Measuring Performance of Google Cloud Serverless Applications

Shophine Sivaraja
Department of Computer Science
University of Georgia
Athens, GA, USA
shophine@uga.edu

Siraaj Fadoo
Department of Computer Science
University of Georgia
Athens, GA, USA
saf33030@uga.edu

***Abstract*—Serverless computing has garnered a lot of attention over the past few years. The advantages are enticing: not having to worry about provisioning virtual machines, scaling, etc. so that developers can instead focus on code logic. A large vendor of these serverless functions, and one in which we chose to focus on in this paper, is Google with their Google Cloud Functions service. There is some information publicly available on the performance of Google Cloud Functions compared to other Cloud providers' functions (like Amazon Web Services Lambda, Microsoft Azure Serverless), as well as benchmark tools to measure Google Cloud function performance under various configurations. However, there is not much information on the performance of holistic, serverless applications deployed on Google Cloud, so we wanted to explore this. This includes end-to-end response time and performance expectations. In addition, we composed our serverless applications of purely API calls so that we could determine if the end-to-end response time was similar to the sum of API's response time, or if there was something in the Google Cloud Platform hindering performance.**

The reason why we are interested in this performance information is because it would be helpful for developers to know before deciding to use the platform. Currently, the Google Cloud Functions Service Level Agreement (SLA) only includes information on uptime percentage and error rate [1]. With information on serverless application performance, developers can not only know how reliable the service is, but also how well it performs with realistic applications.

***Index Terms*—Google Cloud, Functions, Serverless, Application, Response Time, Service Level Agreement (SLA)**

I. Introduction

As anyone in the Cloud research field would note, serverless computing presents many advantages to traditional methods of deploying an application in the Cloud. Scaling, provisioning of resources, and basically all low level infrastructure management is abstracted from developers so that they only have to worry about business logic. Another major advantage is cost: the model is similar to pay-as-you-go, so there is no paying for idle resources. The more you execute the function, the more you pay.

Because of these advantages, serverless computing has grown rapidly in recent years. In 2017 it was predicted that the serverless market size would grow to $7.72 billion by 2021 [2]. Moreover, in 2018 it was noted that serverless computing had an annual growth rate of 75% [3].

Despite serverless computing's advantages and astonishing growth, there is still a lack of information on its performance, specifically when it comes to holistic applications. Most serverless functions are used to perform modular, microservice type functionalities. Some example use cases include multimedia transformation, backend API's for mobile applications, and Cloud database automation tasks [4], [5]. What has also been noted however, is a trend towards more essential use cases for serverless functions, and we believe even the ability to build entire web applications using them [3]. When it comes to full applications deployed on Google Cloud functions [12], there is indeed a lack of publicly available information on what kind of performance to expect. Moreover, Google's Cloud Functions SLA focuses on uptime percentage and error rate [1]. While this can be useful information for developers, it would also be very helpful to know what kind of serverless application performance to expect. This can inform them on whether an application is feasible to develop on Google Cloud Functions or not, or even if a competing platform's serverless option is a better choice.

Because of the lack of information when it comes to Google Cloud serverless application performance, we chose to develop a series of applications ourselves, deploy them on Google Cloud Functions, and test them extensively.

A. Approach

Our approach for measuring the performance of Google Cloud serverless applications focused on execution response time. We developed 4 serverless applications: 3 were of similar complexity while the last 1 had more API calls. The names of each application are "Dog Classifier," "City Information," "Movies," and "Car Pooling." All of the serverless applications we developed included purely API calls. After one API was called, the callback function would then invoke another API, etc. until enough data was gathered to send an HTML response back. We measured the execution time of each API and the end-to-end execution time of the application.

