An investigation of the impact of language runtime on the performance and cost of serverless functions

David Jackson

Gary Clynch

An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions

David Jackson, Gary Clynch
Department of Computing
Institute of Technology, Tallaght
Dublin, Ireland
david.jackson@postgrad.it-tallaght.ie
gary.clynch@it-tallaght.ie

***Abstract*—Serverless, otherwise known as "Function-as-a-Service" (FaaS), is a compelling evolution of cloud computing that is highly scalable and event-driven. Serverless applications are composed of multiple independent functions, each of which can be implemented in a range of programming languages. This paper seeks to understand the impact of the choice of language runtime on the performance and subsequent cost of serverless function execution. It presents the design and implementation of a new serverless performance testing framework created to analyse performance and cost metrics for both AWS Lambda and Azure Functions. For optimum performance and cost management of serverless applications, Python is the clear choice on AWS Lambda. C# .NET is the top performer and most economical option for Azure Functions. NodeJS on Azure Functions and .NET Core 2 on AWS should be avoided or at the very least, used carefully in order to avoid their potentially slow and costly start-up times.**

***Keywords*-serverless; FaaS; lambda; aws; azure; functions; performance; cloud**

I. Introduction

Serverless computing is a branch of cloud computing which has evolved from the virtualisation of compute, storage and networking towards increased abstraction of the underlying infrastructure to the point where all that is provided for deployment is the code itself (Hendrickson et al., 2016). A serverless platform manages all aspects of resource management, deployment and scaling transparently. Serverless applications are composed of multiple individual functions, each of which can be implemented in a choice of programming languages, based on the runtimes supported by the serverless platform.

This paper seeks to understand what impact the choice of language runtime has on the performance and subsequent cost of serverless function execution. In serverless billing models, performance and cost are intrinsically linked, based on a "pay only for what you use" model. Given the choice of language runtimes available in each serverless platform, there would be an expectation that some might perform faster than others. This might be also expected to lead to a difference in overall costs to run functions in different languages.

This paper aims to isolate the performance of serverless platforms, in order to understand how long it takes to initialise the internal container environment necessary for execution of a function. It aims to eliminate the performance characteristics of the language itself by measuring completely empty test functions.

This paper presents a series of tests against two major commercial serverless platforms: AWS Lambda and Microsoft Azure Functions. A new test framework, titled "Serverless Performance Framework" (SPF), is introduced in order to collect the necessary metrics for analysis in an automated way across multiple cloud platforms.

AWS Lambda supports a total of five different runtimes (.NET Core, Java, Python, NodeJS and Go), all of which were evaluated in this research. Where there are multiple versions available for a single runtime, the latest version available was chosen. For Azure Functions, the testing was limited to C# and NodeJS for purposes of cross comparison with the same runtimes that are available on AWS.

Serverless platforms generally use a container-based workload management system internally in order to execute individual functions and provide the ability to scale on demand. If possible, a serverless platform will re-use an existing execution container rather than creating a fresh environment to execute a function. This is referred to as "warm-start" and would be expected to result in reduced latencies compared with a "cold-start" scenario. In cold-start, there is no available container for re-use, so a fresh container must be created and initialised with the function code and all required dependencies before the function execution can begin. This paper performs function testing against both these scenarios.

II. Serverless Review

Each time a serverless function is invoked, it is executed internally on a platform-managed server via a container which is (potentially) provisioned in real-time. Fox et al. (2017) describe this approach as being *"server-hidden"*. This section describes recent research into performance and cost considerations under this FaaS approach, as they relate to this paper.

A. Serverless Performance Considerations

Cold-start refers to the time taken to create a fresh container to execute a function and perform any necessary runtime initialisation. Limiting this "cold-start" effect is cited by Varghese and Buyya (2018) as being a key focus for a responsive serverless implementation. This effect is observed by Ishakian et al. (2017) in their study on the suitability of using serverless functions for deep-learning tasks, where they evaluate both "cold" and "warm" start scenarios.

