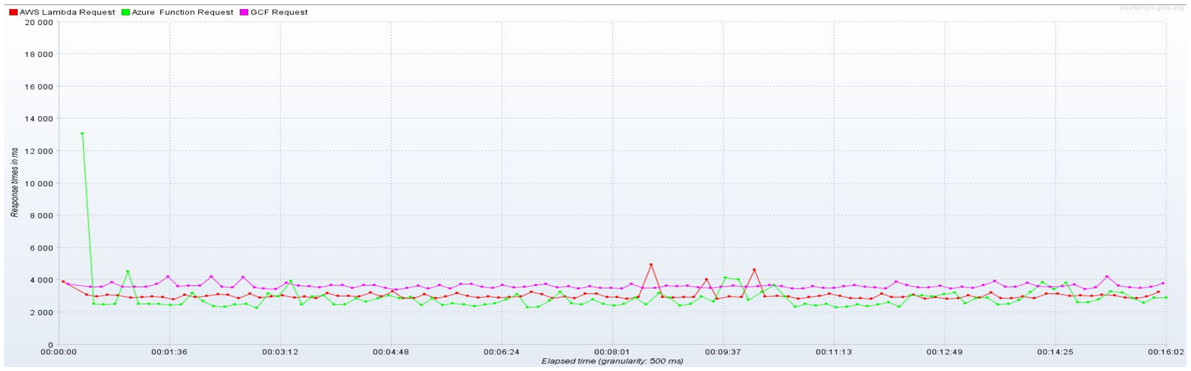


Figure 15: Response time - Test 3

Table 3: Summary Report - Test 3

Label	#Samples	Average	Min	Max	Std. Dev.	95% Line	99% Line	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
AWS Lambda Request	100	3053	2825	4973	306.14	3265	4637	0.10471	75.18	79.89	735239
GCF Request	100	3634	3416	4229	156.31	3887	4221	0.10472	75.18	79.9	735193.2
Azure Function Request	100	2905	2290	13095	1113.91	3886	4552	0.10481	75.26	79.97	735314.1

In this case, a single request at a time was considered but the loop iterations were increased to 100. Even here we see similar throughput and Azure having a slightly greater throughput.

DISCUSSION

Finally, based on the above test experiments, we can deduce the following:

- AWS Lambda has the most consistent performance i.e. a very low standard deviation. When there are multiple parallel users or multiple consecutive transactions the performance is quite stable.
- Azure Functions have a high cold start time for initial instances however it load-balances the requests eventually. When the number of consecutive requests is increased we can observe that Azure functions show stable and comparatively better response times. Due to high response latency initially, Azure functions have a very high standard deviation.
- Google Cloud Functions, compared to the former two has better connection time.

The selection of the platform would entirely depend on the use case and performance requirements. If better usability, low cold start time and consistent performance is desired, then ***AWS Lambda*** would be the most appropriate choice. However, if there are high number of users and requests per second due to which the cold start time will not impact the performance, then ***Azure Functions*** would be a great choice as it provides high throughput.

However, the other platforms are also maturing and progressing. The technological advancements might improve the transactions per second and reduce the latencies thereby increasing the throughput. Currently, the connection time for ***Google Cloud Functions*** is better due to its massive server infrastructure. This might change in the future as the other competitors including AWS and Azure are constantly trying to improve their backend infrastructure.

TECHNOLOGY STACK

REFERENCES

1) Serverless Computing Comparison Guide: Aws Lambda Vs. Microsoft Azure Functions Vs. Google Cloud Functions, http://techgenix.com/serverless-computing-comparison-guide/
2) Comparing Serverless Architecture Providers: AWS, Azure, Google, IBM, and other FaaS vendors (Dec 2018), https://www.altexsoft.com/blog/cloud/comparing-serverless-architecture-providers-aws-azure-google-ibm-and-other-faas-vendors/
3) Eirini-Eleni Papadopoulou (Aug 2018), "Pick a serverless fight: A comparative research of AWS Lambda, Azure Functions & Google Cloud Functions", https://jaxenter.com/comparison-aws-azure-google-147706.html
4) Liang Wang et al, "Peeking Behind the Curtains of Serverless Platforms", http://pages.cs.wisc.edu/~swift/papers/atc18-serverless.pdf
5) Rohit Akiwatkar (Sep 2018), "AWS Lambda vs Azure Functions vs Google Cloud Functions: Comparing Serverless Providers", https://www.simform.com/aws-lambda-vs-azure-functions-vs-google-functions/