We ran each application 50 times to gather the data. As we measured execution times and then analyzed the results, we looked for several things. One, we created normal distributions for each API to determine their skew. We also took the sum of the API execution times and compared this number to the end-to-end execution time. Assuming no loss in execution time due to backend processing, the end-to-end response time should

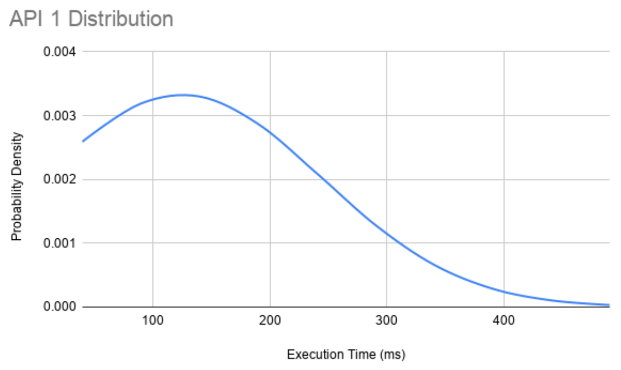

Fig. 1. API 1 distribution for Car Pooling application.

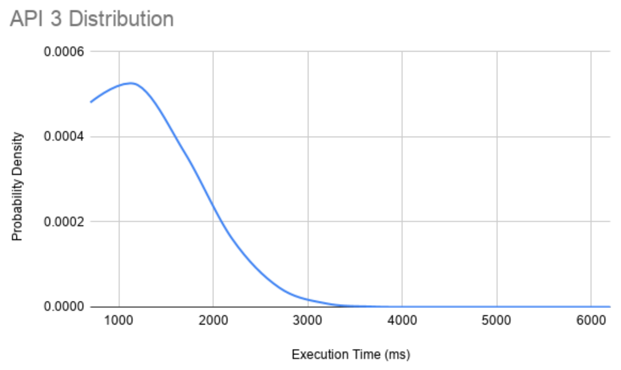

Fig. 2. API 3 distribution for City Information application.

be equal to the sum (there was almost no code logic besides the API calls). We looked closely at how the end-to-end response times and sums differed. This included comparing normal distributions between the end-to-end response times and sums. In addition, we calculated the Root Mean Square Error (RMSE) to determine how off the actual times (end-to-end) were from the expected times (sum of API's).

B. Results Overview

There were several trends we noticed in the results that we gathered. For one, most of the individual API distributions were skewed right - illustrating a lower limit. In addition, the spread of end-to-end response times was much closer to the spread of sum of API's times for our most complex application, Car Pooling. This application included 10 API calls while the other ones included 4. What we can roughly conclude from this is that the more API calls in a serverless application, the more the performance aligns with what is to be expected. However, developing additional serverless applications of increasing complexity (more API calls) would be required to verify this conclusion. This is discussed more in the "Limitations" subsection of Section V. ("Discussion").

The other conclusion we were able to come to after analyzing our results was that application performance was fairly in line with what was to be expected. The end-to-end response times were close to the sum of API's response times in 3 out of the 4 of the applications. This is shown in RMSE values of 6.55 ms, 7.41 ms, and 16.93 ms in the Dog Classifier, Movies, and Car Pooling applications respectively. However, the City Information RMSE was 835.62 ms. The reason for this was mainly due to a large outlier in one trial run in which the actual and expected response times differed by 140.13%.

Overall, we found that Google Cloud serverless applications performed fairly up to expectations, increasingly so for a more complex application.

II. Related Work

There are several other research papers related to our work. One that influenced our initial idea was Jayathilaka et al.'s measurement of web-facing API's exported from PaaS apps [7]. In this research, it is noted how PaaS SLA's focus on availability rather than performance, and we wanted to apply this to serverless computing [7]. In Jayathilaka et al.'s study, response time predictions were made for individual API's as opposed to our research which also looked at end-to-end response times [7]. In addition, their platform of choice was PaaS, while we chose to focus on serverless [7].

Another related study by Maissen et al. developed a benchmark suite to measure the performance of serverless functions on different Cloud providers [6]. The providers they studied include Amazon Web Services (AWS) Lambda, Microsoft Azure Functions, Google Cloud Functions, and IBM Cloud Functions [6]. Within each of these platforms, various runtime environment and memory size configurations were tested [6]. We chose to focus on one platform (Google Cloud Functions) with Node.js 12.0 as our runtime environment and the same memory size allocated to each function instance. Maissen et al.'s study also measured the performance of Cloud functions in the form of microservice like tasks as opposed to full web applications. For instance, one function titled "faas-fact" was used to factorize an integer and perform matrix multiplication [6]. In our case, we developed holistic web applications with a front-end and back-end and set of realistic functionalities. Finally, we factored in API response time into the end-to-end response time of the application, as opposed to just function execution time.

