PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications

Nikhila Somu, Nilanjan Daw, Umesh Bellur and Purushottam Kulkarni
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
{nikhilasomu, nilanjandaw, umesh, puru}@cse.iitb.ac.in

***Abstract*—Function-as-a-Service (FaaS) is a new offering in the cloud services domain. Multiple Cloud Service providers have come up with their own implementations of FaaS infrastructure providing end-users with a multitude of choices. Each such platform provides a non-overlapping set of features which satisfies a subset of users. A tool that automates capturing how a function behaves under different configurations of a platform and across platforms will, therefore, be useful for end-users intending to deploy applications as a collection of FaaS units. In spite of the presence of a few benchmarking tools for FaaS offerings, they lack the comprehensive breadth required to understand the performance aspects of the design choices made by the end-users. Most tools focus on tuning resource parameters like memory, CPU requirements and measure metrics like execution time. They lack the option to measure the effects of abstract features like function chaining and choice of function triggers. We present PanOpticon - a tool that automates the deployment of FaaS applications on different platforms under a set of tunable configuration choices and presents the users with performance measurements for each configuration and platform.**

***Index Terms*—serverless computing, benchmarking, automation, cross-platform**

I. Introduction

Serverless Computing or Function-as-a-Service (FaaS) is a cloud service model that abstracts out the notion of a server and lets the user deploy his/her application by only providing the business logic. The service provider handles the responsibility of resource provisioning, scaling and consolidation. The application code consists of smaller units called functions. Each of these functions is configured to be triggered on a specific event type. This model of application deployment charges users only for the amount of resources consumed by their application in servicing the requests/events. Serverless platforms auto-scales resources based on the number of incoming requests for the application ensuring maximum resource consolidation.

Serverless based architecture stacks have been embraced by companies like Netflix, Codepen, Coca-Cola etc., who have deployed parts of their application on a serverless platform [1] [2]. There are also various smaller companies like CloudSpoilt, Locise, etc., that totally or partially leverage serverless offerings [3]. Many serverless development companies are also sprouting up [4]. All this goes to show that Function-as-a-Service is gaining popularity in the Software Industry as an inexpensive way to deploy applications.

On the other side of the picture, we have Serverless providers like Amazon Web Services, Google Cloud Platform, Microsoft Azure Cloud, IBM Cloud, etc. Each of them has different implementation of the serverless platform, price points and support services.

Businesses looking to deploy their application using the serverless paradigm are hence presented with many options: option of provider, and within each provider, the options in various supporting services, for example, the choice of database service, be it traditional SQL based databases or schema-less databases, or the choice of function trigger models, HTTP/RESTful API based or a publisher-subscriber based model, etc. Each of these design choices has an impact on the performance of the application and in turn an impact on the end-users. A secondary aspect of designing FaaS based architectures is the slicing of business logic into sub-parts so that the functions can run efficiently and at the same time keeping the number of such functions manageable. Thus, businesses looking to design and deploy application logic as Serverless architecture needs to quickly design and test multiple designs spanning multiple tool stacks and multiple providers to get the best performance-value trade-off. Here arises the need for a tool that abstracts out platform-specific details as much as possible and allows a user to deploy his application on different platforms and also deploy different realizations of his application on the same platform, so that their relative performance can be compared and the optimum architecture be deployed, with minimal effort and time. To compare these deployments, the applications need to be stress-tested across multiple parameters, and metrics like function execution time, function memory usage, time spent in the supporting services, etc., need to be extracted and presented for comparative studies.

In this paper, we present our tool, PanOpticon, that exactly does the above.

II. Background and Related Work

A considerable amount of work has been put into developing an automated platform for benchmarking cross-platform performance. Wang et al [5] describes one such framework. Their open-sourced platform is mainly targeted towards benchmarking AWS Lambda and measuring its performance. The authors use multiple parameters like CPU, IO, and network latency, while measuring metrics like function execution time, response

