Microservice Performance Analysis and Optimization Strategies in Serverless Computing

Yu-Cheng Chen, Michael Enriquez, Sai R.S.M Grandhi, Vishesh Sharma
Department of Computer Science and Engineering
Santa Clara University
Santa Clara, USA

***Abstract*—As the pioneers of technological advancement move forward, many of them choose to do so through the many avenues of cloud computing. Serverless computing provides many of the benefits that innovators are seeking such as being efficient, elastic, and light weight. As an event-based model many users choose Function-as-a-Service because they pay for only the resources that they use with no over provisioning. Despite the many benefits serverless cloud computing provides there is always room for improvement. In this paper we aim to analyze the performance of different microservices on the popular serverless computing platform AWS Lambda. On this platform we intend to examine how well serverless functions perform when running certain benchmarks that focus on different computational areas such as elasticity, memory, load balancing, and compiled vs interpreted programming languages. All of these metrics will be tested on both warm, rapidly repeated usage, and cold, stagnant infrequent usage of the containers running the code. Our goal is to demonstrate that the variance of these factors can be manipulated in ways to improve performance and reduce latency, as well as provide cost reduction strategies without sacrificing time or resources.**

***Index Terms*—Cloud computing, Function-as-a-Service, Serverless Cloud Computing, Amazon Web Services**

I. Introduction

Cloud Computing has ushered in several new approaches to the field of web infrastructure. Dependence upon physical components has rapidly reduced beacuse the combination of Virtualization, Utility computing gave rise to service models such as Software-as-a-Service (SaaS), Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS). As cloud computing continues to evolve, more and more features are being explored. One such feature is the use of serverless platforms which provide function-as-a-service (FaaS). This service allows users to take advantage of the benefits cloud computing has to offer without the complicated overheads, and without the over-provisioning of Infrastructure-as-a-Service which operates at a Virtual Machine (VM) level. This abstraction allows FaaS to provide savings in space and cost of cloud resources.

Because the usage of FaaS executes callable functions on a per runtime basis, there is no permanently running server ready to execute these functions instantly. Although the time to execute is generally considered to be low enough, there is a phenomenon dubbed as a ”cold start” which describes the increased latency of a executing a function which has not been called recently. Lowering this latency can be important for several reasons such as accurate real time execution, increased throughput, and consistency. Our project aims to analyze several different factors that influence the performance of different microservices in serverless computing platforms in the hopes of using this information to construct optimization strategies for different scenarios as well as determine which optimizations are cost effective to use.

II. Problem Analysis

A. Platform

Amazon Web Services: Amazon Web Services (AWS) provides serverless computing through its Amazon Lambda product. As AWS describes it, Lambda provides pay as you go computing without the need for complicated administration that can scale as necessary to meet demand without ever over provisioning, the basic use case can be seen in Fig 11. Lambda supports Node.js, Python, Java and C#. Additionally, monitoring is provided through Amazon Cloudwatch and handles scaling automatically freeing the user to focus on designing their products. Amazon prices Lambda based on total requests and/or total duration with one million invocations and 400,000 GB-seconds per month free, and charging $0.20 per additional million requests, and $0.00001667 for every additional GB second (price is dependent on memory allocation).

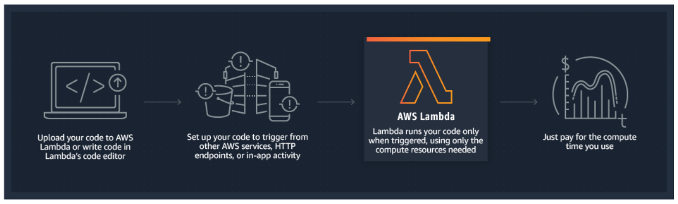

Upload your code to AWS Lambda or write code in Lambda's code editor
Set up your code to trigger from other AWS services, HTTP endpoints, or in-app activity
AWS Lambda
Lambda runs your code only when triggered, using only the compute resources needed
Just pay for the compute time you use

Fig. 1. Guide to the general use of AWS Lambda

B. Factors Considered

Elasticity: When cold requests are sent to microservice platform, there is a need for provisioning on the host platform before requests can be responded to and due to this the latency varies in different server states. Two types of server states would be considered in this project,

1) container cold, when the first request is sent to the container hosting the microservice code, and

2) warm when repeated request sent to a container already exist.