Work done by Kim et al. also recognizes the lack of realistic application performance testing for serverless functions [8]. In this paper, the researchers test machine learning, image processing, and other common workloads on several cloud provider serverless platforms like AWS Lambda and Google Cloud [8]. However, the testing done here is latency for each workload as opposed to end-to-end application latency [8]. In addition, the differences between cloud providers and memory size are looked at whereas we exclusively focused on Google Cloud Functions [8].

III. Approach

In order to come to some generalizations on Google Cloud serverless application performance, we wanted to make mul-

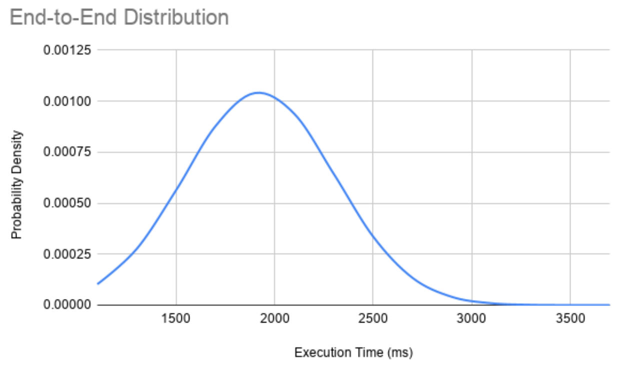

Fig. 3. Observed response time for Dog Classifier application.

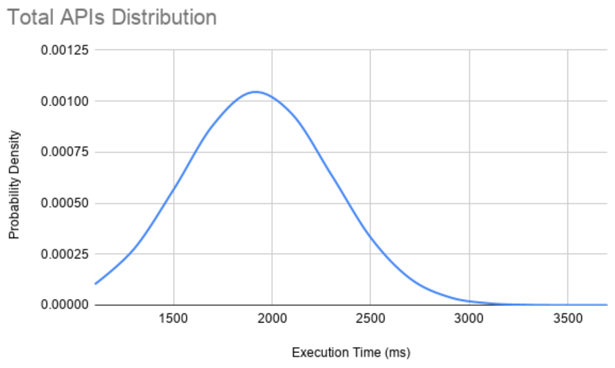

Fig. 4. Expected response time for Dog Classifier application.

tiple example applications. Furthermore, these applications should use different API's and have different complexities. In total, we created 4 serverless applications. They are as follows:

A. *Dog Classifier*

The Dog Classifier application generates a random dog image, resizes the image, uses machine learning to classify what it is (Google Cloud's Vision API), and then writes the results to a Cloud database (Google Firebase). In total, 4 API's are called.

B. *City Information*

The City Information application takes a city parameter and then displays useful information about it. This includes the current temperature, the sunrise and sunset times, and the latitude and longitude. Finally, the results are written to a Cloud database. In total, 4 API's are called.

C. *Movies*

The Movies application displays information about a specific movie, as well as current movies in theatre, upcoming movies, and popular movies. In total, 4 API's are called.

D. *Car Pooling*

The Car Pooling application displays directions and information about an example destination like currency, population, nearest cities, etc. Then a confirmation email is sent to the user and the results are written to a Cloud database. In total, 10 API's are called.

All 4 of these applications were written in JavaScript in a Node.js 12.0 environment. In addition, all of our serverless applications used a memory size of 256 MiB which translates to roughly 268 MB. We wanted to run all of our functions in the same runtime environment and with the same memory allocation so that these factors did not affect the performance. Instead, our goal was for the code complexity and difference in API's to be the main factor affecting performance. Lastly, we used HTTP as our trigger method to invoke the functions.

We developed and tested each application locally using the Google Cloud "Functions Framework for Node.js" module [10]. Once satisfied with our application, we then deployed the functions on Google Cloud and conducted our tests from there.