time, throughput, etc. However, the authors neither provide any test cases modelled after any complex FaaS scenarios nor look into the effects of function chaining, two things that can have a considerable effect on real-life applications. A slightly different perspective to look into the performance of a serverless framework is to look into it from the microservices perspective [6]. The authors used indirect methods to measure and understand the way different platforms deploy serverless functions. They used the /proc/stat file provided by Linux to identify VMs and related containers, and metrics like incoming requests to VM count ratio, execution time, etc. This information was used to look into provider cold, VM cold and container cold start latency. However, these studies were again targeted mostly at AWS lambda. Also, the studies were primarily targeted at establishing benchmarks for serverless platforms and the authors didn't mention any efforts to opensource a general-purpose toolbox for end-users. Lee et al [7] used three types of workloads to measure platform performance, CPU intensive, IO intensive and network intensive. The authors also considered different function triggers like HTTP based, Object Store-based and Database based triggers to measure each platform's maximum dequeuing size. DeathStarBench [8] is an opensource benchmarking tool that models different real-life examples as test cases in the form of a social network, a media streaming server, an e-commerce service, a banking system and a drone swarm coordination system. The authors tested the possibility of running such systems using a cloud-based stack using metrics like QoS violation and compared results between a traditional VM based deployment vs a serverless function based deployment. Even though DeathStarBench models real-life use cases it fails to provide a general-purpose interface to allow end-users to test their application suites. [9] describes a HyperFlow [10] based benchmarking suite for the common serverless cloud infrastructures. They used Merssene Twister as their load generation module along with Linpack for their tests. Apart from these, [11] [12] details similar approaches to benchmarking public clouds. Figure 1 gives a comparative study of the existing benchmarking tools and their respective capabilities measured across multiple parameters, where a tick indicates the capability being present while a blank cell suggests an absence of the feature from that tool.

III. PanOpticon Description

We developed a tool that allows user to deploy his custom application on a chosen FaaS provider and also provides a way to test the application and compare a few different realizations of the same application. Here different means that either the architecture of the applications is different, or the application is deployed on several different FaaS platforms, or both. The various components of the tool are shown in Figure 2. The following part of this section is a brief description of the tool. PanOpticon takes as input a folder that contains:

1) XML input file: This file provides an easy way for the user to specify any application. Application details like its name, language and which cloud provider to use for deployment can be provided. As of now, the tool is capable of deploying python3 functions on AWS and GCP. Function configuration details like memory, timeout and region can be specified. A range of memory configurations can be specified and the functions are deployed and tested at each of those configuration points.

```
<serviceName></serviceName>
<runtime></runtime>
<provider></provider>
<memory></memory>
<timeout></timeout>
<region></region>
<memoryStep></memoryStep>
<memoryStart></memoryStart>
<memoryEnd></memoryEnd>
```

The application section of the file is used to describe the application in terms of the functions it is made up of, by giving details like which file it resides in and which function handler in the said file is to be executed on an event. Apart from this, details such as what is the trigger for the function and which function triggers it, i.e., a way to specify the function chain, can be provided. Chaining here means, one function performs an activity that translates into an event and the other function is configured to be triggered by that event. It is depicted in Figure 3.

```
<application>
    <function name="">
        <filename></filename>
        <handler></handler>
        <trigger type=""></trigger>
        <triggeredBy></triggeredBy>
    </function>
</application>
```

And finally, workload parameters like number of concurrent users, number of requests by each user and the time for creation of given number of users, can be specified. The application after deployment is tested using these parameters.

```
<workload>
    <noOfUsers></noOfUsers>
    <loopCount></loopCount>
    <rampUpTime></rampUpTime>
</workload>
```

2) Application code: This includes the files that contain the function code and other helper modules. With the tool, a set of invocation methods are provided specific to each provider, which aid the user in function chaining.

PanOpticon is loosely divided into three modules: deployment, workload and metrics.

A. Deployment module

This module takes the XML input file and parses it to identify all the functions and the resources that need to be deployed. A very handy framework called serverless [13] has been leveraged to deploy user application components onto

Paper / Tools	Peeking Behind the Curtains of Serverless Platform	Serverless Computing: An Investigation of Factors Influencing Microservice Performance	An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems	Evaluation of Production Serverless Computing Environments	An evaluation of open source serverless computing frameworks	Benchmarking Heterogeneous Cloud Functions	Using a Microbenchmark to Compare Function as a Service Solutions
Parameters							
Memory assigned to the function	✓	✓				✓	✓
Concurrent invocation of the same function	✓	✓	✓	✓	✓	✓	
Arrival rate of requests			✓				
Workload description[alternating periods of load and rest]			✓	✓			
Function chain length							
Auto-Scaling enabling/disabling(for open source platforms)					✓		
Benchmarks							
CPU	✓	✓		✓	✓	✓	✓
Memory		✓		✓			✓
IO	✓			✓			
Network	✓			✓			
Complex benchmark representing an FaaS application			✓				
Co-location benchmarks for different workloads	✓			✓			
Function chaining benchmark							
Metrics							
Execution time	✓	✓		✓		✓	✓
Response time	✓		✓	✓	✓	✓	
Throughput			✓	✓			
Function Runtime information	✓			✓		✓	
Data center in-Data center out time							
CPU utilization	✓						
Memory used at the platform							✓
N/W throughput	✓						
I/O throughput	✓			✓			
Ratio of successful requests					✓		✓
Cost billed at platform				✓			✓
Chaining overhead							
Performance isolation of colocated function instances	✓			✓			
Function instance idle time	✓	✓					
Container lifetime/ Function runtime instance lifetime	✓						
Scalability metric (no of VMs vs arrival rate of requests)	✓	✓					
Scalability metric (no of containers vs arrival rate of requests)	✓	✓		✓			
Cold start latency(New VM)	✓						
Cold start latency(Existing VM)	✓	✓		✓			
Warm start latency	✓	✓					
Code/configuration update delay	✓			✓			