McGrath (2017), investigates serverless latencies through warm and cold start testing. The tests were performed against functions implemented solely in NodeJS. Serverless functions which were essentially empty were deployed to AWS Lambda. They were designed in this way in order to allow measurement of serverless framework performance and internal container re-use, abstracting out any programming language performance itself, although it does include the time taken for a round-trip API call from the test harness. This is a similar approach to the one taken for this paper, although this paper's approach removes any API or network latencies from the results.

B. Serverless Cost Considerations

Serverless functions provide the ability to scale up and down rapidly, to the point where no cost is incurred if the function is not in use (Baldini et al., 2017). This leads to most commercial serverless implementations being based on a fine-grained billing model based on function execution time measured in sub-second intervals (often per 100ms).

Lynn et al. (2017) note the potential cost advantages provided by commercial serverless cloud platforms. However, Baldini et al. (2017) discuss the need to evaluate appropriate workloads for cost-effectiveness on serverless platforms, noting that *"the frequency at which a function is executed will influence how economical it can be"*. For example, Adzic and Chatley (2017) and Villamizar et al. (2016) present cost-saving examples for serverless applications against traditional VM-based hosting in the order of 99.8% and 57%. However, the throughput presented in these examples equate to only 0.003 TPS and 7 TPS respectively. On a similar theme, Eivy (2017) encourages a thorough evaluation of whether a serverless solution will deliver the expected cost benefits in practice. He suggests estimating function usage using TPS (Transactions-per-Second), noting that AWS and Azure free tiers of one million free requests per month only amount to 0.38 TPS. For a clearer comparison, any free tier allocations are not included in this paper's calculations.

The impact of the language choice on performance was cited by McGrath (2017) as an area of future work. Given cost transparency provided by serverless billing models, understanding the impact on cost of a function's language runtime is important and is the subject of this paper.

1) Serverless Cost Modelling: Leitner et al. (2016) present a comprehensive microservice cost modelling framework, named "CostHat", which aims to provide cost information to developers in real-time as they make changes. CostHat is a useful model to apply to a serverless architecture, given serverless functions usually conform to the common microservice characteristic of providing a single clearly-defined capability or function (Fowler and Lewis, 2014). The power of the CostHat[1] model is in its recursive nature, calculating costs based on downstream service dependencies. The formula is based on various costs associated with function execution including compute, API and I/O. This paper presents a CostHat model

[1]https://github.com/xLeitix/costhat

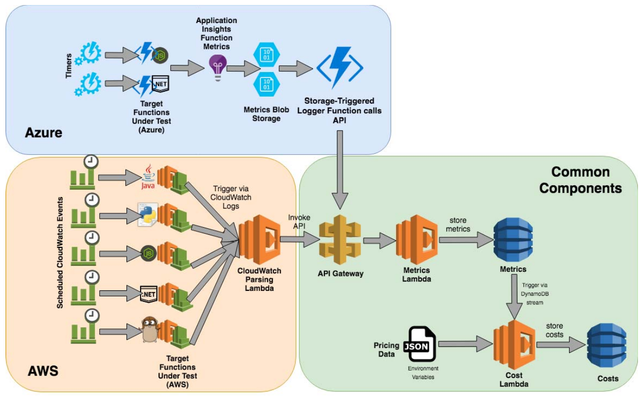
Azure
Timers
Target Functions Under Test (Azure)
Application Insights Function Metrics
Metrics Blob Storage
Storage-Triggered Logger Function calls API
AWS
Scheduled CloudWatch Events
Java
.NET
Target Functions Under Test (AWS)
Trigger via CloudWatch Logs
CloudWatch Parsing Lambda
Invoke API
Common Components
API Gateway
Metrics Lambda
store metrics
Metrics
Trigger via DynamoDB stream
Pricing Data
JSON
Environment Variables
Cost Lambda
store costs
Costs

Figure 1: Architecture of Serverless Performance Framework

in order to apply performance and cost test results against a realistic serverless architecture.

III. Serverless Performance Testing

The following five runtimes were measured on AWS Lambda: NodeJS, Go, Python, Java and .NET Core 2. These were chosen as the five available languages on Lambda. Two additional runtimes would be measured on Azure Functions platform: NodeJS and .NET C#. These were chosen from the available Azure Functions runtimes to enable cross comparison with their equivalents on AWS Lambda.

A series of cold-start and warm-start tests were designed. Previous research by McGrath (2017) and Ishakian et al. (2017) has shown that the actual lifetime of a serverless execution container is indeterminate, however their research showed that a cold-start interval of 1-hour would be sufficient to guarantee execution in a new container. The "Warm Start" interval rate was set to be 1-minute intervals. The tests were designed around empty test functions to measure the time taken to setup the function's execution environment. All tests were performed in batches as described in sections IV and V. These batches were executed in early 2018.

A. Serverless Performance Framework

A new test framework was created to enable consistent, automated metrics gathering for this research. In addition it has the purpose of eliminating any external latencies (such as API invocation) from test results. This framework is dubbed the "Serverless Performance Framework" (SPF). The components were created via the popular open-source serverless development tool called "Serverless Framework"[2]. This simplifies serverless function development and deployment across a number of different serverless platforms. The technical implementation details of the solution are described in Figure 1. Full source code is available on GitHub[3].

[2]https://serverless.com/framework/

[3]https://github.com/Learnspree/Serverless-Language-Performance-Framework

1) Common Components: To enable recording of data from any serverless platform, all functionality for recording, calculation and analysis of performance and cost metrics is exposed through a standard API. Currently, there is just a single POST operation in the API, which triggers the "Metrics Lambda" function and performs the task of storing the provided metrics and any derived data into persistent storage (DynamoDB table). This component is implemented as a .NET Core 2 AWS Lambda function.

The "Cost Lambda" function is triggered via an event when the main performance metrics are persisted to DynamoDB (labelled "Metrics" table in Figure 1). It takes those values, combines with the latest pricing data and calculates the estimated cost of execution of that function. The pricing data is stored to DynamoDB "Costs" table. This component is implemented as a NodeJS (6.10) function.

2) AWS Components: Test components that were specific to testing AWS Lambda functions were required which ultimately connect with the common SPF components via the API Gateway.

The AWS test functions to be measured were implemented as completely empty functions. The purpose of this was to eliminate from the testing anything other than the performance of the serverless platform in creating the environment for code execution. Note that AWS Lambda supports two flavours of Python (2.x and 3.x) and NodeJS (4.x and 6.x). In these cases, the latest framework versions (Python 3.6 and NodeJS 6.10) were measured.

By default, AWS Lambda sends three entries to CloudWatch Logs for every lambda function execution. This includes an execution "REPORT", which contains all the required metrics such as execution duration, memory and function name. The AWS Logger component of the SPF performs the task of parsing this CloudWatch entry and translating these values into a call to the SPF API. This logger component is implemented as a NodeJS (6.10) AWS Lambda function.

3) Azure Components: Like the AWS components, Azure components also store metrics via the same SPF API. The test functions were implemented in NodeJS 6.10 and .NET C#. Test functions were configured to integrate with Azure Application Insights, which collects rich logging and telemetry data.

Unlike for its AWS equivalent, CloudWatch, it is not possible to trigger the logger function from Application Insights directly. Instead, Application Insights was configured for continuous export to Azure Storage. Azure Functions can be triggered from the insertion of data into a specified Azure Storage container. This ability was used to allow the logger function to send performance data for processing via the SPF API. The logger function in Azure performs a very similar function to its equivalent in AWS Lambda, parsing the required metrics to send to the API.

IV. AWS Test Results

This section presents the tests results for AWS Lambda. Tests were performed on each of the five language runtimes available in AWS: NodeJS, Python, Go, Java and .NET C#. All tests were performed on empty functions in order to measure purely the serverless platform's performance in creating and running the environment to execute the function.