We are interested in the respond time difference caused by container provisioning.

Load Balancing: Cloud providers usually distribute requests across containers based on CPU load to handle heavy traffic. Incoming microservice requests could influence load balancing and hence the performance. We want to analyze the affects done by quantity of requests serviced by individual containers and hosts upon service execution time. In this paper we would like to address the following questions:

1) How does load balancing vary for hosting microservices in serverless computing?
2) How do computational requirements of service requests impact load balancing, and ultimately microservice performance?

Memory Reservations Since infrastructure is shared by other containers, serverless computing platforms allow users to set memory reservations. While Amazon Lambda users are allowed reserve 128 MB to 3008 MB, Azure function apps do not require memory reservations but can charge a user for every 128 MB increments. These reservations are then applied to Docker containers which may share the resource of a machine and would impact the performance. Using different reservation configurations, we hope to see how cold start and warm start invocations affect run time of the microservice code.

Programming Languages AWS Lambda and other Serverless platforms allow developers to write functions in several different languages. While JavaScript (JS) and Python have been offered since the initial offering of AWS Lambda, other heavy-duty languages such as Java, and C# soon became available. Programming languages like C# and Java tend to have larger build bundle sizes than programs written in Python or JavaScript. It should also be noted that Python, JS are interpreted languages and Java/C# re compiled languages. Moreover, since AWS Lambda uses Node.js runtime to run JS functions, the event-driven feature of Node.js library could influence the initialization time of the function. By running experiments in different languages, it is possible gather insight that would allow developers to make the correct programming language choice for their functions. Our hypothesis is that since compiled languages offered by AWS Lambda, like Java, C# have large bundle sizes, they would take a longer time during cold start compared to functions written in JS or Python.

III. Experimental Setup

A. Elasticity

The elasticity experiment was programmed in Python 3.6 running on AWS lambda with the lowest memory allotment of 128 MB. The actual function being run was comprised of a few different sections. One key focus of this experiment relies on telling whether or not the current instance being run is a warm or cold invocation, so a method was developed to determine the state of the container being run because this information is not directly told to you by AWS.

The first section of the function checks for the existence of a temporary file and creates the file if it was not detected. AWS Lambda containers allow for write access to the file system in the /tmp directory, so the function first checks to see if a specific file we named exists in that directory. If the file exists we know that the container invoking the function at this moment has been used before to run this code and is therefore a warm invocation. In contrast, if the file is not detected we know that that container is running our function instance for the first time, and will create a uniquely named temporary file for following invocations to look for. The function outputs a response code of 200 for a warm invocation and 201 for cold.

In order for there to be noticeable execution time differences, the function does a series of mathematical operations in order to induce computational stress upon the system. This was done by creating an array of 10,000 randomized operands, which would then be indexed randomly to select what type of mathematical operation to do. The options in question were the basic algebraic operations add, subtract, multiply, and divide. This process was then repeated 200,000 times. On average this process took just under 14 seconds to complete on the fasted recorded instance of the Lambda invocation of the code.

To actually test the differences in execution time and general latency, we leveraged the AWS API Gateway in order to create a persistent endpoint link with which we could use to trigger the Lambda function via web connections. This was necessary because in order to test varying levels of concurrency we leveraged a project we found on Github called "Hey."

Hey is a small program written in the Go programming language that runs a provided number of requests in the provided concurrency level and prints statistical information after execution. By running Hey on our AWS API Gateway link with different concurrency values we were able to progressively scale multiple degrees of concurrency.

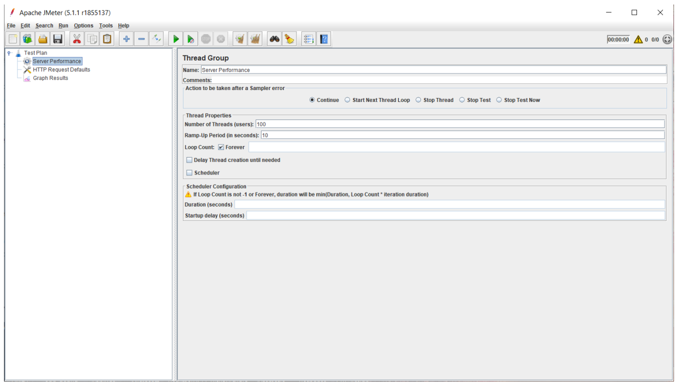
The last part of the experiment necessary for testing all of our cases was guaranteeing whether or not a container was cold or warm. Because the time for a container to become cold after being warm previously is not a set time one solution would be to wait in excess of the possible time to cool, however this is inefficient. It turns out that changing one of the higher level environmental variables such as memory allotment is enough to force a container to depreciate and guarantee a cold container. So simply changing the memory from 128 MB to 256 MB and back immediately was enough to guarantee that all of the next tests would start with cold containers for whatever degree of concurrency that was specified.