Fig. 1. Infographic showing capabilities of existing Tools

the chosen FaaS provider. The target platforms are AWS [14], GCP [15]. The target runtime is python3 across platforms.

1) Amazon Web Services Lambda: The FaaS offering by AWS is called lambda. Each deploy-able function is known as a lambda function. The following chaining methods are supported by the tool for AWS [16]:

- Amazon S3: is an object storage service. A Lambda function is triggered when there is a file upload to a S3 bucket.
- Amazon DynamoDB: is a NoSQL database service. A Lambda function can be triggered each time a DynamoDB table is updated, i.e., a record being inserted/updated/deleted.
- Amazon API Gateway: A lambda function can also be triggered using HTTPS endpoints
- Amazon Simple Notification Service: is ideally a publish/subscribe message delivery service. A lambda function can be triggered when a message is published on a topic to which it is subscribed.
- Amazon Simple Queue Service: is a distributed message queuing service. A lambda function can be triggered when a message is present in the queue.
- Amazon SDK: A way to invoke the function directly using client libraries.

2) Google Cloud Functions: The FaaS offering by GCP is simply called Google Cloud Functions(GCF). The following chaining methods are supported by the tool for GCF [17]:

- Google Storage: A storage service by GCP similar to AWS S3.
- Google PubSub: A publish/subscribe distributed message

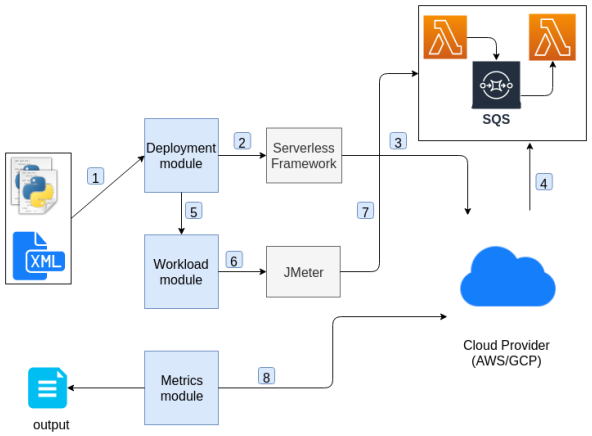

Fig. 2. PanOpticon Architecture

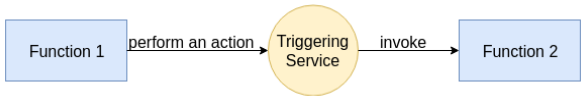

Fig. 3. Function Chaining

delivery service by GCP similar to AWS SNS.

- HTTP trigger: A cloud function can be triggered using a HTTP endpoint.

B. Workload module

This module helps simulate real life workload scenarios for the application. At the core of the module, we use Apache JMeter [18]. It has the ability to load and performance test many different protocols like HTTP, HTTPS. We configure the first function in the application chain with a HTTPS endpoint and use that to load test the application. JMeter's Thread Group is used which is a set of threads(simulating a number of users) that trigger a function a specified number of times. Ramp up time specifies how long JMeter takes to create all the threads. JMeter essentially performs closed loop performance testing, i.e., a number of users are simulated and each user sends a request to the function, waits for a response and then fires the next request. The configurable parameters in PanOpticon are number of users, number of requests each user sends and the ramp up time. These are mentioned in the same XML file that is used to describe the application.

C. Metrics module

This module is used to collect useful metrics like execution time and memory usage of every function and latency of each chain component in the application. Using this information, an estimate of the data center in and data centre out time(Time spent by the application inside the provider's platform to service a request) can be estimated. This does not include the network latency and hence is location agnostic. Also, an estimate of cost of a single request can be calculated using the above function execution metrics.