As previously noted, each application is composed of purely API's. After one API is called, the next API is immediately invoked in the callback function [14]. This allowed us to calculate precise response times for each API, as well as the end-to-end response time when the last API callback function was hit. In order to calculate the response times, we took advantage of the Console object in JavaScript. In particular, this object contains 2 methods, console.time() and console.timeEnd(), that we used to measure latency [9] [13]. These 2 functions can also take a string parameter denoting a label so that multiple timers can be running at the same time [9]. For instance, we invoked the end-to-end response time timer with console.time("end-to-end"), and when the final API callback function was invoked, we ended the timer with console.timeEnd("end-to-end"). Through our research, we found using these 2 methods from the Console object to be an accurate way to calculate response time.

The calculated times were exported to the console, which Google Cloud provides an interface for named the "Logs Explorer." To actually run the serverless application, we created a script that automatically triggered the HTTP function. After this, we went back to the Logs Explorer, downloaded the logs to CSV, and then filtered them out with the console.time labels we created using regular expressions. This was definitely a necessary step as the logs had an abundance of other information such as project id, build id, log name, and timestamps. Moreover, we chose to use a script to run the application as it made the testing process much quicker than manually refreshing an HTTP link in the browser [11].

We chose to run each application a total of 50 times. In order to counteract outliers not representative of the true performance of the application, as well as warm and cold start, we wanted to run the application a plethora of times. Fifty was a number we found to be sufficient enough to mitigate some of these factors that may skew our results.

When running our application and collecting response times, there were several things we were looking for. Firstly, we looked at how the performance of each API within an appli-

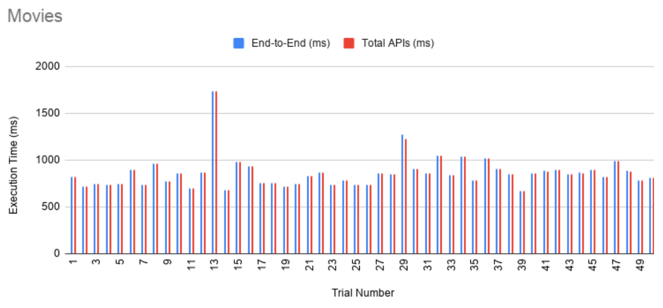

Fig. 5. Observed vs. Expected response times for Movies application.

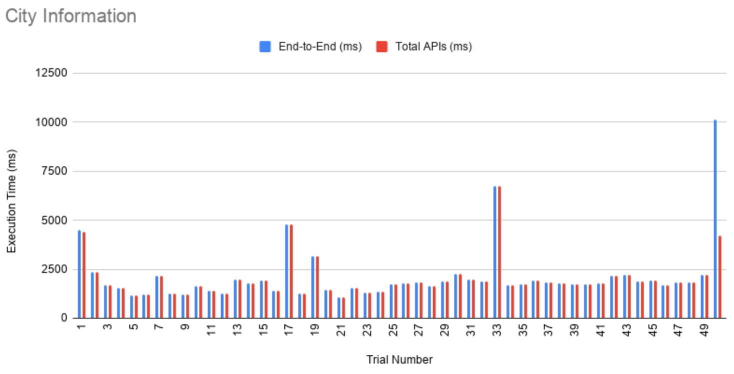

Fig. 6. Observed vs. Expected response times for City Information application.

cation was distributed. Looking at the skew of the distribution could point us to whether there was a lower or upper limit to the execution time. Next, we looked at if the end-to-end response time was similar to the sum of the API's response time of the application. For statistical purposes, we used the end-to-end response time to represent our "actual" or "observed" value. In contrast to this value, we used the sum of API's time to represent the "expected" value. This was because the applications were purely API calls, so theoretically with no extra latency from the Google Cloud Platform, the total execution time should be equal to adding up the execution times of all of the API's (total is equal to sum of parts). We were particularly interested in this comparison because if the observed and expected response times of the applications did not align, then we would know that there was something in the Google Cloud Functions platform that was affecting performance.

IV. Experiments

All response times were measured in milliseconds and to the thousandths decimal place. In addition to measuring 50 executions times for each API in each application, as well as the end-to-end and sum of API's times, we calculated overall means, medians, and standard deviations. The standard deviation in particular allowed us to see how spread out the data was from the average. Calculating these values also allowed us to create normal distributions for API's, end-to-end, and sum of API's performance. Lastly, we calculated the Root Mean Square Error (RMSE) between the observed and expected response times of each application. We also contemplated calculating the Mean Absolute Percentage Error, but we found that this was a similar value to RMSE but typically seen as less accurate.