B. Load Balancing

In this experimental setup, we used a simple 'Hello World' Ruby (version 2.5) program to execute 6 sets of 100 warm concurrent requests on AWS Lambda function by attaching the Application Load Balancer to it and connected a listener to the Application Load Balancer for checking for the connection requests. Service requests were hosted using 128 MB containers on AWS Lambda with varying ramp-up intervals between the concurrent requests.

We used JMeter [fig 2] for connecting with the HTTP server of the corresponding AWS Lambda function, which was used to execute the concurrent requests and also for varying the ramp-up period (time taken for the whole set of concurrent requests to get executed completely in that given time frame) from 10 to 180 seconds for each set of 100 concurrent requests.

AWS X-Ray was configured and attached to the 'Hello World' Ruby lambda function to monitor the traces of each execution, which was used to calculate the average response time for different configurations of a set of concurrent requests.



Thread Group

Name: Server Performance

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test Stop Test Now

Thread Properties

Number of Threads (users): 100

Ramp-Up Period (in seconds): 10

Loop Count: [✓ Forever]

Delay Thread creation until needed

Scheduler

Scheduler Configuration

If Loop Count is not -1 or Forever, duration will be min(Duration, Loop Count * iteration duration)

Duration (seconds)

Startup delay (seconds)

Fig. 2. JMeter Setup

C. Memory Reservations

By measuring the effects done by different memory configurations, we intended to show the cold start performance that could be influenced by memory reservations. In this experiment we compared the COLD start time of several AWS Lambda functions of both Python 3.7 and Java 8 languages, each function with 128MB, 256MB, 512MB, 1024MB, 2048MB and 3008MB memory allotted in order to observe the relation between memory reservations and COLD start times, and the set of Java functions served as a comparison so we could confirm that the same relation holds for different language AWS Lambda functions.

In terms of the setups, we used standard hello-world script for all the functions, and AWS CloudWatch Events were configured to trigger all function every two hours with a cron job using the expression: $cron(00/2**?*)$; and also AWS X-Ray was used to record traces of every function execution, the data collected by AWS X-Ray contains AWS::Lambda service's start time, end time and function's time stamp of initialization, invocation and overhead etc. which allowed us to calculate the COLD start time of each execution.

We then left the functions to run for four days to collect data, and we used a Python script which utilized AWS CLI (Command Line Interface) along with AWS X-Ray API to extract required information and then calculate the COLD start time by $(AWS :: Lambda :: Functionstart_time - AWS :: Lambdastart_time)$, we could finally get the aggregate result from average COLD start time for each memory configuration.

We expected that memory reservations and COLD start times would have a linear relationship, if the time AWS Lambda service spends when configuring the container environment and initializing the server-less function is related to the memory capacity allotted.

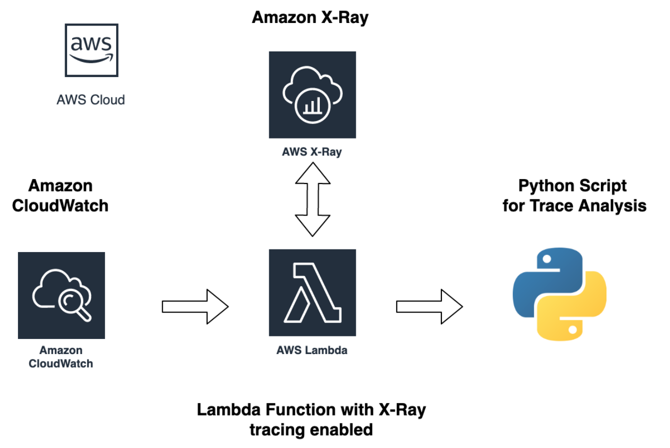

AWS Cloud
Amazon X-Ray
AWS X-Ray
Amazon CloudWatch
Amazon CloudWatch
AWS Lambda
Python Script for Trace Analysis
Lambda Function with X-Ray tracing enabled

Fig. 3. AWS Lambda infrastructure for Memory Reservations and Programming Languages Experiment

D. Programming Languages

To test the effects of programming language choice upon cold start times, we chose Java, Python and JavaScript programming languages. The reason for these choices is that, while JS and Python are interpreted programming languages, Java is a compiled language. As previously mentioned Moreover, several enterprise applications these days are written in Java, Python and with the rise of Node.js, JavaScript is also being used extensively at places like Capital One, Netflix etc.

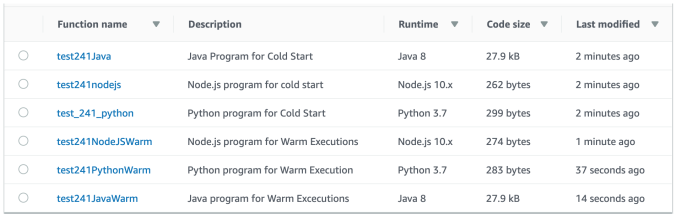
In terms of the configuration on AWS Lambda, we used a standard hello-world script for all the functions implemented in different languages. This is chosen to maintain consistency across our experiments. AWS has several other products that assist in logging metrics about the different AWS computational products being used. The services that we are used for this experiment are Amazon CloudWatch and AWS X-Ray. We used Amazon CloudWatch to setup events that could be triggered at certain times and store metrics related to duration of each execution. Each execution is traced from its invocation till complete execution using AWS X-Ray service. In total, we setup our experiment's infrastructure six lambda functions: three cold functions, and three warm functions in Python3.6, Node.js 10x, and Java 8 programming languages.

During the initial days of AWS Lambda, it was recorded that the container hosting the serverless function would go idle after being inactive for more than five to ten minutes. However, AWS has changed that recently to at least 30 minutes. Hence, we decided to run our cold start functions every forty minutes. To carry out this execution automatically, a CloudWatch event was attached to each of the functions with a cron job using the expression: $cron(0/40 * **?*)$. The cold start functions were executed in this manner for two days. Another CloudWatch event was attached to the warm executions experiment. The warm start tests were run for an hour every two minutes using the cron job: $cron(0/222-23?***)$, where 22-23 represented the time of the day (an hour) when the tests ran for the three functions.

To get aggregated results, we needed a way to collect all the results and find the average time taken to setup and initialize the program. This was done by first enabling AWS X-Ray in the Debugging and error handling dialog of AWS Lambda configuration page. All traces are stored in JSON file that is later parsed to retrieve appropriate results. Similar to the memory reservations calculations, for each trace we calculated the time taken for the AWS Function start time against the AWS Lambda start time:

$(AWS :: Lambda :: Functionstart_time - AWS :: Lambdasta$

Our hypothesis is that since Java build size is large, as shown in Fig 4., the COLD start times will be larger compared to Python or JavaScript.



	Function name	Description	Runtime	Code size	Last modified
○	test241Java	Java Program for Cold Start	Java 8	27.9 kB	2 minutes ago
○	test241nodejs	Node.js program for cold start	Node.js 10.x	262 bytes	2 minutes ago
○	test_241_python	Python program for Cold Start	Python 3.7	299 bytes	2 minutes ago
○	test241NodeJSWarm	Node.js program for Warm Executions	Node.js 10.x	274 bytes	1 minute ago
○	test241PythonWarm	Python program for Warm Execution	Python 3.7	283 bytes	37 seconds ago
○	test241JavaWarm	Java program for Warm Excecutions	Java 8	27.9 kB	14 seconds ago

Fig. 4. Lambda functions, 3 for COLD requests, 3 for WARM requests, in Python, Java, Node.js

IV. Experiment Results

A. Elasticity

After conducting our experiment on the elasticity of serverless AWS Lambda there were several observations that we took note of. The warm invocation concurrency test results are apparent in figure 6 while the results of concurrent cold tests can be found in figure 5. The graph for cold concurrency tests showcases the amount of additional latency that was observed when increasing the concurrency when compared to a baseline value of a concurrency of one which was considered zero in the Y axis of our graph. As evident in figure 5, there is a steady increase in invocation latency that rises roughly in parallel compared to the level of increased concurrency up to an addition of nearly 2 full seconds. Two seconds is a non-insignificant delay that could have somewhat severe or unacceptable complications for real-time systems. In contrast, figure 6 showcases the total time of execution for our elasticity test function at the varying levels of concurrency. This figure showcases nearly the opposite of the cold start scenario where execution time tended to either stagnate or diminish with an increase in concurrency. This highlights the importance of keeping a function warm when using Lambda or any serverless framework if there is any sort of time critical element to the pipeline of a given application. Even the slowest invocation of the function in question during a warm start scenario was faster than the fastest cold start invocation.

B. Load Balancing

Each stress test reflects the amount of CPU time and average response time with respect to the ramp-up period. After conducting the stress test, the average response time with respect to the ramp-up period was recorded in the graph [fig 7].

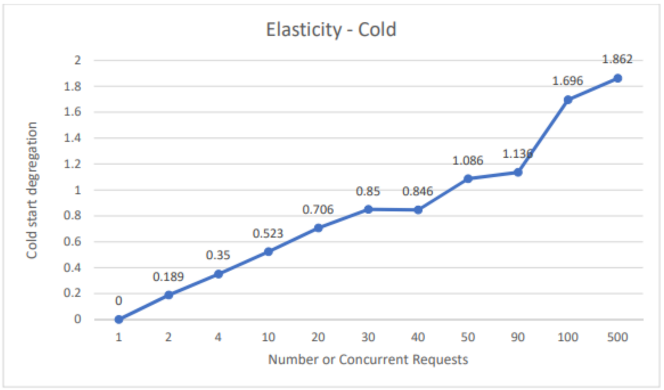


Fig. 5. Function execution time differences as a result of increasing concurrency levels in cold containers.

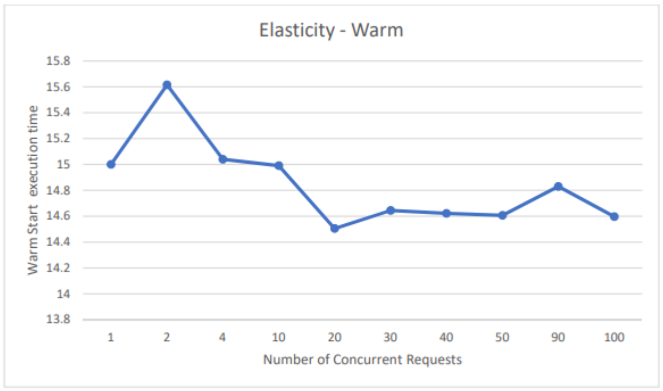


Fig. 6. Total execution time of warm functions invoked at varying levels of concurrency

We notice that when the ramp-up period is 30 seconds the average response time for 100 concurrent is 54.5 seconds, and the average response time steadily decreases with the increase in the ramp-up period, to 52.5 seconds for the 100 concurrent requests when the ramp-up period is increased to 180 seconds.

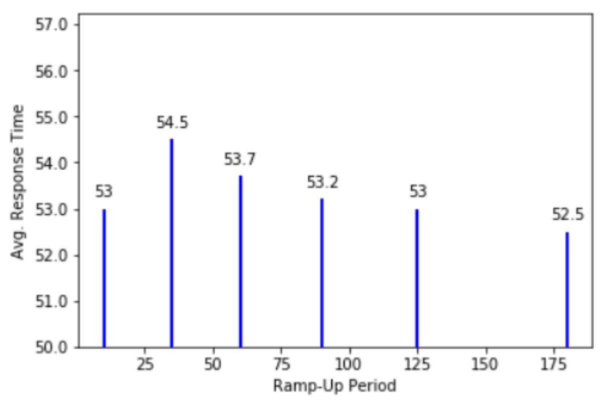


Fig. 7. Average response time with respect to varying ramp-up time for each concurrent requests.

As indicated from the graph, there is a lower average response time with increase in the ramp-up period, therefore as the time interval between the concurrent requests increases, the average response time for the CPU decreases, this implies that the CPU can service each job more rapidly with the increase in the time interval between the service requests as the load balancer services only a given set of requests in a given time interval thereby increasing the wait time for other requests which are in the queue.

C. *Memory Reservations*

There are some noticeable things we observed from the figure 8 and figure 9, the results show that increasing of the memory allocated to the function tends to decrease the COLD start time, and there are 10 to 17 ms decrease of COLD start time per GB of memory reservation depending on different languages, and Java has an overall higher COLD start time comparing to Python.

Although both graphs show the similar results, they are not very intuitive, before the experiments we expected an increasing COLD start time instead of decreasing, however, because AWS Lambda functions' CPU computing power is provisioned according to the memory capacity allocated to the functions, users are not able to modify it during the configuration, the difference of CPU capacity could be the reason that makes COLD start time decreases with a higher memory reservation.

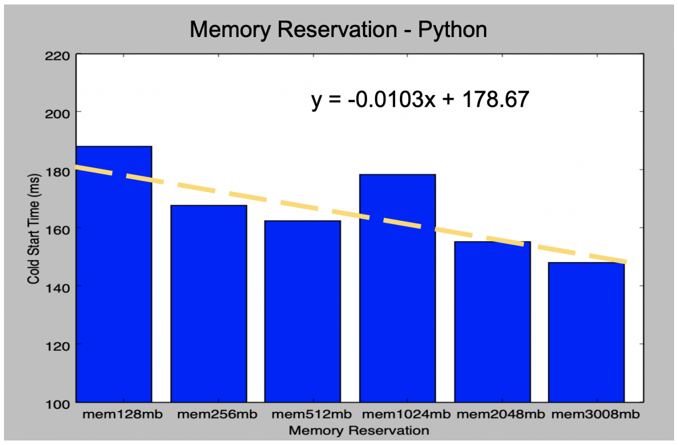


Fig. 8. Python function COLD start time as a result of increasing memory reservations.

D. *Programming Languages*

As expected, COLD starts are usually take longer than warm starts. This can be seen in the X-Ray trace results as shown in Fig. 10 that displays the average of Node.js average run-times. In Fig. 10, we can see that after the $AWS :: Lambda :: Function$ is executed, the warm function, titled $test241NodeJSWarm$, executes in 36ms while the cold function $test241nodejs$ takes 181ms.

Average results, plotted in Fig. 11, show that our hypothesis is proved. Functions written in Java take a long time to start compared to functions written in Python and Node.js. Once again, this could be because of the larger build sizes as all the functions are consistent in terms of functionality. Python3 is the fastest, with average cold start time at 150.43ms, while Node.js trails at 157.65ms and Java8 function averages 338ms. It should also be noted that Java8 functions' cold start times for both experiments is very close, 338ms for cold and 306.75ms for warm functions.

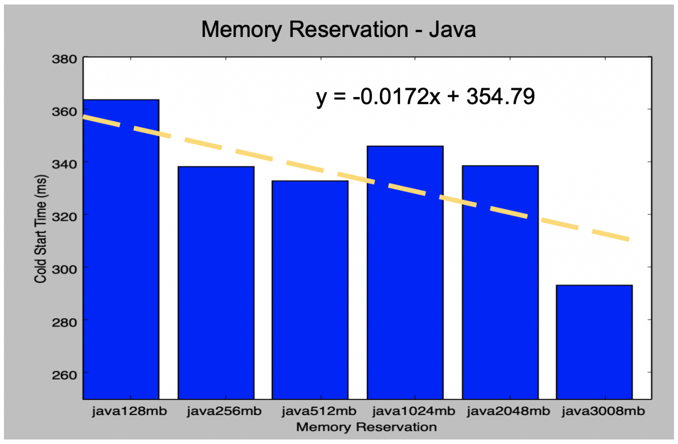


Fig. 9. Java function COLD start time as a result of increasing memory reservations.

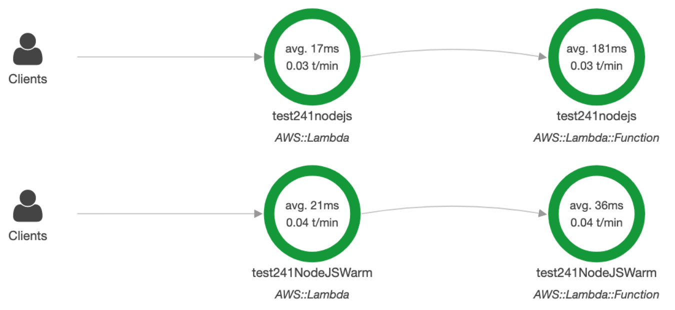


Fig. 10. X-Ray trace results for Node.js

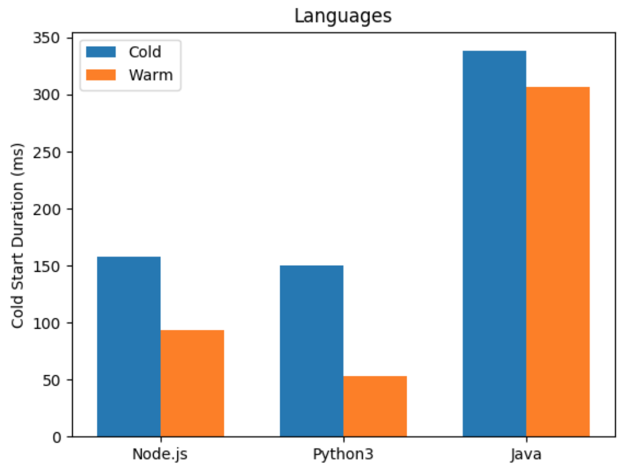


Fig. 11. X-Ray trace results for Node.js

V. Related Work

Previous works include research conducted by Microsoft in association with Colorado State University's Department of

Computer Science. This team evaluated cold start requests to Microsoft Azure and Amazon Lambda function app services based on elasticity, load balancing, provisioning variation, infrastructure retention and memory reservations. The experiments also were included comparisons of the factors by executing functions on a VM that hosts the microservice code.

Another research paper, worked on by researchers from University of Bamberg, Germany, tested cold start invocations upon AWS Lambda and Azure functions. The evaluated their hypothesis on the following factors, programming languages, deployement packet size, memory/CPU setting, number of dependencies, concurrency level, prior executions, and container shutdown.

VI. Conclusion

Our goal throughout this project was to divine some general truths about the varying effects of different environmental specifics on the performance of microservices running on serverless cloud. Through four different experiments we were able to make a few observations which we believe can help guide users in making informed decisions about what environmental settings to prioritize.

Cold start requests can somewhat significantly delay the total time of execution of a function running in serverless and this issue is only further worsened as the number of concurrent requests increases in a cold start scenario. The recommended solution is to somewhat regularly determine the average level of concurrency present in a given application pipeline and ping the lambda function with this level of concurrency as often as necessary to keep the majority of invocations warm. The average response time decreases when there is a greater delay between the execution of WARM service requests in case of Load Balancing, therefore it is recommended to add more load balancers in place when time is considered a crucial factor for rendering service requests. There is a slight benefit to increasing the amount of memory available to a Lambda function so if the cost is worthwhile to obtain the slight reduction in cold start latency then it may be recommended, but this is conditioned upon cost/benefit for the application specified. When using different programming languages we observed that languages that are compiled based (Java) had higher COLD start time as compared to the interpreter based programs (Python, Node.js), therefore it is recommended to use interpreter based programs while expecting faster delivery of execution results.

VII. Acknowledgement

We would like to acknowledge Santa Clara University as well as our professor Abhishek Gupta for offering and teaching the COEN 241 Cloud Computing class respectively.

References

[1] Diwaakar, Lakshman. “Resolving Cold Start in AWS Lambda.” Medium, Medium, 14 Oct. 2017, https://medium.com/@lakshmanLD/resolving-cold-start-in-aws-lambda-804512ca9b61.

[2] Yan Cui. ”How does language, memory and package size affect cold starts of AWS Lambda?”

[3] Manner, Johannes & Endreß, Martin & Heckel, Tobias & Wirtz, Guido. (2018). Cold Start Influencing Factors in Function as a Service. 10.1109/UCC-Companion.2018.00054.

[4] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly and S. Pallickara, ”Serverless Computing: An Investigation of Factors Influencing Microservice Performance,” 2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, 2018, pp. 159-169. doi: 10.1109/IC2E.2018.00039

[5] Maiaroto, Tom. “Azure Cloud Functions vs. AWS Lambda.” Serif & Semaphore, Serif & Semaphore, 4 Apr. 2016, serifandsemaphore.io/azure-cloud-functions-vs-aws-lambda-caf8a90605dd.

[6] https://aws.amazon.com/lambda/

[7] https://en.wikipedia.org/wiki/Serverless_computing