1) Amazon Web Services Lambda: The function execution time and memory usage is extracted from logs that are written by the lambda function to CloudWatch Logs [19] and averaged. AWS X-Ray [20] is distributed tracing system that is used to debug applications built using the microservices architecture. It allows us to see the end-to-end path of an request to the application. Using AWS X-Ray, the tool captures the time spent in individual services that make up the application and hence gives an estimate of the time taken to service the request by the platform. The following metrics are reported for AWS: 1) Function execution instance metrics: Average execution duration, average billed duration, memory size configured for the function and average maximum memory used by the function are reported for every function that make up the application. 2) Average time spent inside every AWS service that make up the application: This includes services of type AWS::Lambda, AWS::S3, AWS::SNS, AWS::SQS and AWS::DynamoDB.

2) Google Cloud Functions: Stackdriver Monitoring [21] service by GCP is used to extract the mean function execution time and memory usage. Stackdriver Trace [22], a distributed tracing service by GCP, is used to trace a certain parts of the application, specifically, the invocation/triggering mechanism. Stackdriver Trace for Python applications is enabled by using OpenCensus [23]. OpenCensus is a set of instrumentation libraries for collecting trace and metric data. These two together have been used to find the latency of the individual triggers, i.e., the time taken by a function to perform an action that in turn triggers another function. The following metrics are reported for GCP: 1) Function execution instance metrics: Average execution duration and average memory used by the function are reported for every function that make up the application. 2) Average latency of triggers in the application of type Google Storage, Google PubSub and HTTP.

In summary, Figure 2 shows the end-to-end workflow of PanOpticon: 1) Deployment module takes as input the XML file and the functions code and modules 2) It parses the XML file to identify the appropriate resources and functions of the application and creates a YAML file which is consumed by serverless framework 3) and 4) serverless framework uses the cloud providers APIs to deploy all the resources previously identified 5) Deployment module provides the workload module with the deployment details 6) Workload module inserts workload information in a JMX file and feeds it to JMeter 7) JMeter appropriately tests the application 8) Metrics module uses cloud provider APIs to fetch useful metrics of function execution instances and also metrics information of the additional services used in function chaining.

IV. Experiments

In this section, we detail a set of experiments to showcase the capabilities of PanOpticon we have described above. The first experiment showcases the basic set of capabilities of the tool like deploying a function at different memory slots to perform a comparative study of their performance across the slots and across platforms. We then look into the effects of

function chaining on serverless workloads. We declare via a Markup file the chaining workflow, which the tool then automatically deploys to test the different parameters. We also look into different trigger types like publisher-subscriber based triggers, HTTP based triggers, etc., and their comparative performance. We also showcase a simple chat server which we test via the tool to get the metric data. For the experiments, two systems were used alternatively to run the tool. First system had the following configurations: 3.6GHz Intel i7-7700 8-core CPU, 7.7GB RAM, Ubuntu 16.04 LTS. Second system had the following configurations: 2.2GHz Intel i5-5200U 4-core CPU, 7.7.GB RAM and Ubuntu 18.04.3 LTS. The final experiment aims at benchmarking AWS and GCF platforms and was performed on a server with the following configuration: 2.10GHz Intel(R) Xeon(R) E5-2683 v4 64-core CPU, 128GB RAM, Ubuntu 18.04.3 LTS.

A. *Testing functions at different memory allocations*

Back et al [24] suggests that increase in memory allocation increases performance of serverless workloads due to a proportional increase in CPU allocations. Our tool allows end users to test their functions at different memory allocation limits to find the best performance - cost balance. To showcase such a scenario, we deployed a sample application on AWS consisting of three functions chained together(function1-function2-function3) that each calculate factorial of a large number. We chose factorial to simulate CPU intensive work that a function might do before going on to invoke the next function in the chain. The tool was set to deploy the sample application at 256 MB, 512 MB, 768MB, 1024 MB memory allocations. The application was tested using 1000 requests(10 users, 100 requests each). The function latency observations are shown in Figure 4. We can observe that as memory allocated to each function increases, the function execution time decreases owing to the additional CPU allocated.

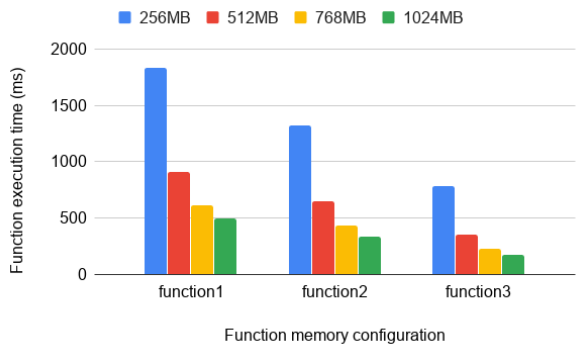

Fig. 4. Experiment consisting of deploying functions at various memory configurations and measuring function execution time