Some example data is shown in the various figures throughout this paper. As you can see in Figures 1 and 2, normal distributions were formed for each API. Moreover, we created normal distributions for the observed and expected application response times, again in order to compare their similarity. This can be seen in Figures 3 and 4. Finally, we created graphs for each application comparing the end-to-end and total API response times to visually denote any discrepancies. These can be seen in Figures 5 and 6.

One discrepancy can be seen in trial number 50 for the City Information application (Figure 6). While the actual end-to-end response time was 10,124 ms, the total API time (expected) was 4,216.039 ms. This was a 140.13% difference between observed and expected times. This was also such a high outlier that it affected the distributions of the City Information observed and expected response times.

Most of the individual API distributions were skewed right. In addition, the distributions for observed and expected response times of the applications were skewed right for the City Information and Movies apps but more evenly spread out for the Dog Classifier and Car Pooling apps.

In terms of RMSE, the Dog Classifier, Movies, and Car Pooling applications had fairly small values of 6.55 ms, 7.41 ms, and 16.93 ms respectively. The City Information application on the other hand had an RMSE of 835.62 ms. Finally, the differences in standard deviation between observed and expected response times were .38%, 48.32%, and 1.50% for the Dog Classifier, City Information, and Movies applications respectively. The Car Pooling application had a difference in standard deviations of .13%.

V. Discussion

There were several generalizations we came to after analyzing our data results. Firstly, most of the individual API distributions were skewed right. What this tells us is that there is probably a lower limit to how quickly they can perform. This might represent a limit on how quickly Google Cloud Functions can make the call, but more likely it is a limit on the server side of each API in terms of how quickly it can serve a request. The distributions for observed and expected times for each application were also skewed right in 2 instances (City Information and Movies) and had little skew in the other 2 cases (Dog Classifier and Car Pooling). Again, the right skew is probably from their being a lower limit on each of the API's composing the apps, which results in a lower limit on the overall response time.

Another conclusion we were able to come to was related to the standard deviations of observed and expected times.

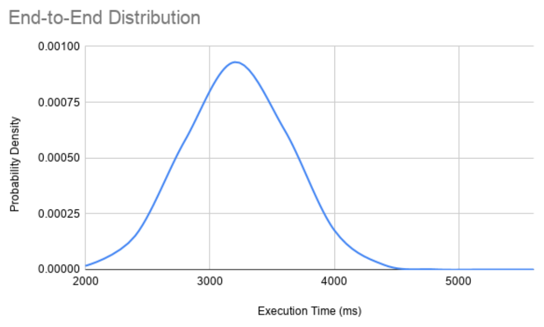

Fig. 7. Observed response time for Car Pooling application.

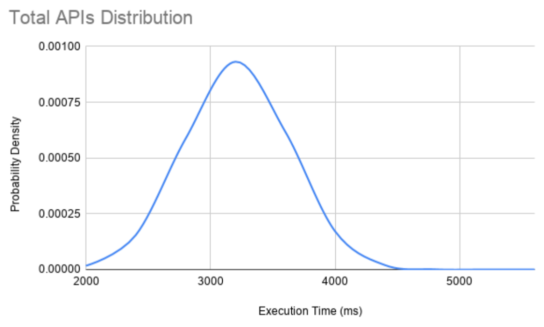

Fig. 8. Expected response time for Car Pooling application.

We found that the spread of data between observed and expected times was closer for our most complex application, Car Pooling, than for the other applications. This is illustrated in the difference between observed and expected standard deviations of .13% in the Car Pooling application, as opposed to .38%, 48.32%, and 1.50% in the other apps. As previously stated, the Car Pooling application has 10 API calls while the other ones have 4. Because of this additional code complexity, we surmise that this is the reason for the spread of data of observed and expected times to be closer than for our other applications. In other words, the more complex a Google Cloud serverless application, the more in line the performance should be with expectations.