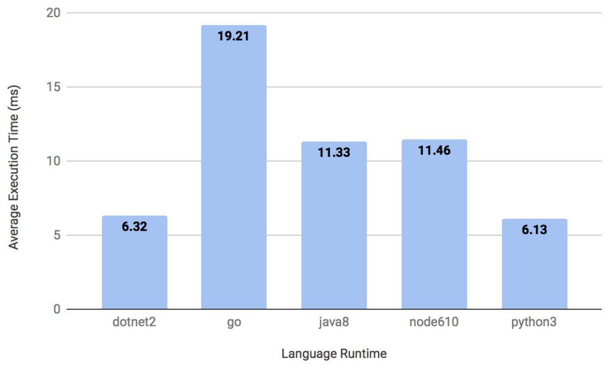

Figure 2: Average of Execution Times (ms) across all Warm Start Testing in AWS Lambda

A. Warm Start Tests

In total, four separate 1-hour warm-start tests were executed on AWS Lambda. In total, 248 individual tests were executed against *each* of the five language runtimes. The average execution time for a *completely* empty function in each runtime is presented in Figure 2.

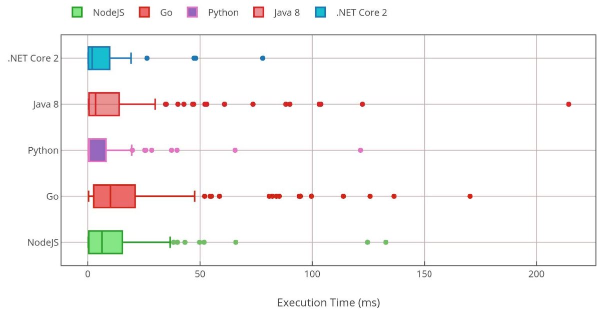

Figure 3: Box Plot of AWS Lambda Warm Start Tests (ms)

The results show, on average, that Python is just ahead of its nearest rival at an average execution time of 6.13ms. The surprise is the appearance of .NET Core 2 as a close second. This was quite unexpected, as the JIT (Just-in-Time) nature of the compiler would be expected to result in slower performance (Hendrickson et al., 2016). It outperforms even the dynamically typed NodeJS, as does Java. C# .NET and Java applications, executed at runtime via the .NET CLR (Common Language Runtime) and JVM (Java Virtual Machine), were expected to take longer to initialise. The laggard in the first warm-start function test is Go, at an average runtime performance of more than 300% of Python and .NET Core 2. Go, although a statically typed language, has certain features (such as native binary compilation) that suggested faster performance than it showed.

A comparative box-plot of the warm start results is displayed in Figure 3. What this diagram helps illustrate is Golang's poor

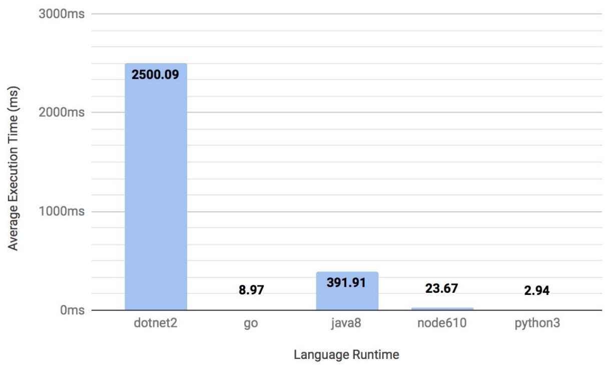

Figure 4: Average of Execution Times (ms) across all Cold Start Testing in AWS Lambda

average execution time. Compared to the other four runtimes, it's execution times are far more variable and distributed, shown by the relatively large fourth quartile, denoting that the longest 25% of all tests were relatively widely distributed (from 21.09ms up to 47.61ms). The box-plot also illustrates the consistency in execution of the top performers, .NET Core 2 and Python.

B. *Cold Start Tests*

A full cold-start test was run over a longer period than the warm-start testing, consisting of a total 144 hours (6 days). This involved 144 individual invocations of *each* runtime's empty test function. This was done in order to measure cold-start performance across different overall environment conditions which may occur at different days of the week or times of the day.

The results showed some interesting contrasts to the warm-start tests. Figure 4 shows the average cold-start execution time of the five language runtimes. .NET shows the largest increase (a massive 39,558%) in cold start time vs. warm-start scenarios. Java also shows a significant (although not quite as dramatic) relative increase of 3,459%. The other runtimes show more consistent performance between cold and warm-start. What is difficult to understand is the clearly better performance of Go and Python in the cold-start tests against warm-start. They perform over 50% slower in *warm-start* scenarios. This is counter-intuitive to the expected pattern and requires future investigation.

.NET Core 2 showed unexpectedly strong performance in the initial warm-start test. This makes its dramatically slow performance in this cold-start scenario surprising. Average empty function duration has increased from 6.32ms to 2500.09ms. This provides some interesting guidelines in the suitability of .NET Core as a language of choice for AWS Lambda. The most obvious conclusion is that, if possible, .NET should only be used in functions that are frequently accessed and are less prone to significant scale-out events.

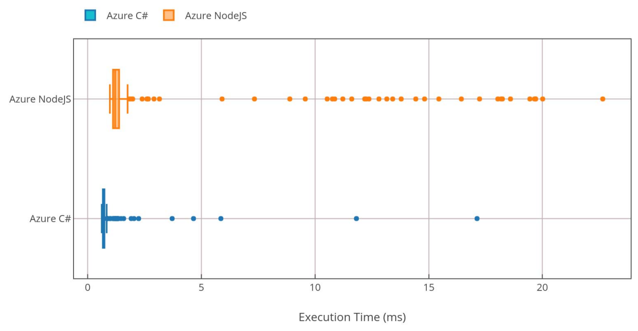

Figure 5: Box Plot of Azure Functions Warm Start Performance (ms)

V. Azure Test Results

This section presents the tests performed on the Microsoft Azure Functions serverless platform and describes the results produced. Tests were against two language runtimes: C# .NET and NodeJS. Azure Functions assigns memory to functions dynamically and not in the pre-defined way AWS Lambda is configured. In the process of testing, it was observed (via Azure CLI and Application Insights metrics) that each function executed comfortably consumed less than the 128MB minimum billing threshold for Azure Functions.

A. *Warm Start Tests*

Warm-Start tests were performed at 1-minute intervals. There were a total of 273 Warm-Start tests over a total 4.5-hours for both Azure runtimes tested. As can be seen from the box-plot in Figure 5, C# performs consistently faster of the two runtimes tested in Azure, showing an *average* sub-millisecond performance of just 0.93ms. This compares to 4.91ms for NodeJS.

B. *Cold Start Tests*

There were a total of 144 Cold-Start tests (for both runtimes) over a 6-day period. These were performed at the same 1-hour intervals as in AWS Lambda testing.

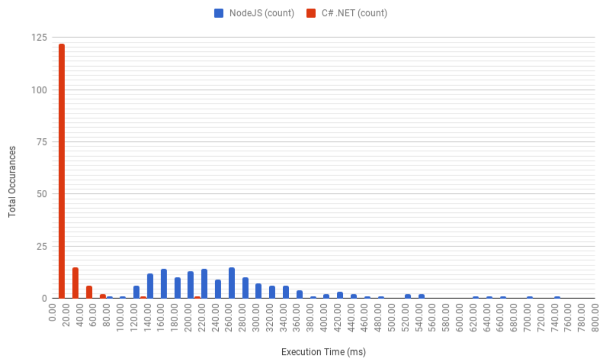

Figure 6: Histogram of Azure Functions Cold Start Performance (ms)

In the Cold-Start scenario, C# performs significantly better than NodeJS - 16.84ms average compared to 276.42ms. The histogram displayed in Figure 6 provides details on this disparity. C# shows considerable consistency, with the vast majority of tests falling in the 0-20ms bucket. The NodeJS runtime shows significant variability in execution time, with a relatively even "bell curve" across the 20ms buckets between 120ms and 340ms.

VI. Comparison of AWS Lambda and Azure Functions

This section describes a comparison between Azure Functions and AWS Lambda based on the two runtimes tested in Azure: NodeJS and .NET C#.

A summary of the average execution times in each scenario is shown in Table I. It shows that each serverless platform has advantages. For NodeJS, AWS Lambda shows a significant advantage in terms of cold-start performance (23.67ms vs. 276.42ms average). The box-plot shown in Figure 7 adds more detail as to the spread of test results in this cold-start scenario for NodeJS. Azure is clearly much more optimised for C# support than NodeJS. Perhaps this is related to the internal containers on Azure which are currently based on windows container technology versus the linux-based containers in use on AWS. However, this would need specific further study.

For C# .NET performance, the situation is reversed. Table I shows that Azure Functions significantly out-perform AWS Lambda in warm-start and particularly in cold-start scenarios. In a warm-start, AWS performance is reasonable at an average of just 6.32ms per execution (compared to 0.93ms on Azure) and both platforms are equivalent from a cost perspective. However, the particularly poor cold-start performance of AWS Lambda (average 2500.09ms) compares badly with Azure in both performance and cost.

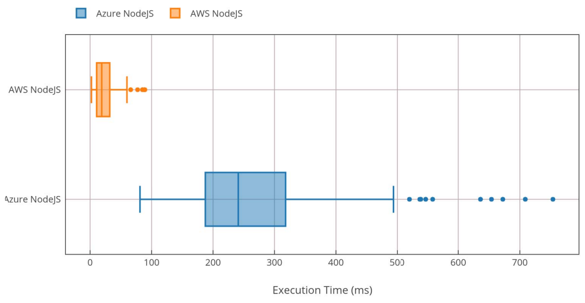

Figure 7: Box Plot of Cold Start Performance (ms) for NodeJS on Azure and AWS

For warm-start tests in C#, the box plot in Figure 8 shows a comparison of performance between Azure and AWS. Azure significantly (but not unexpectedly) out-performs AWS. Given .NET is a core technology for Microsoft, Azure Functions would be expected to have solid support for C#. Also, it is worth noting that .NET Azure Functions are implemented as

Serverless Platform	Language Runtime	Warm Start Average (ms)	Cold Start Average (ms)
AWS	.NET C#	6.32	2500.09
AWS	NodeJS	11.46	23.67
Azure	.NET C#	0.93	16.84
Azure	NodeJS	4.91	276.42

Table I: Summary of Average Performance Between Azure and AWS

c-sharp "script" (.CSX extension) files running on windows containers. This is different to AWS which uses the open-source .NET CLR (Common Language Runtime) on linux containers.

VII. Cost Analysis

The cost of serverless functions are directly related to their execution times. This is due to the prevalent billing model across the major serverless platforms of cost per milliseconds of execution. Both Azure and AWS bill in 100ms blocks.

There are three main factors in a function's execution cost - execution time, fixed invocation cost per individual function execution and memory allocated to the function. Both platforms provide similar "free-tier" allocations of 1 million executions per month and 400k GB/s of execution time[4][5]. For the purposes of a consistent comparison across platforms, these free-tier allocations are excluded from cost calculations.

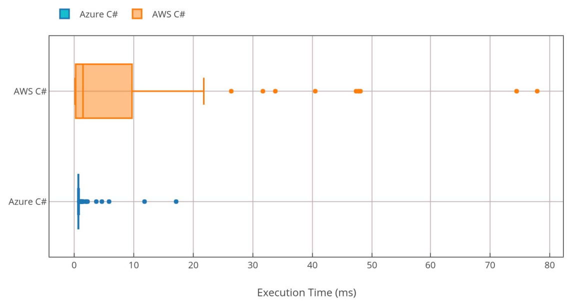

Figure 8: Comparison of Warm-Start C# Function Performance (ms) Between Azure and AWS

A. AWS Lambda

Table II contains the cost calculations based on the performance data from all cold-start tests. Note that warm-start tests were omitted as all runtimes' average execution time were below 20ms, well below the 100ms billing increment meaning all costs were the same at $0.41. Costs shown were calculated using latest AWS Lambda pricing of $0.20 per million function invocations and $0.00001667 per GB/s of execution time, applied to average execution times recorded.