B. *Comparison of latency of different chaining mechanisms*

As described in the previous section, our tool supports six chaining mechanisms for AWS and three chaining mechanisms for GCF. In this experiment, we set out to measure the latency of each of these chaining mechanisms. To achieve this, applications of chain length two were deployed i.e. function 1 invokes function 2 via a chaining mechanism. A different application was deployed for each chaining mechanism. And finally, each application was tested at three different workload settings- 10 users firing 10 requests each, 50 users firing 10 requests each, 100 users firing 10 requests each. The average time spent in each triggering service of AWS is shown in Figure 5 and the average time taken by the function to send a request to the triggering service plus the time taken inside the triggering service of GCF is shown in Figure 6. In AWS, S3 trigger takes the highest time and DynamoDB trigger takes the least time. In GCF, Google storage trigger takes the highest time and Google PubSub trigger takes the least time.

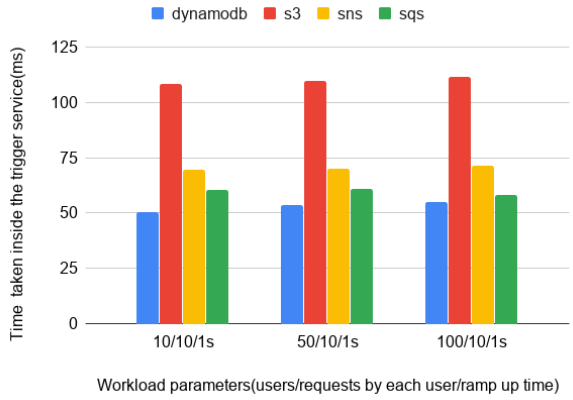

Fig. 5. Experiment to measure the time spent inside a triggering service(AWS)

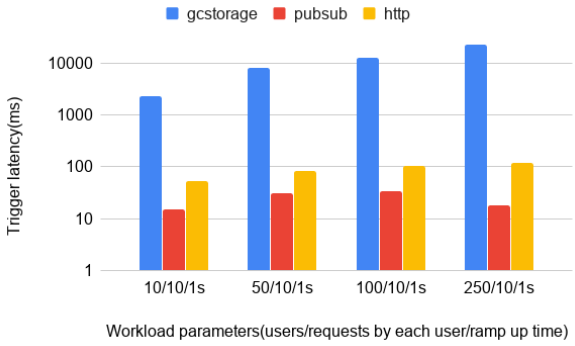

Fig. 6. Experiment to measure trigger latency(GCF)

C. *Capacity tests*

This experiment was done to showcase the workload module of the tool. A sample application(function 1 invoking function 2 using a HTTP enpoint) was deployed on GCP and load tested at three different configurations: 25 users sending 40 requests each, 25 users sending 70 requests each, 25 users sending 100 requests each. The function execution time of the functions and the latency of the trigger, in this case, HTTP POST request is reported in Figure 7.

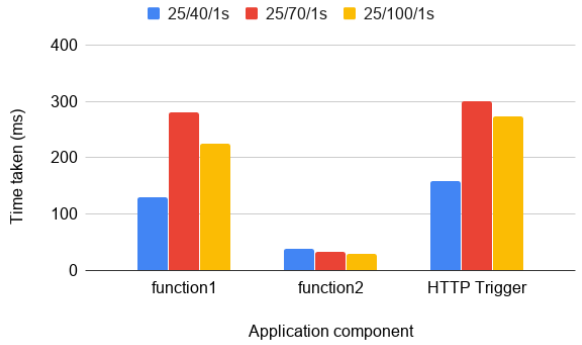

Fig. 7. Experiment to showcase testing an application using different workloads(GCP)

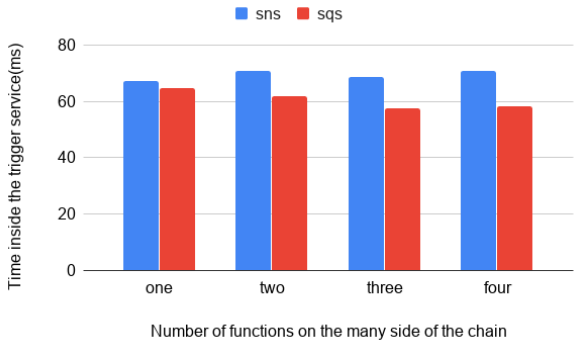

Fig. 9. Experiment to see the effect(if any) of one to many chains on time spent inside the triggering service

D. One to many function chains

PanOpticon also has the capability of deploying one to many chains. A one to many chain consists of one function invoking a trigger point that is tied to two or more functions. It is as shown in Figure 8. We wanted to see the effect of one to many chains on trigger latency. The target chaining mechanisms were SNS and SQS of AWS. The many side of the function chain had 2,3,4 functions. The applications were subjected to a workload of 10 users and 100 requests each. As seen from Figure 9, the triggering latency doesn't differ as the number of functions on the many side are increased. One interesting observation is that for SNS trigger, x instances of function 1 triggered x instances of function 2,...,n. But in the case of SQS, x instances of function 1 triggered x_i instances of function i where $\sum_{i=2}^{n} x_i <= x$. This shows that some sort of bottleneck was hit for SQS and few messages were dropped from the queue.