Finally, the overall Google Cloud serverless application performance we measured was fairly in line with expectations. This can be seen in the RMSE values of 6.55 ms, 7.41 ms, and 16.93 ms for the Dog Classifier, Movies, and Car Pooling applications respectively. These are all small differences between the observed and expected end-to-end response times. The City Information application had a substantially higher RMSE value of 835.62 ms. However, this is mainly due to the 50th trial run in which the observed application response time was 140.13% higher than the expected time. We predict that this was an error on the Google Cloud Platform side, and some facet of this backend held up the application from completely finishing.

All in all, we found that the applications we developed using Google Cloud Functions performed fairly up to expectations, and with increased accuracy for our more complex application.

A. Limitations

There are several potential limitations to our work that should be acknowledged. In our research we assumed that the end-to-end and sum of API's response times would be equal, and if they were not we attributed it to the Google Cloud Functions platform. However, there were minimal amounts of code that also could have been causing this difference. For instance, after an API was called in each application, oftentimes a variable in JavaScript would be instantiated before the next API call. These small lines of code when added together could be causing the difference in observed and expected times.

One generalization we made that could be stronger is that the more complex a serverless application, the more accurate the performance should be on the GCP. We came to this conclusion after comparing the standard deviations of the Dog Classifier, City Information, and Movies apps (all of less complexity - 4 API's) to the Car Pooling app (more complexity - 10 API's). However, because this is just one level of increased complexity, this could just be a coincidence. We would need to develop additional applications of various levels of complexity to really gauge how it affects performance accuracy. For instance, developing applications of 8 API calls, 15, 20, and 30 would be helpful to corroborating our conclusion.

An assumption we made was that the Console object in JavaScript that we used to measure response time was accurate. However, if this object was logging to the console at some degree of error, this could cause differences in the observed and expected times that we were seeing. There is some debate as to which JavaScript mechanism provides the most accurate timing information. One alternative to the Console object is the Performance interface with its performance.now() method. Another mechanism to time in JavaScript is by using the Node.js function process.hrtime(). Each of these timing mechanisms provide slightly different results and depending on how accurate each one is, might also lead to false conclusions. Therefore, we wanted to note this potential limitation in our research.

Lastly, in our experiments, we only tested under 1 runtime environment (Node.js 12.0) and with 1 memory configuration (256 MiB). If we wanted to truly generalize Google Cloud serverless application performance, we would need to test functions under different configurations as well. Other runtime environments or memory sizes could lead to different results than the ones we gathered, so this is another potential limitation to our research.

B. Further Work

Most of the further work we wish to do addresses the above limitations. First, we would like to minimize the small amount of code in our applications that could be causing differences in response times. If we were not able to completely eliminate this code, then we would like to at least account for their

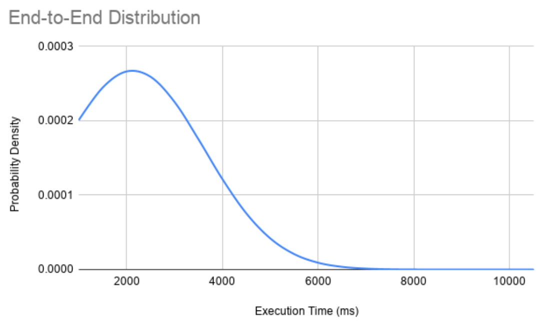

Fig. 9. Observed response time for City Information application.

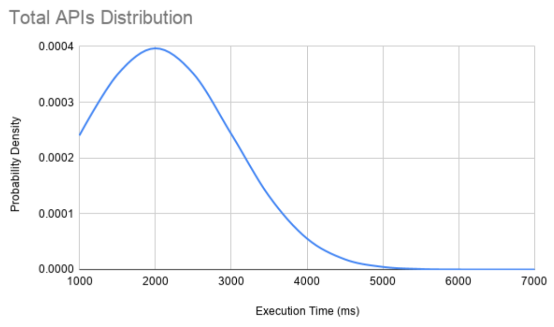

Fig. 10. Expected response time for City Information application.

execution time so as to get a more accurate read on the overall application performance.

Second, we would like to develop additional serverless applications. This includes more complex applications with more API calls and with different functionalities and types of API calls (machine learning, database, etc.).

To address the potential timing errors, we would like to investigate the other mechanisms (Performance interface and process.hrtime()) further to ensure a more accurate assessment of performance. Finally, we would like to test our applications under multiple runtime environments and memory sizes to determine whether this has any effect on Google Cloud serverless application performance.