The long initialisation times for Java and, in particular, .NET Core in cold-start have a significant effect on cost. Calculated

[4]https://aws.amazon.com/lambda/pricing
[5]https://azure.microsoft.com/en-us/pricing/details/functions/

Language Runtime	Average Execution Time (ms)	Average Billed Duration (ms)	Average Cost Per Million ($)
C# .NET	2500.09	2600.00	5.61775
Golang	8.97	100.00	0.408375
Java 8	391.91	400.00	1.0335
NodeJS	23.67	100.00	0.408375
Python	2.94	100.00	0.408375

Table II: AWS Lambda Cold-Start Performance Mapped to Cost

Language Runtime	Cost Per Day @ 100-TPS	Cost Per Day @ 30k-TPS	Cost Per Year @ 100-TPS	Cost Per Year @ 30k-TPS
C# .NET	$48.54	$14,561	$17,716	$5,314,840
Golang	$3.53	$1,059	$1,288	$386,355
Java 8	$8.93	$2,679	$3,259	$977,774
NodeJS	$3.53	$1,059	$1,288	$386,355
Python	$3.53	$1,059	$1,288	$386,355

Table III: Cost of Cold-Start execution at Varying Throughput (TPS)

per million requests, the cost for .NET is $5.62 compared with just $0.41 for the top three runtimes (Golang, NodeJS and Python). This is a 1,371% higher cost. For Java, the cost of $1.03 is 251% higher. To put this into context, Table III shows these costs applied to increasing levels of throughput, measured in TPS (Transactions Per Second). While 30k TPS may seem very high for a single function, a realistic system totalling 30k TPS is discussed via the CostHat model in section VII-C.

B. Azure Functions

Unlike AWS Lambda, Azure Functions are not pre-assigned a memory allocation. Instead, they are dynamically assigned memory based on their execution. From a billing perspective, this is measured in 128MB increments based on the *maximum* recorded memory consumed by the function, with a minimum of 128MB. Note that all functions tested in Azure were running within a maximum of 128MB memory.

Costs were calculated using latest Azure pricing of $0.20 per million function invocations and $0.000016 per GB/s of execution time. The data presented in Table IV shows a comparison between the costs of running .NET C# and NodeJS functions based on the performance and cost figures recorded during testing the *cold-start* scenario. The NodeJS runtime has the potential to cost double that of a C# function in Azure. At the extreme high load example of 30k TPS, this could lead to extra annual running costs for a single function of over $378k.

C. Cost Hat Model

The "CostHat" model is a microservice cost-modelling algorithm developed by Leitner et al. (2016). This is a useful model to investigate the costs of a complex set of serverless functions which combined could represent a high throughput system. To demonstrate cost implications of language runtime on a high-volume system (30k TPS), a CostHat model[6] of a slightly modified version of the SPF was created (see Figure 9).

[6]https://github.com/Learnspree/costhat/tree/spf_tests

In this sample architecture, each single call to the Test Controller Function results in a total of 30 function invocations. The numbers in the diagram indicate the number of invocations of each downstream function based on a single call to the top-level "Test Controller" function. This implies that a scalability test running at a rate of 1,000 TPS would result in overall system throughput of 30k total function executions.

The current implementation of the metrics function using .NET Core2 now has hugely significant cost implications. Based on actual performance and cost metrics recorded for this research, the CostHat model reveals an overall running cost at 1,000 TPS of $31,463 per day. Applying the performance of the NodeJS Cost Metrics function to the model, this could reduce to $8,716 per day (a reduction of 72%).

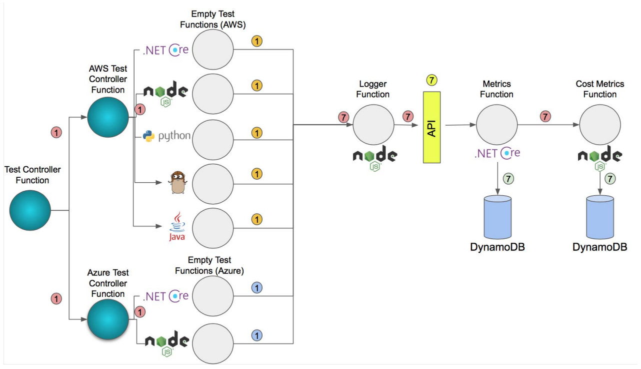

Figure 9: CostHat Model of Modified Serverless Performance Framework

VIII. Conclusion

There were significant differentials between language runtimes on the two serverless platforms tested - AWS Lambda and Azure Functions. For optimum performance and cost-management of serverless applications, Python is the clear choice on AWS Lambda. Similarly, and perhaps unsurprisingly, C# .NET is the clear best choice for Azure Functions, and in fact across both serverless platforms that were measured. The performance of NodeJS in Azure Functions in cold-start scenarios demands caution on its usage on that platform, as with Java and especially C# .NET on AWS Lambda.

Cold-start scenarios expose the cost implications of choosing a poorly performing runtime. Measuring costs per million requests, AWS .NET Core (C#) was shown to cost $5.62 compared with $1.03 for Java and just $0.41 for the other supported AWS Lambda runtimes (Python, NodeJS and Go). The relatively poor performance of NodeJS runtime in Azure Functions in cold-start scenarios also has a significant cost implication. Functions implemented in NodeJS cost, on average, 200% of the C# function cost in the cold-start scenario ($0.80 per million requests compared to $0.40).

One million requests per day equates to a moderate throughput of just over 10-TPS. However, considering an overall

Language Runtime	Cost Per Day @ 100-TPS	Cost Per Day @ 30k-TPS	Cost Per Year @ 100-TPS	Cost Per Year @ 30k-TPS
.NET C#	$3.45	$1,036.80	$1,261	$378,432
NodeJS	$6.91	$2,073.60	$2,523	$756,864

Table IV: Cost of Cold-Start execution at Varying Throughput (TPS) for Azure Functions

enterprise-level eco-system of many serverless functions an overall combined throughput of 30k TPS is realistic and a high rate of cold-start scenarios is possible. An example of such a system was presented via the CostHat model (Leitner et al., 2016). This showed the increased cost caused by a downstream function implemented in .NET was an extra $22,747 per day (361% of the cost if this function was implemented in NodeJS).

Overall, the composition of functions in serverless applications is a crucial design decision, which if done in an appropriately fine-grained manner, can lead to a more flexible but also more cost-effective solution in the long term, as functions can individually be implemented in the appropriate runtime to suit their purpose and expected throughput.

Acknowledgment

The authors would like to thank Technology Ireland ICT Skillnet for their support.

References

Adzic, G. and Chatley, R. (2017), Serverless computing: Economic and architectural impact, *in* 'Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering', ESEC/FSE 2017, ACM, New York, NY, USA, pp. 884–889.

Baldini, I., Castro, P. C., Chang, K. S., Cheng, P., Fink, S. J., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R. M., Slominski, A. and Suter, P. (2017), 'Serverless computing: Current trends and open problems', *CoRR* .
URL: *http://arxiv.org/abs/1706.03178*

Eivy, A. (2017), 'Be wary of the economics of "serverless" cloud computing', *IEEE Cloud Computing* (2), 6–12.

Fowler, M. and Lewis, J. (2014), 'Microservices', http://www.martinfowler.com/articles/microservices.html. Accessed: 2017-12-04.

Fox, G. C., Ishakian, V., Muthusamy, V. and Slominski, A. (2017), 'Status of serverless computing and function-as-a-service (faas) in industry and research', *arXiv preprint arXiv:1708.08028* .

Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H. (2016), 'Serverless computation with openlambda', *Elastic* p. 80.

Ishakian, V., Muthusamy, V. and Slominski, A. (2017), 'Serving deep learning models in a serverless platform', *arXiv preprint arXiv:1710.08460* .

Leitner, P., Cito, J. and Stöckli, E. (2016), Modelling and managing deployment costs of microservice-based cloud applications, *in* 'Proceedings of the 9th International Conference on Utility and Cloud Computing', ACM, pp. 165–174.

Lynn, T., Rosati, P., Lejeune, A. and Emeakaroha, V. (2017), 'A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms.', *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Cloud Computing Technology and Science (CloudCom), 2017 IEEE International Conference on, CLOUDCOM* p. 162.

McGrath, G. (2017), Serverless Computing: Applications, Implementation, and Performance, PhD thesis, University Of Notre Dame.

Varghese, B. and Buyya, R. (2018), 'Next generation cloud computing: New trends and research directions.', *Future Generation Computer Systems* (Part 3), 849 – 861.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A. and Lang, M. (2016), Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures, *in* '2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)', pp. 179–182.