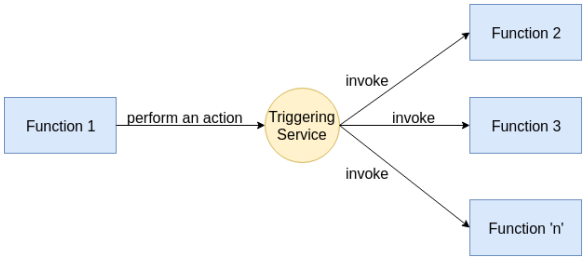

Fig. 8. A one to many chain

E. Deploying a sample application on AWS and GCP

To demonstrate the effectiveness of the tool in comparing platforms, we deployed a chat server application on AWS and GCP. The application consists of two HTTP endpoints to register users and login to the system, which uses a JWT based token authentication to authenticate users. A third HTTP endpoint is provided to send messages among users. Finally, a fourth HTTP endpoint is provided which is called at periodic intervals to backup user credentials to a storage

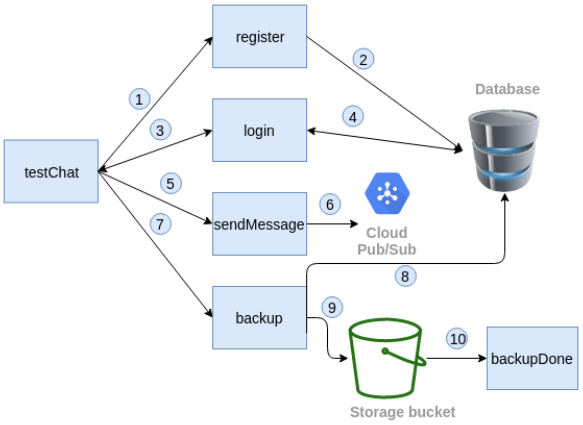

Fig. 10. Sample Chat Application

bucket. Figure 10 shows the modular architecture of the chat application. Apart from the Database and pub/sub topic, all other components were deployed using our tool. The functions were allocated 1024MB. The chat application was tested using testChat function that performed registration, login, sending message and backup one after the other. One end-to-end test consists of: 1) A HTTP request is sent to register function with a username password pair, 2) register function puts the pair in a database, 3) and 4) A HTTP request is sent to login function with an existing username password pair, login checks the database for its existence and if it does exist, then it returns a JWT token, 5) A HTTP request is sent to sendMessage function with the JWT token and message, 6) On successful verification of the token, the message is published to a pub/sub topic, 7) A HTTP request is sent to backup function, 8) backup function fetches all the username and password pairs from database, 9) And uploads them to a storage bucket, 10) backupDone function is triggered once the upload is done. The application was tested using 10 users sending 100 requests each to testChat. The function execution time and memory usage of all functions deployed on AWS and GCP are plotted in Figure 11 and Figure 12. We can see that similar amounts of memory was used on both platforms.

But the function execution time is considerably lower in case of AWS.

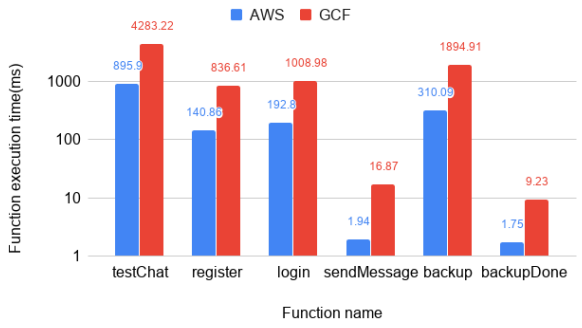

Fig. 11. Function execution time of each function in the sample application

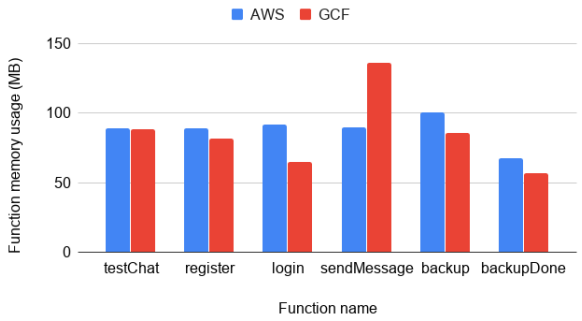

Fig. 12. Function memory usage of each function in the sample application

F. AWS and GCF benchmarking

As a final experiment, we wanted to benchmark AWS and GCF platform's throughput and average latency. As already mentioned, JMeter performs closed loop testing. Hence to perform open loop performance testing, we used loadtest [25]. Using its parameters, we set the arrival rate of requests. The number of requests were adjusted to make sure the experiment lasts approximately 200s. The tool reported throughput, average latency, number of errors and total time elapsed. Using this we calculated throughput of successful requests(goodput). We repeated the experiment for a no-op function, a CPU intensive function(calculates factorial of a large number) and a memory intensive function(sequential writes to a large array). Figure 13 and 14 shows the goodput achieved for AWS and GCF for the three functions. As seen from these, goodput gradually increases for small loads and flattens out eventually at saturation. The highest goodput observed for AWS is 496 req/s for the no-op function and 383 req/s for GCF for the no-op function. Figure 15 and 16 shows the average latency observed for AWS and GCF for the three functions. This calculation includes both error-ed and successful requests. Figure 17 and 18 shows the number of error-ed requests vs the arrival rate of requests. These were of two types: one type was due to throttling of requests by the platform due to reaching a

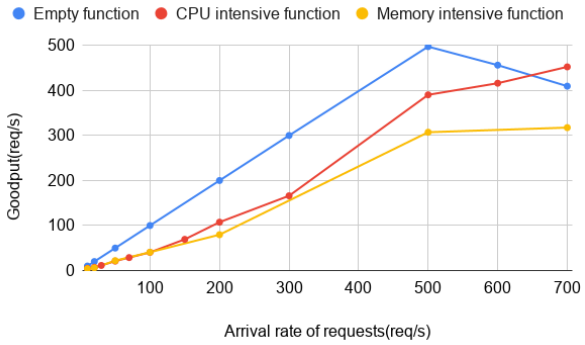

Fig. 13. AWS throughput of successful request(Goodput)

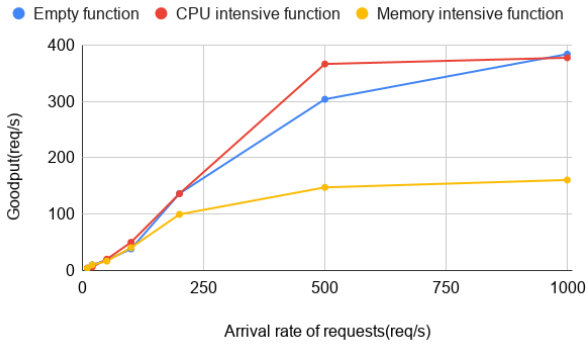

Fig. 14. GCF throughput of successful request(Goodput)

maximum limit of sorts and second error was due to a cause unknown, we speculate that it was due to requests unable to get into the platform side queue or due to loadtest's inability to sustain the request generation. The first type of error was only seen in AWS as AWS has a maximum concurrency limit of 1000, i.e., only a 1000 function instances can be active at once. We can also see that the number of errors increase as we increase the arrival rate of requests.

V. Conclusion

In our article, we discussed the need for a comprehensive tool for efficient testing of serverless workloads spanning technologies and service providers. We then introduced our tool, PanOpticon, which provides a comprehensive set of features to deploy end-user business logic across platforms at different resource configurations for fast evaluation of their performance. Finally, to test the effectiveness of our tool, we conducted a set of experiments testing separate features in isolation. An experiment comprising of a chat server application was conducted to test the effectiveness of the tool in complex logic scenarios. And finally, we show some benchmarking test results of AWS and GCP. As an extension to this work, we plan to support more platforms in the near future.

References

[1] "Companies using serverless in production." [Online]. Available: https://dashbird.io/blog/companies-using-serverless-in-production/

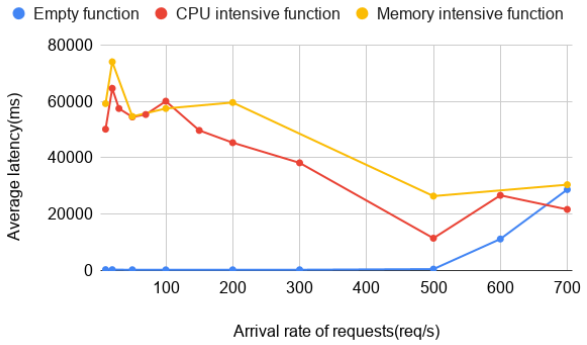

Fig. 15. Average latency of a request(AWS)

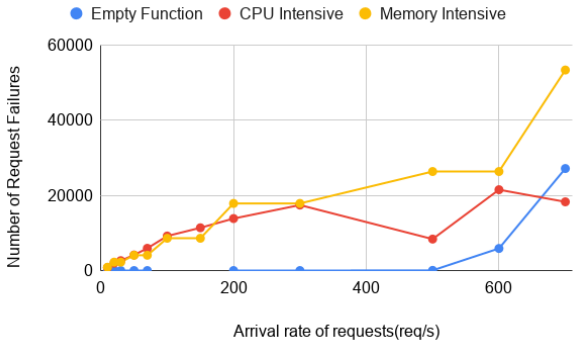

Fig. 17. Number of errors observed during experiment(AWS)

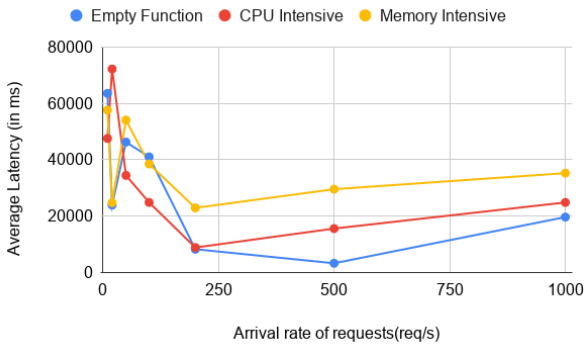

Fig. 16. Average latency of a request(GCF)

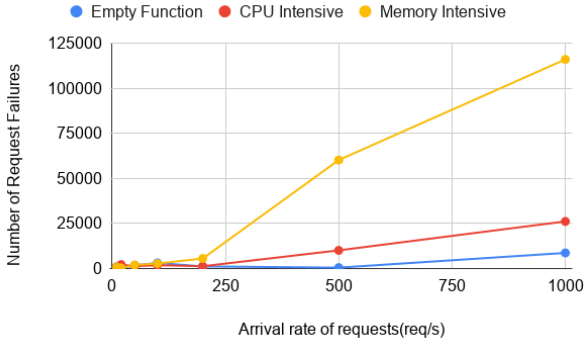

Fig. 18. Number of errors observed during experiment(GCF)

[2] "Aws lambda customer case studies." [Online]. Available: https://aws.amazon.com/lambda/resources/customer-case-studies/

[3] "Case studies of aws serverless apps in production." [Online]. Available: https://winterwindsoftware.com/real-world-serverless-case-studies/

[4] "Serverless development companies." [Online]. Available: https://serverless.com/partners/

[5] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 133–146.

[6] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169.

[7] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 442–450.

[8] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 3–18.

[9] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *European Conference on Parallel Processing*. Springer, 2017, pp. 415–426.

[10] B. Balis, "Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows," *Future Generation Computer Systems*, vol. 55, pp. 147–162, 2016.

[11] S. K. Mohanty, G. Premsankar, M. Di Francesco *et al.*, "An evaluation of open source serverless computing frameworks." in *CloudCom*, 2018, pp. 115–120.

[12] N. Kaviani and M. Maximilien, "Specserverless." [Online]. Available: https://docs.google.com/document/d/1e7xTz1P9aPpb0CFZucAAI16Rzef7PWSPLN71pNDa5jg/edit

[13] "Serverless framework." [Online]. Available: https://serverless.com/

[14] "Amazon web services." [Online]. Available: https://aws.amazon.com/

[15] "Google cloud platform." [Online]. Available: https://cloud.google.com/

[16] "Aws lambda triggers." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html

[17] "Gcf triggers." [Online]. Available: https://cloud.google.com/functions/docs/concepts/events-triggers

[18] "Apache jmeter." [Online]. Available: https://jmeter.apache.org/download_jmeter.cgi

[19] "Aws cloudwatch logs." [Online]. Available: https://aws.amazon.com/cloudwatch/features/

[20] "Aws x-ray." [Online]. Available: https://aws.amazon.com/xray/

[21] "Gcp stackdriver monitoring." [Online]. Available: https://cloud.google.com/monitoring/

[22] "Gcp stackdriver trace." [Online]. Available: https://cloud.google.com/trace/

[23] "Opencensus." [Online]. Available: https://opencensus.io/

[24] T. Back and V. Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," in *European Conference on Service-Oriented and Cloud Computing*. Springer, 2018, pp. 146–160.

[25] "loadtest." [Online]. Available: https://www.npmjs.com/package/loadtest