Outside of improving our experiments when related to Google Cloud, we are also interested in serverless application performance in general. This means developing applications and measuring their performance on other Cloud platforms like AWS Lambda, Microsoft Azure Serverless, and IBM Cloud Functions. We are curious if we would see similar results on these other platforms, like for instance if an increasingly complex serverless application deployed on AWS Lambda would also lead to observed and expected response times being more in line with each other. One of the motivations in our research was to inform developers thinking about creating a serverless application on Google Cloud, but with additional work on other platforms, a developer could be even more informed and know which platform to jump to if Google Cloud is not viable.

Overall, serverless computing is a fairly new field with lots of room for improvement and exploration, so there are many more use cases and further work to be done. Our application code is open source and currently available on Github.

VI. Conclusion

In this paper, we attempted to address the lack of information for developers on Google Cloud's serverless application performance. In particular, we found this lack of information in the Google Cloud Functions Service Level Agreement.

To measure serverless application performance on the Google Cloud Platform, we developed 4 applications and benchmarked their response times. In addition, we composed each application of purely API calls. The reason we did this was to measure the response time of each API call and then total all of them together. Theoretically, the end-to-end response time (first API call to callback of last API call) should be equal to this total, so we used the end-to-end and sum of API's response times as the observed and expected values. If there was a difference in these values, we attributed it to the GCP.

After developing the applications and testing them extensively, we found that for the most part, the serverless application performance was in line with what was to be expected. Moreover, we found that with the more complex application we developed, the performance was even more in line with expectations.

For future work, there are several steps we would like to take like accounting for extra code within the applications, developing additional applications of various complexities, and exploring additional Cloud platforms like AWS Lambda, Microsoft Azure Serverless, and IBM Cloud Functions.

References

[1] "Cloud Functions Service Level Agreement (SLA)." *Google Cloud*, Google, 9 Jan. 2020, cloud.google.com/functions/sla.

[2] Castro, Paul, et al. "The Rise of Serverless Computing." *Communications of the ACM*, Dec. 2019, pp. 44–54.

[3] Null, Christopher. "The State of Serverless: 6 Trends to Watch." *TechBeacon*, Micro Focus, 22 Jan. 2019, techbeacon.com/enterprise-it/state-serverless-6-trends-watch.

[4] "Serverless Advantages and Use Cases." *Dashbird*, Dashbird, 7 Oct. 2020, dashbird.io/knowledge-base/basic-concepts/serverless-advantages-and-use-cases/.

[5] Cardoza, Christina. "10 Use Cases for Serverless." *IT Ops Times*, D2 Emerge LLC., 30 May 2018, www.itopstimes.com/cloud/10-use-cases-for-serverless/.

[6] Maissen, Pascal, et al. "FaaSdom." *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, 2020, doi:10.1145/3401025.3401738.

[7] Jayathilaka, Hiranya, et al. "Response Time Service Level Agreements for Cloud-Hosted Web Applications." *Proceedings of the Sixth ACM Symposium on Cloud Computing - SoCC '15*, Aug. 2015, pp. 315–328., doi:10.1145/2806777.2806842.

[8] Kim, Jeongchul, and Kyungyong Lee. "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service." *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, doi:10.1109/cloud.2019.00091.

[9] Sasidhar Vanga. "Measuring Code Execution Time in Browers." *2018 Medium Blog*. URL: https://medium.com/leantaas-engineering/measuring-code-execution-time-in-browsers-9595fd2776bd

[10] Google Cloud Platform. "Functions Framework for Node.js." *2010 Github*. URL: https://github.com/GoogleCloudPlatform/functions-framework-nodejs

[11] Ciro S. Costa. "Measuring HTTP response times with cURL." *2018 Blog Post*. URL: https://ops.tips/gists/measuring-http-response-times-curl/

[12] Google Cloud. "Cloud Functions." *2010*. URL: https://cloud.google.com/functions/

[13] MDN Web Docs. "Console Object." *Mozilla*. URL: https://developer.mozilla.org/en-US/docs/Web/API/console

[14] MDN Web Docs. "Using Fetch." *Mozilla*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch