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

Thesis by

David Jackson B.A. (Mod.)

Supervised by:

Gary Clynch B.Sc, M.Sc

In Partial Fulfillment of the Requirements for the
degree of
MSc Applied IT Architecture

INSTITUTE OF TECHNOLOGY, TALLAGHT
Department of Computing

Tallaght, Dublin 24, Ireland

2018
Defended May 21st, 2018

ABSTRACT

Serverless is a compelling evolution of cloud computing based on very discrete, fine-grained functions that are highly scalable and event-driven. Otherwise known as "Function-as-a-Service" (FaaS), this relatively new paradigm represents a move away from stateful VM or container-based computing. It has the potential to radically change application architectures from a cost and design perspective. 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 thesis seeks to understand the impact of the choice of language runtime on the performance and subsequent cost of serverless function execution. It discusses the inherent link between performance and cost for serverless-based applications. The design and implementation of a new serverless performance testing framework is presented, which was created to generate the metrics required for performance and cost analysis. The results generated using this test framework against AWS Lambda and Azure Functions are presented and analysed. These show that there are significant differences in performance between language runtimes within each platform. 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. .NET Core 2 on AWS and NodeJS on Azure Functions should be avoided or at the very least, used carefully in order to avoid their potentially slow and costly start-up times.

ACKNOWLEDGEMENTS

To my wife, without whose incredible support this thesis would not have been possible to complete.

To my children - thank you for reminding me to close the laptop occasionally!

A big thanks to my supervisor, Gary Clynch, who provided fantastic advice and feedback throughout the dissertation.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

Chapter 1

INTRODUCTION

Serverless is a highly scalable cloud-based architectural approach to event-driven applications. A common characteristic of serverless architectures is the logical decomposition of applications into multiple discrete, stateless functions. The serverless platform manages all aspects of resource management, deployment and scaling transparently. Due to the ephemeral nature of serverless functions, cloud platforms supporting serverless are generally required to exhibit low latencies in terms of function startup. Most serverless implementations support function implementations in a variety of languages, summarised in Table 1.1. Between invocations, serverless functions do not use any underlying resources, which means that for consumers there are no servers to manage (Fox et al. 2017). This also leads to the typical serverless billing model of "pay-per-use" where billing is in sub-second chunks and there are no application hosting costs to pay when the application is not in use.

This thesis 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 (see Table 1.1), 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.

Serverless applications generally follow an event-driven "micro-service" architectural style where the application is decomposed into very small individual functions which perform tasks specific to a single domain or capability. While the primary targets of testing performed in this research are individual functions, it is important to understand the wider impact of the test results as applied to a more realistic serverless-based microservices architecture. The costs calculated by this research were applied to a CostHat model (Leitner et al. 2016) of a sample architecture in order to understand the impact of performance testing results on an overall system.

This research aims to specifically target the performance of serverless platforms, in order to understand how long it takes to initialise the internal container-based execution environment necessary for execution of a function. It aims to eliminate the performance characteristics of the language itself by measuring completely empty test functions. Measurements were taken directly from the metrics provided by the serverless platforms themselves, rather from any test harness which is executing the functions. The purpose of this is to eliminate any other factors in the recorded function execution times, including any network latencies that would be involved in API calls. The test functions were triggered by scheduled events, rather than API calls or other direct invocation methods. To achieve these measurements, a series of tests were performed against two major commercial serverless platforms: AWS Lambda and Microsoft Azure Functions. AWS Lambda leads the market, being the first to provide a major commercial serverless platform in 2015[1].

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 research will perform function testing against both scenarios.

AWS Lambda supports a total of five different runtimes (see Table 1.1). As the most popular serverless platform, all of these were evaluated in this research. Where there are multiple versions available for a single runtime, the latest version 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.

A new test framework, titled "Serverless Performance Framework" (SPF), is introduced in order to collect the necessary metrics for analysis. This test framework is created and deployed using the existing popular open-source development tool known as "Serverless Framework[2]". The new SPF framework will standardise the different units of measurement provided by each serverless platform into milliseconds (ms) of execution and megabytes (MB) of allocated memory. Costs are all measured in US Dollars ($) and calculated per-million function executions to aid clarity, as unit costs for a function are in the order of 10^{-5} ($0.00001) per execution.

[1]https://docs.aws.amazon.com/lambda/latest/dg/history.html
[2]https://serverless.com/framework/

Serverless Platform Name	**Supported Languages**
AWS Lambda	NodeJS, Java, Go .NET Core (C#), Python
Azure Functions	NodeJS .NET Core(C#, F#)
IBM OpenWhisk	NodeJS, Java Python, Swift
Google Cloud Functions	NodeJS

Table 1.1: Major Serverless Platforms - Supported Language Runtimes

1.1 Motivation

Serverless is still a relatively new cloud computing approach that has only seriously been commercially available since 2015. That said, there has recently been some excellent research into serverless, and specifically performance. One such example is the research of McGrath & Brenner (2017), who outline the impact of the language choice as an area of future work. Given the attraction of transparency of costs to businesses of cloud computing and the inherent transparency provided by serverless pay-per-use billing models, the understanding of the impacts of language choices on serverless performance (and therefore costs) is particularly important.

1.2 Contribution

The overall goal of the proposed research is to demonstrate the relative performance and cost implications of using different implementation languages for serverless functions. This research is conducted on two major serverless platforms, being AWS Lambda and Azure Functions. Testing was performed in both cold-start and warm-start scenarios, targetting a total of seven different language runtimes across the two platforms. Tests were executed using a new serverless performance framework created for this research. The output of this testing is then analysed and discussed.

The remainder of this thesis is structured as follows. In Chapter 2, a review of current research into serverless is conducted, with particular focus on issues relating to the performance attributes and cost implications of serverless, with a view to setting a foundation for research into the effect of the language runtime on those characteristics. In Chapter 3, the performance testing approach is described and a new test framework, named "Serverless Performance Framework" (SPF), is introduced to produce the required data for analysis. Chapter 4 discusses the results of testing performed against AWS Lambda, in terms of both performance and cost. The CostHat model (Leitner et al. 2016) is applied to this output to produce a cost model of a realistic serverless architecture. In Chapter 5, a similar analysis is performed on results for the two language runtimes tested on Azure Functions. Finally, Chapter 6 discusses the overall results and their implications with a description of future areas of study related to this research.

Chapter 2

LITERATURE REVIEW

A critical literature review was performed on existing research into serverless computing. This was particularly focused on the performance aspects of serverless. Given the pay-per-use billing model common in cloud-hosted serverless solutions, the costs are intrinsically linked with performance and execution time. The purpose of this review is to set a foundation for research into the effect of the language runtime on both function performance and cost.

2.1 Background

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. Containerisation increased this compute abstraction to the OS layer to increase scalability and portability. Now serverless, or FaaS (Function-as-a-Service), has introduced a new paradigm where the application runtime is also abstracted and all that is delivered for deployment is the application code itself (and associated dependencies). Hendrickson et al. (2016) successfully illustrates this evolution in Figure 2.1.

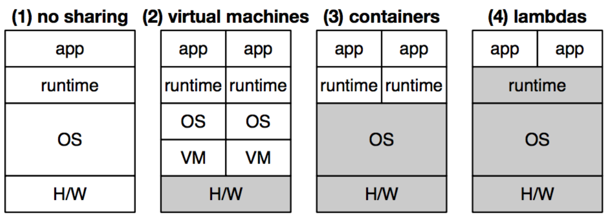


Figure 2.1: Evolution of Resource Sharing in Cloud Computing (Hendrickson et al. 2016)

Crane & Lin (2017) point out that *"serverless computing does not actually mean that code can run without servers"*. Each time a function is invoked, it is executed within an ephemeral container which is (potentially) provisioned in real-time. As a result of these characteristics, they must be inherently stateless (Nasirifard et al. 2017), instead using stateful backing services such as a cache or database.

These changes in cloud computing have coincided with a parallel evolution of service-oriented architecture: the advent of the popular "micro-services" architectural style (Fowler & Lewis 2014). This involves very discretely defined services that are limited in scope to performing one task only. FaaS is a natural fit for this kind of de-composed architecture. Polyglot programming (Wampler & Clark 2010) is a novel aspect of the micro-service approach. This means that given the small and independent nature of each service (function) in a system, development teams now have the potential to choose the best implementation language to suit their skills and the purpose of that service. Reflecting this, most serverless implementations support function implementations in a variety of languages, summarised in Table 1.1. Note that Javascript (via NodeJS) is the only universally supported language runtime. Azure, as expected, have a bias towards Microsoft languages. Amazon Web Services (AWS) added (as of December 2017) support for Google's "Go" (Golang) language. This research concentrated on the runtimes supported by AWS Lambda and Azure Functions, including both NodeJS and C# .NET which are present on both platforms. This gives the opportunity for cross-comparison across platforms of both performance and cost.

2.2 Serverless Performance Considerations

There are two scenarios used in the testing performed for this research, designated as "cold-start" and "warm-start". Cold-start refers to the time taken for a serverless platform to create a fresh container to execute a function and perform any necessary runtime initialisation. Limiting this "cold-start" effect is cited by Varghese & Buyya (2018) as being a key focus for a responsive serverless implementation. The effect is observed by Ishakian et al. (2017) in their study on the suitability of using serverless functions for deep-learning tasks. They evaluate the difference in performance between "cold" and "warm" start scenarios by performing separate test sequences. Cold-start tests wait 10 minutes between function invocations, while warm-start tests involve sequential invocations separated by shorter intervals. Warm-start refers to the "re-use" of internal containers in an effort to reduce the cold-start latency.

McGrath (2017) performed two distinct sets of performance tests on AWS Lambda. One of these was labelled *"back-off tests"* and was used to evaluate the effect of internal container reuse when executing functions through warm and cold-start testing. 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. This is a similar approach to the one taken for this thesis. The performance testing approach for this thesis is described in detail in Chapter 3.

Some interesting observations on serverless performance tuning include those from Eivy (2017), who makes a very important observation in terms of performance tuning serverless functions on AWS Lambda, noting that *"fine-tuning your functions below 100ms won't save any money"*. This means that performance tuning an individual function any less then 100ms is wasted effort as each execution will cost the same. Of course, this assumes the billing model will stay as-is, but these can be subject to change. This changeability is one of the reasons why the Serverless Performance Framework (SPF) was created for this research, with automation and reproducibility a key design aspect. This gives the ability to automatically run performance tests in future to take account of any changes in serverless billing models from cloud providers. The design of the SPF is presented in Chapter 3.

Changeability applies to other aspects of cloud and serverless platforms. As the focus of this research, it is worth noting also that Malawski et al. (2017) list the supported language runtimes of AWS Lambda as only Java, Python and NodeJS. Since then, that list has expanded to include also .NET Core 2.0 and Golang. The addition in December 2017 of the "Go" language to the runtimes supported by AWS Lambda will provide new and novel performance comparisons not yet studied in previous research. "Go" is a framework that is built upon the pillars of rapid development due to a concise language, and fast performance due to native compilation targets (Meyerson 2014). This is expected to perform well against the other AWS options. Additionally, new services such as "Fargate" allow the ability to run docker containers in a similar serverless model (Hunt 2017).

This thesis was limited in scope to testing purely empty functions in order to allow the extension of testing to multiple serverless providers and as many individual language runtimes as possible. The initial scope of the research is all five supported languages (NodeJS, Java 8, .NET Core, Python and GoLang) in a single serverless implementation (AWS Lambda). This subsequently extends cross-platform to Azure Functions (NodeJS, C#). The testing approach is described in detail Chapter 3. Other than empty functions, there are many interesting test cases to explore but which are out of scope for this research. These include the observation by McGrath (2017) that function startup performance can be adversely affected if large third-party libraries are bundled inside a function as this requires copying of these libraries over the network to initalize the runtime environment. Additionally, database interaction was a test performed by Hoang (2017) and noted as a bottleneck issue by McGrath (2017). Given that serverless functions are stateless in nature, they require a stateful backend such as a database to store any application state. This makes data access a common serverless function use-case and would make an ideal aspect to future enhancements to this research.

Lin (2018) builds upon tests performed by Cui (2017*a*) in measuring AWS Lambda performance. In a similar vein to this research, measurement was performed on all AWS supported languages but both experiments focused only on "warm-start" times to assess runtime performance. Additionally, this experiment included API Gateway latencies as tests were triggered via HTTP requests. In contrast, the test approach for this research described in Chapter 3 aims to eliminate all external latencies such as API Gateway layer or network latencies by testing the target functions directly. While Cui (2017*b*) did perform a separate cold-start experiment which has a similar approach to this research, this was a one-off 24-hour test which enforced cold-start by re-deploying a function in each iteration. The SPF framework created to execute performance testing in this research (see Chapter 3) has been designed for automation to allow continuous and repeated testing.

2.3 Serverless Cost Considerations

Serverless applications generally scale rapidly, based on the underlying serverless framework's ability to quickly spin up new instances of a function when required. The functions scale down equally well, to the point where no cost is incurred if the function is not in use (Baldini et al. 2017). Fox et al. (2017) observe that *"No server is easier to manage than* <u>*no*</u> *server"*. This refers to the fact that for serverless applications there are no servers to manage (Varghese & Buyya 2018). This also leads to most commercial serverless implementations being based on a fine-grained "pay-per-use" billing model. For example, AWS and Azure bill per 100ms of execution time. This puts an increased focus on function performance and any variance in startup-time for functions based on different language runtimes have the potential to affect not only response times but application hosting costs also, which is a key aspect of this research discussed in the cost analysis of Chapters 4 and 5.

The criticality of understanding the expected throughput of an application is cited by Baldini et al. (2017), who 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"*. On a similar theme, Eivy (2017) encourages a thorough evaluation of whether a serverless solution will deliver the expected cost benefits in practice. Eivy focuses on the expected volume of function executions to understand where the cut-off lies that makes it more cost-effective to stick with a more typical server-based deployment. Focusing on scientific applications, Malawski et al. (2017) conclude that AWS Lambda is cheaper than equivalent EC2 infrastructure for highly parallel workloads that are required for short periods of time. While the frequency of execution of a function is a key cost consideration, the implementation language of the funciton will also have a potentially important role in any architectural decisions regarding cost effectiveness and is the main focus of this thesis.

Villamizar et al. (2016) perform a cost comparison between a traditional monolithic application deployed to AWS EC2, a de-composed microservices application deployed via EC2 VMs and a serverless implementation using AWS Lambda functions. The VM-based monolith and microservice scenarios were deployed to a variety of EC2 configurations. The focus was to compare relative infrastructure costs to support the same system throughput, measured in TPS (Transactions Per Second). Interestingly, Lambda was shown to be just over 77% cheaper than the monolith implementation and up to 57% cheaper than the VM-based micro-services. However, cross-referencing these results with the research of (Eivy 2017), it can be deduced that the TPS in the test scenario here was only a maximum of 7 TPS. Running the same tests as Villamizar et al. (2016) at a much higher TPS might reveal interesting data about the optimum deployment model at higher throughput. Adzic & Chatley (2017) presents a cost-savings figure in a similar comparison study of up to 99.8%, although the system throughput presented in this example equates to only 0.003 TPS. The cost calculations for this thesis, to be outlined in Chapters 4 and 5, consider a range of system throughput up to a 30k TPS limit representing an enterprise application under heavy load. Additionally, the costs calculations generated by the testing performed for this research are applied in section 4.6 to a CostHat model (Leitner et al. 2016) to examine the cost implications to a large-scale serverless architecture with a throughput of 30k TPS.

Regarding serverless pricing, Eivy (2017) suggests estimating function usage using TPS rather than the provided AWS billing examples of monthly usage - noting that their free-tier offering (AWS offer 1 million free Lambda invocations per month) can be used up very quickly on any high-volume service. He calculates that if your service expects 500 TPS (a high but reasonable throughput), the free tier provides just 0.38 TPS of that 500. An important factor of this calculation that Eivy uses is that each individual function will receive the full free tier allocation - leading to the possible design implications of decomposing the codebase into smaller functions. However, this is not currently the case; the AWS Lambda pricing[1] indicates that the free-tier allocation is spread across all functions, not per-function. Regardless, to aid clarity in cost comparisons across severless platforms any free-tier allocations are ignored in the cost calculations presented in Chapters 4 and 5.

The testing performed for this thesis was entirely based on the minimum allowable memory allocation of 128MB for both AWS Lambda and Azure Functions testing (although, as described in Chapter 5, Azure allocates memory dynamically but bills at a minimum level of 128MB). While outside this scope, Ishakian et al. (2017) observe some interesting relationships between memory allocation, function duration and cost. Firstly, as might be expected, they observed an inverse relationship between memory size and execution time, as performance improves due to the faster CPU associated with higher memory allocations. Secondly, there is a non-linear relationship between memory size and cost; even though billing rates increase for Lambda functions as more memory is allocated to it, the resulting faster execution times offset that higher cost. Again, this would be an expected outcome, given how Lambda allocates CPU and I/O resources in proportion to the chosen memory allocation (Malawski et al. 2017). Similarly, Conning (2017) measures the optimum memory allocation to assign to an AWS Lambda function for lowest-cost execution. Conning shows interesting results that show costs actually decrease as more memory is allocated to the function in addition to the benefit of faster execution times.

Choosing the appropriate programming language is a vital component of managing the performance and execution costs of serverless functions and is the subject of this research. Hendrickson et al. (2016) provides insights into the practical implementation issues of language support on a serverless platform. In particular, the authors note that most languages supported by serverless frameworks are interpreted, using JIT (Just In Time) compilers that have features to optimise code as it runs. They observe this is a performance optimisation challenge, as the serverless function *"may not run long enough on any one machine to provide sufficient profiling feedback"*. This characteristic of JIT compilers affects both .NET and Java on AWS Lambda and is considered when analysing the performance results presented in Chapter 4.

[1]https://aws.amazon.com/lambda/pricing

Cost Factor	Description
Compute	Cost of actual CPU compute time consumed by service execution. For serverless functions, this is assumed by the formula to be linearly dependant on the number of requests (given the usual pay-per-use billing model).
API Calls	Cost of each individual API call invocation. Considered by the formula to be approximately linearly dependent on the number of service requests. For serverless, this applies only to functions accessed by an API layer (e.g. AWS API Gateway)
I/O Operations	Costs of any database interaction or file system access. Like API calls, considered approximately linearly dependent on the number of service requests.
Additional Options	Costs of any other associated services which the service consumes or exposes. For example, the cost of elastic IP addresses. Generally considered a constant value.

Table 2.1: CostHat (Leitner et al. 2016) Formula Input Metrics

While Azure Functions support C# .NET, as discussed in section 5.3, the C# script runtime in Azure differs from the traditional JIT compilation of the .NET CLR used in AWS Lambda.

2.4 Serverless Cost Modelling

There has been some recent research into the creation of cost modelling formulas for serverless (and other cloud infrastructure models). Leitner et al. (2016) present a comprehensive cost modelling framework named "CostHat", which aims to provide cost implication data to developers in real-time as they make changes. This is especially aimed at micro-service based architectures - implemented in either traditional VM-based deployments or Serverless. They present a formula for cost-calculations based on the downstream relationships that are common between individual services in a microservice-based system.

The power of the CostHat model presented by Leitner et al. (2016) is in its recursive nature. A simple dependency graph of services A, B, C and a newly-added dependency (labelled ’D’) is presented in Figure 2.2. The execution times of new

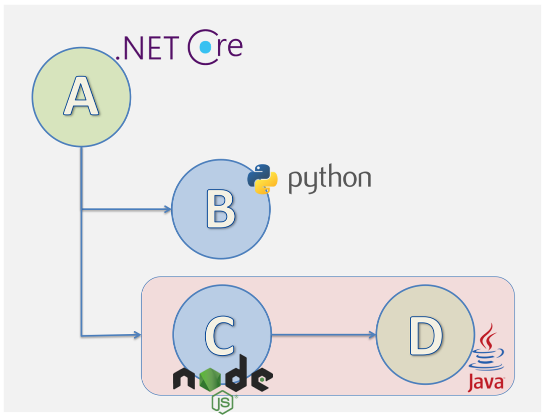


Figure 2.2: Sample CostHat Microservices Dependency Graph

service D has a significant effect on the cost characteristics of upstream services A and C based on the average number of invocations of 'D' that they make. With function 'D' implemented in Java, it would be expected to have a significant effect on upstream function performance, given the expectation that Java will perform poorly compared to some of the other language runtimes on offer (Hoang 2017).

The CostHat formula is based on key cost and performance factors of each individual microservice in a system. It is defined as requiring specific inputs as illustrated in Table 2.1. The values in the table can mostly be gathered via metrics provided by the serverless platform (e.g. via CloudWatch logs for AWS Lambda). With the cost inputs generated as described in Chapter 4, the CostHat model is leveraged in section 4.6 to demonstrate the impact of the cost results produced against a realistic microservices architecture. A sample Python-based implementation of the model is available as open-source on GitHub[2].

2.5 Summary

One of the common characteristics of serverless cloud computing is its billing model which is generally on a "pay-per-use" basis. There is an almost linear relationship between execution time and execution cost. The strength of this relationship is based on the size of the billing increments, currently in 100ms increments. This puts an increased focus on function execution time, which can vary depending on the starting state of the execution environment. Serverless platforms generally host function code via a pool of internal containers, which can either be initialised on-

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

demand to execute the code (referred to as cold-start) or can re-use an existing container (warm-start). Previous studies have shown that the starting-state of a container can have a material effect on function performance.

Serverless is popularly regarded in the industry as being a cost-effective option compared to traditional VM-based cloud computing. This is based on the savings made due to the removal of any need for server management or low server utilisation during quiet periods. However, some studies have pointed out that any saving greatly depends on the throughput experienced by the application. Due the pay-per-request billing model of serverless, at higher loads this could become more expensive than traditional VM hosting. Choosing the most appropriate language runtime for serverless functions could have a material effect on this, the understanding of which is the aim of this thesis.

Chapter 3

PERFORMANCE TEST DESIGN

Chapter 1 introduced the main aim of this research, being the investigation of the impact of language runtime on the performance and cost of serverless applications. Chapter 2 expanded on this to provide areview of research into serverless functions with a specific focus on aspects related to performance and cost, noting that both of these are strongly related to each other given the "pay-per-use" billing model common with serverless platforms. This chapter will outline the approach to performance testing for this research and provides details of the Serverless Performance Framework (SPF) created to perform this testing.

Testing was performed against two of the major commercial serverless platforms - AWS Lambda and Azure Functions. These were chosen due to their market leading position and their support for similar language runtimes, allowing comparisons to be made across platforms. The following five runtimes will be 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.

The creation of a new performance testing framework was required for this research for a number of reasons. Automation was a key requirement of any test harness in order to produce repeatable results from a number of test batches. This testing was required to be triggered on a configurable schedule across multiple cloud-hosted serverless platforms and across any number of different language runtimes. While individual metrics could have been gathered from native services on each cloud platform, the ability to aggregate data from across this spectrum of platforms and runtimes was important to allow efficient and consolidated analysis of test results. Some existing research of a similar nature was reviewed in Chapter 2, which included some examples of other test harnesses created for those studies. However, they had slightly different approaches to their testing which included other components in the performance results such as API Gateways or network latencies. The approach taken for this research to performance testing of serverless functions is to isolate

Test Type	CRON Specification	Interval Period
Cold Start	*"0 * * * *"*	1 hour
Warm Start	*"* * * * *"*	1 minute

Table 3.1: Serverless Performance Test Intervals

all other components from the test results, taking execution metrics directly from the logging and telemetry available within the cloud platforms and triggering the functions based on events rather than direct test-client invocations via APIs.

All target functions to be measured are empty functions. As discussed in section 2.2, this is in order to isolate the test results to purely the performance of the serverless platform itself in creating the execution environment to run the function. By implementing an empty function, the actual language performance is removed from the equation. The purpose of this is to understand how efficiently each tested serverless platform supports its different runtime options.

3.1 Test Scheduling

A series of cold-start and warm-start tests were designed. The "Cold Start" rate was set at 1-hour intervals. As mentioned earlier in section 2.2, cold-start refers to the time taken by a serverless platform to initialise a new execution container in order to execute a function's code. In order to optimise performance, when possible, previously created execution containers are recycled by a serverless platform. The actual lifetime of an execution container in most serverless platforms is indeterminate, as specified in the research conducted by McGrath (2017). Ishakian et al. (2017) settled on a 10-minute cold-start interval in their testing. Back-off performance testing results presented by McGrath (2017) showed that 15-minute intervals was sufficient for Azure Functions. For AWS Lambda, the results of McGrath (2017) showed no observable cold-start slowdown within the highest interval used in his research of 30 minutes. Based on this research, a 1-hour period was considered a reliable timeframe in order to ensure container recycling by the AWS Lambda and Azure Functions platforms. For consistency, the same interval was set for both AWS Lambda and Azure Functions testing.

The "Warm Start" interval rate was set to be 1-minute intervals. This was shown by the research mentioned above for cold-start periods to be reliably well within the typical lifetime of a serverless execution container for both AWS Lambda and Azure

Functions.

The serverless functions for both AWS and Azure are triggered based on a defined schedule. AWS CloudWatch Events and Azure Function Timers are used to support this for AWS and Azure respectively. They provide the ability to schedule regular tasks on either an interval rate or a CRON schedule specification. Rates can be specified in minutes, hours or days. The interval rates for this research are summarised in Table 3.1.

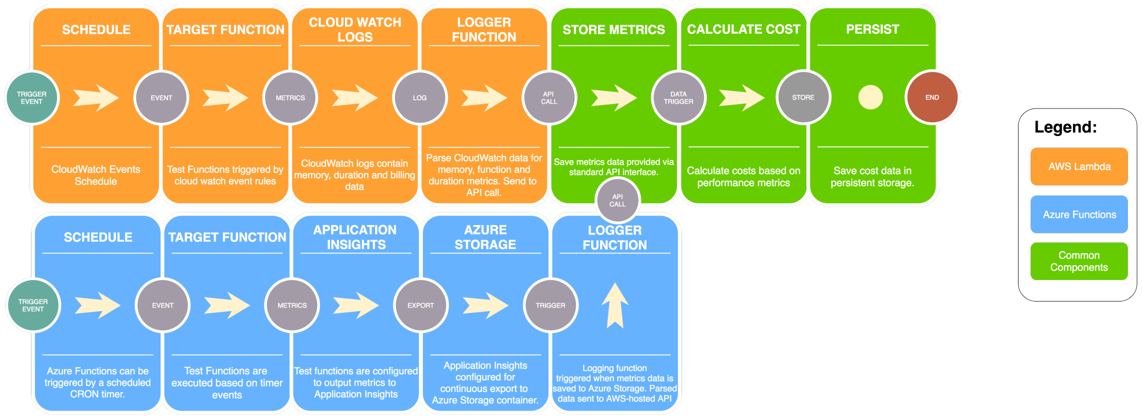

SCHEDULE
TARGET FUNCTION
CLOUD WATCH LOGS
LOGGER FUNCTION
STORE METRICS
CALCULATE COST
PERSIST
TRIGGER EVENT
EVENT
METRICS
LOG
API CALL
DATA TRIGGER
STORE
END
CloudWatch Events Schedule
Test Functions triggered by cloud watch event rules
CloudWatch logs contain memory, duration and billing data
Parse CloudWatch data for memory, function and duration metrics. Send to API call.
Save metrics data provided via standard API interface.
Calculate costs based on performance metrics
Save cost data in persistent storage.
API CALL
SCHEDULE
TARGET FUNCTION
APPLICATION INSIGHTS
AZURE STORAGE
LOGGER FUNCTION
TRIGGER EVENT
EVENT
METRICS
EXPORT
TRIGGER
Azure Functions can be triggered by a scheduled CRON timer.
Test Functions are executed based on timer events
Test functions are configured to output metrics to Application Insights
Application Insights configured for continuous export to Azure Storage container.
Logging function triggered when metrics data is saved to Azure Storage. Parsed data sent to AWS-hosted API
Legend:
AWS Lambda
Azure Functions
Common Components

Figure 3.1: High Level Overview of Serverless Performance Framework

3.2 Serverless Performance Framework

The diagram in Figure 3.1 provides a high-level overview of the SPF (Serverless Performance Framework) created for this research. The colour coding indicates the components that are specific to AWS (orange), Azure (blue) and those that are a common part of the framework which will be used for testing all serverless platforms (shaded green). This includes an API that will be discussed when the framework is examined in detail in section 3.6. The grey circles in the diagram are intended to illustrate the flow of test data and the interfaces connecting each stage in the test execution process. The AWS components illustrate the phases in test execution from the initial event scheduling (via CloudWatch Events) to the collection of function execution metrics from CloudWatch Logs, before parsing these to deliver via the API to the common storage components. The Azure Functions components illustrate a similar set of phases, with the exception of an extra step required to transfer Application Insights data containing function execution metrics to Azure Storage, enabling the triggering from there of the logger function which delivers the metrics to the common storage components by the same API used for storing AWS metrics. The common components (in green) are mainly concerned with the storage of incoming metrics via the published SPF API. The following sections provide details of each of these components. Figure 3.2 contains a more detailed overview of the test framework.

3.3 SPF Common Components

This section describes the components of the SPF that record the performance and cost calculations provided to it via an API call. See Figure 3.2 for an overview of the system components and their relationships. These components are common, meaning that they can be used by tests measuring any serverless platform (not just AWS Lambda), providing they can provide the necessary data to conform to the API (see section 3.6).

The architecture of the solution is described in Figure 3.2. Note that for full source code and usage guide of the framework, see the project repository which is available on GitHub[1].

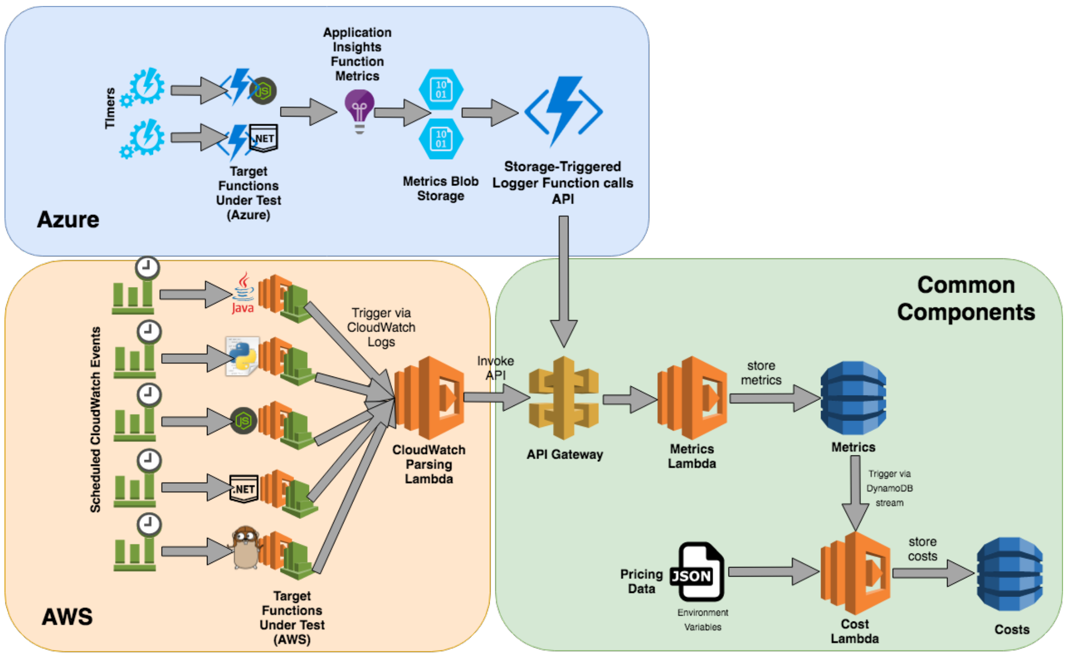

Application Insights Function Metrics
Timers
Target Functions Under Test (Azure)
Metrics Blob Storage
Storage-Triggered Logger Function calls API
Azure
Common Components
Scheduled CloudWatch Events
Java
Trigger via CloudWatch Logs
Invoke API
CloudWatch Parsing Lambda
API Gateway
Metrics Lambda
store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

Figure 3.2: Architecture of Serverless Performance Framework

Metrics Lambda

This function is triggered via the API and performs the simple task of storing the provided metrics into persistent storage. It also calculates some derived data

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

useful for performance comparisons such as function name, language runtime, function duration, memory allocation and serverless platform. This component is implemented as a .NET Core 2 AWS Lambda function.

Cost Lambda

This function is triggered via an event when the main performance metrics are persisted to DynamoDB "Metrics" table (see Figure 3.2). It takes those values, combines with the latest pricing data for the relevant cloud provider and calculates the estimated cost of execution of that function. The pricing data is set as an environment variable on the function and is initialised with a value as specified in the function template created via the Serverless Framework[2]. That calculation is then also stored to DynamoDB persistent storage (in the "Costs" table) in the same way as the initial performance metrics. This component is implemented as a NodeJS (6.10) function and deployed via the Serverless Framework. As discussed in section 4.6, the cost details stored here are the inputs required to feed into the CostHat model (Leitner et al. 2016) used for modelling the cost of a realistic microservices architecture.

3.4 SPF AWS Components

This section discusses at a high level the flow of execution and events that are part of the SPF which are specific to testing AWS Lambda functions. These ultimately connect with the common SPF components described in section 3.3 for storage and calculation of performance and cost data via the API Gateway, which is discussed in detail in section 3.6.

Target Function

The lambda test functions were implemented as completely empty functions. The purpose of this was to eliminate from the testing any particular language runtime attributes in terms of execution performance. The elimination of this allows accurate measurement of the performance of the serverless platform itself in terms of creating the execution environment for the function code. For AWS Lambda, there are five empty test functions in total which represents the five language runtimes supported by that platform. These are: Java 8, .NET Core 2, Python, NodeJS and Go. These were all created and deployed via the open-source Serverless Framework[3] tool. This simplifies serverless function development and deployment across a number of

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

different serverless platforms. 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.

It should be noted that while this research neared completion, AWS released support for NodeJS runtime 8.10 (Lima 2018). While the results presented in Chapter 4 represent the measurements already taken based on NodeJS 6.10 runtime, a short series of comparison test runs were executed against the 8.10 runtime in order to assess whether there was a material difference in terms of performance. These comparative measurements will be mentioned later in Chapter 4 when describing the NodeJS warm and cold start results.

Cloud Watch Logs

By default, AWS Lambda sends three entries to CloudWatch Logs for every lambda function execution. A sample of this output is contained in the code listing below:

```
START RequestId: 0969f327-e65d-4d7c-ab0c-d998dc5537a9 Version: 1.0

END RequestId: 0969f327-e65d-4d7c-ab0c-d998dc5537a9

REPORT RequestId: 0969f327-e65d-4d7c-ab0c-d998dc5537a9
Duration: 118.00 ms Billed Duration: 200 ms  Memory Size: 128 MB
Max Memory Used: 38 MB
```

Any Lambda function can potentially log additional metrics to the same CloudWatch log streams. However, for the purposes of this research, the default metrics provided are sufficient. Additionally, it was a design principle of the SPF implementation that the function being tested would not need to be aware that it was actually *being* tested. This allows potential to measure any existing function using the framework without any code changes necessary.

Logger Function

The CloudWatch log entry for *"REPORT"* contains all the required metrics for performance measurement and metrics that can be used to derive actual costs. These include the metrics shown in Table 3.2. Each of these metrics map directly to the inputs defined in the API described later in section 3.6. A sample of a "REPORT" cloud-watch entry is displayed below:

Metric Name	Description	Usage
Request ID	An identifier for each request that is guaranteed to be unique.	Used to correlate performance and cost data together.
Billed Duration	The actual billed duration for the function as calculated by the serverless platform itself.	Used as the default billed duration for cost calculations.
Duration	The length (in milliseconds) of time it took to execute the function.	Key performance metric. Could also be used to re-calculate function costs if billing increments change in future.
Memory Size	Memory allocated to the function. This usually has impact on the compute capacity also.	This is a key metric for cost calculations.
Memory Used	The actual maximum memory used by the function.	Not currently used.
Function Name	Name of the function that executed	Used for analysis of individual functions.
Function Version	Version of the function that was executed.	Used for analysis of individual functions.

Table 3.2: AWS CloudWatch - Default Logged Metrics Stored via SPF API

```
REPORT RequestId: 3897a7c2-8ac6-11e7-8e57-bb793172ae75 Duration:
2.89 ms Billed Duration: 100 ms Memory Size: 1024 MB Max Memory
Used: 20 MB
```

The AWS Logger component performs the task of parsing this CloudWatch "REPORT" entry and translating these values into a call to the API which exposes the common components of the SPF, which in turn persists the metrics and cost data. This logger component is itself implemented using AWS Lambda as a NodeJS (6.10) function. It is created and deployed using the open-source Serverless Framework[4]. Triggers are specified in the function configuration to allow it to be invoked when CloudWatch "REPORT" entries are made by the test functions. Note that a limitation was encountered during development of the framework which is an AWS Lambda limitation of a maximum of five distinct triggers being specified by a func-

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

tion. This means that only five different Lambda functions can trigger the logger to persist their metrics data via CloudWatch. This is merely a limitation in the number of concurrent test functions that can be measured and was not a problem for this particular research, as only five distinct runtimes were measured in AWS.

3.5 SPF Azure Components

This section discusses the components that were created to allow testing of Azure Functions. These connect with the common SPF components in the same manner as the AWS Lambda test components - via the metrics API discussed in detail in section 3.6.

Test Target Function

Again the test functions were implemented as empty functions, with the same purpose as for AWS Lambda of measuring purely the Azure Function platform's performance in preparing code for execution. For Azure Functions, two runtimes were tested via empty functions: NodeJS (runtime version of 6.11.2) and .NET C#. While there are other runtimes available on Azure Functions, these were chosen for purposes of cross-comparison with AWS Lambda.

Application Insights

Microsoft's Azure platform supports rich logging collection, analysis and visualisation for most of its cloud services via Azure Application Insights. The Serverless Framework[5] was used to create the test functions. By default, this configures each function to log their metrics to Application Insights, passing a set of performance metrics which can be then mapped to the SPF API inputs for storage in a consistent format with AWS Lambda testing. See Table 3.3 for details of the recorded metrics and how they map to the API.

An important note for the collected metrics is that time duration is specified in Azure in "ticks". 10,000 ticks is equivalent to 1ms. That meant that any logging of performance data via the SPF's API would need to divide the duration value by 10000 to convert to milliseconds (ms) and allow consistency of analysis with AWS Lambda.

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

API Input Field	Application Insights JSON Element Mapping
Request Id	*request.id*
Function Name	*request.name*
Function Version	(hardcoded to *"#LATEST"*)
Timestamp	*"context.data.eventTime"*
Duration	*"request.durationMetric.Value"*
Billed Duration	(derived from *"request.durationMetric.Value"*)
Memory Size (Allocated)	(currently hardcoded to *"128"*)
Memory Used	(currently hardcoded to *"128"*)
Language Runtime	(derived from *"request.name"*)
Serverless Platform Name	(hardcoded to *"Azure Functions"*)

Table 3.3: Mapping of Azure Application Insights Metrics to SPF API Fields

Azure Storage

The capabilities provided by Application Insights for logging function performance in Azure are equivalent (for the purposes of this research) to Cloud Watch in AWS. What differs between the two is the ability to trigger functions directly from logs or metrics. In AWS Lambda, the logger function was triggered directly by entries made to Cloud Watch logs. It is not possible to trigger a function from Application Insights directly. Instead, an extra step was required.

Application Insights can be configured for *"continuous export"*. This allows selected parts of the data logged for function execution to be automatically sent to Azure Storage for long-term persistence and further processing. This was used to output only the "Request" metrics data from Application Insights which contained the majority of the data needed to be passed to the SPF API. The advantage of this approach is that 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 relevant section of this JSON structured data is listed below:

```
"request": [
    {
        "id": "49b6ac0e-d475-49aa-a562-380080d2168b",
        "name": "empty-csharp",
        "count": 1,
        "responseCode": 0,
            "success": true,
            "durationMetric": {
                "value": 338749.0,
                 ...
            }
        }
    ],
...
    "context": {
        "data": {
            "eventTime": "2018-04-04T10:37:55.842Z",
            "isSynthetic": false,
            "samplingRate": 100.0
        }
    }
```

Azure Logger Function

The logger function in Azure performs a very similar function to its equivalent in AWS Lambda. After converting the incoming "blob" of data from Azure Storage into JSON, it parses this for the required metrics to send to the API. Table 3.3 shows the mapping of the original Application Insights JSON data to the input fields required by the API. Note that for cost-calculations, a memory allocation of 128MB was fixed in the calculations. This was a result of technical complexities in retrieving the actual memory used during function execution. This value requires retrieval based on an API call back to assess the memory used by the function over a particular time period. Azure prices function execution based on *maximum* memory consumed by a function, not a preset allocation. Despite these complexities, sampling of memory used during test runs via Azure CLI (Command Line Interface) showed consistent memory usage well below 128MB (the minimum level used by Azure for billing). A sample CLI command used is shown in the listing below:

```
az monitor metrics list --resource /subscriptions/<sub-id>/
resourceGroups/azure-service-nodejs-rg/providers/
Microsoft.Web/sites/azure-service-nodejs
--metric "MemoryWorkingSet" --interval "PT1M"
--aggregation Maximum --start-time 2018-04-02T16:26:00Z
```

```
s--end-time 2018-04-02T16:28:00Z
```

3.6 SPF API

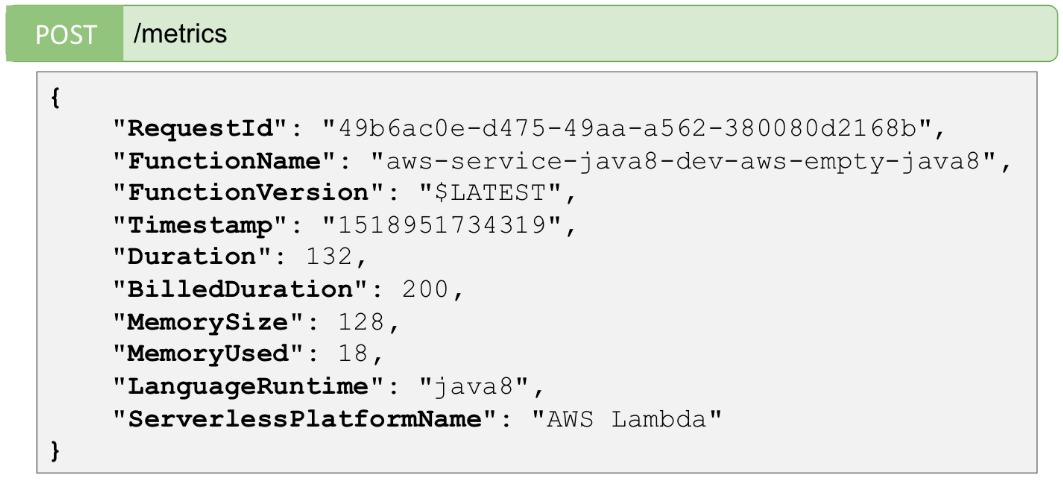
This section describes in detail the design and implementation of the RESTful API created to expose the common functionality for calculation and recording of performance and cost data across all serverless platforms under test.

API Gateway

A key aspect of the SPF test framework is to enable it to record data on function execution from any serverless platform, not just AWS Lambda. To this end, all functionality for recording, calculation and analysis of performance and cost metrics is exposed through a standard API via AWS API Gateway. This allows any framework (e.g. Azure Functions, Google Cloud Functions, OpenWhisk) to gather function execution metrics in any way that is appropriate to that platform and deliver them in a standardised format for analysis via API call.

API Definition

Currently, there is just a single POST operation in the API, although more will be added as part of future work to allow querying of stored performance data in various ways. For example, the ability to search and retrieve metrics summary data would allow creation of dashboards and querying of data through web, mobile or even voice-assistant interfaces. Figure 3.3 contains the formal definition of the existing API, including a sample request body.



POST /metrics

```
{
    "RequestId": "49b6ac0e-d475-49aa-a562-380080d2168b",
    "FunctionName": "aws-service-java8-dev-aws-empty-java8",
    "FunctionVersion": "$LATEST",
    "Timestamp": "1518951734319",
    "Duration": 132,
    "BilledDuration": 200,
    "MemorySize": 128,
    "MemoryUsed": 18,
    "LanguageRuntime": "java8",
    "ServerlessPlatformName": "AWS Lambda"
}
```

Figure 3.3: Definition of SPF API

The single POST operation follows typical REST (Fielding & Taylor 2000) standards of a pluralised noun, *"metrics"*. This expects a body in the request which contains a standard JSON payload of serverless function metrics data, including execution duration, name, version and memory allocation. Optionally, the "billed duration" as calculated by the serverless platform itself can be provided, although this is not essential as it could be calculated based on the other metrics combined with appropriate pricing data. Note that in the example above in Figure 3.3, the language runtime is "Java8". By default, AWS Lambda does not log the language runtime with the rest of the metrics data to CloudWatch logs. Instead, for this experiment, a naming convention was followed to allow derivation of the runtime from the test function name. This could have been achieved by the test function logging the language runtime separately to CloudWatch, but this would have required a separate function to be triggered to update the original record created in the above API call. Also, a design goal of the test framework is to avoid requiring a test function to be aware that is being performance tested or have knowledge of the test framework. This allows in theory any target function to be tested with the SPF.

Metrics and Cost Persistence

The components to persist the serverless function metrics are implemented by a set of serverless (AWS Lambda) functions marked *"Metrics Lambda"* and *"Cost Lambda"* in Figure 3.2. These calculate any derived metrics and store their data in separate DynamoDB tables. This separation of concerns between performance and cost data is important for flexibility in cost calculations. Separating the calculation of cost data allows independent calculation of cost based on the latest pricing data, which is set via environment variables. These could be (in future iterations of the framework) updated via AWS events on an SNS (Simple Notification Service) published topic (Barr 2017). This potentially allows future revision of cost data based on previously-recorded performance metrics.

3.7 Summary

A framework, named "Serverless Performance Framework" (SPF), for performance testing of serverless platforms has been presented. This framework includes a simple API which accepts, parses and stores provided function execution data based on a standard set of inputs. The two most popular serverless platforms, AWS Lambda and Azure Functions, have been plugged into this API to provide metrics and cost data. Sample test functions for each of the major language runtimes supported by each serverless platform have been created. The languages supported for AWS Lambda are NodeJS, Python, Java, Go and C# .NET. The languages supported for Azure Functions are NodeJS and C# .NET. These functions are all completely empty by design. The purpose of this is to measure the performance of the underlying serverless platform in initialising and executing each function, taking out any other latencies. Tests can be triggered based on either cold-start (longer intervals between invocations) and warm-start (shorter intervals intended to promote internal container re-use by the serverless platform) schedules. In Chapters 4 and 5, the results of these tests will be presented and discussed.

Chapter 4

TEST RESULTS - AWS LAMBDA

The aim of this thesis, as described in Chapter 1, is to evaluate the performance and cost implications of the choice of language runtime for serverless function implementation. This includes the assessment of all five available AWS Lambda runtimes: NodeJS, Python, Go, .NET and Java. The test approach and implementation was discussed in detail in Chapter 3. Chapter 5 will discuss separately the results of Azure Functions testing, including a comparison against the AWS Lambda results presented in this chapter. A general discussion of all findings follows in Chapter 6.

This chapter will present and discuss the tests results for AWS Lambda. Test result data was exported from the DynamoDB tables in AWS (see Chapter 3 for details) using a popular open-source tool[1] called *"DynamoDBtoCSV"*, which exports entire DynamoDB table contents to CSV (comma-separated-value) files. *Google Sheets* was then used to import and analyse the CSV data. This chapter also includes detailed analysis of the cost implications of the performance test results, including the application of these costs to a CostHat model (Leitner et al. 2016) to understand how these costs might affect an overall serverless architecture cost footprint.

AWS Lambda testing involved creating completely empty test functions in each of the supported languages: Go, NodeJS 6.10, C# (.NET Core 2), Python 3.6 and Java 8. Each function was executed on the minimum memory allocation available of 128MB. There were two different scenarios under which each of these functions were measured. Cold-Start testing was implemented to measure the performance of each function under conditions where a fresh "container" would need to be created to execute that function. Warm-Start testing was implemented to measure the same functions under conditions where AWS Lambda is likely to be able to re-use an existing "container" from a previous execution of the same function[2]. This should result in faster performance. Cold-start tests involved function execution intervals of 1 hour. Warm-start test function execution intervals were at 1 minute. See Chapter 3 for details of how the test environment was created.

[1]https://github.com/edasque/DynamoDBtoCSV
[2]https://docs.aws.amazon.com/lambda/latest/dg/running-lambda-code.html

4.1 Warm Start Test #1

The first warm-start test was run over a single 60 minute period, involving 60 individual invocations of the five AWS Lambda language runtime functions. These test runs were designed to get an initial impression of respective language framework performance in AWS Lambda under warm-start conditions. The results of this initial warm-start test showed some very surprising findings. Figure 4.1 shows the average execution time across the 60 invocations for each language runtime.

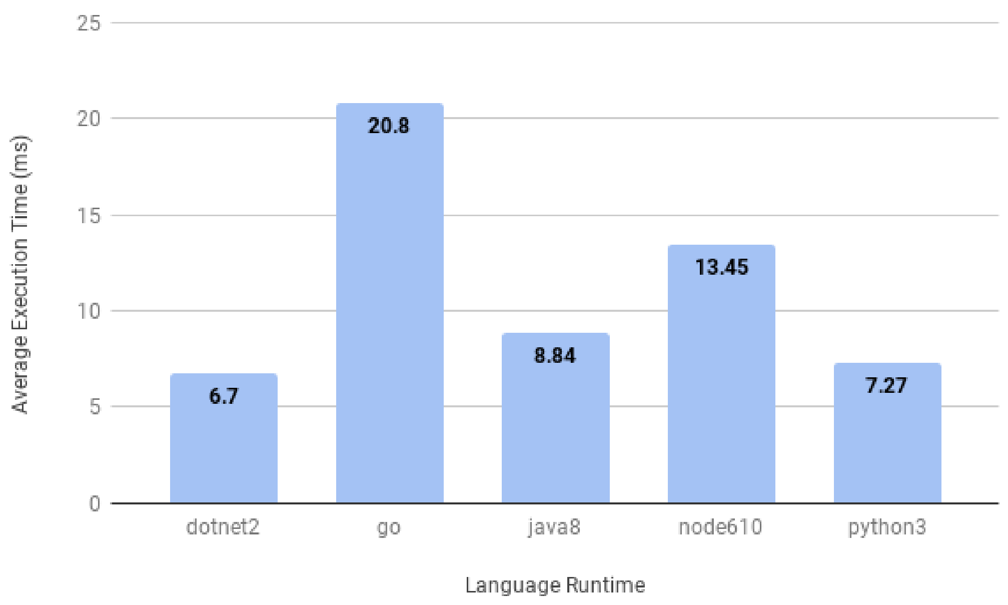


Figure 4.1: Average Execution Time in AWS Lambda Warm Start Test #1

Assumptions before the testing was conducted included expectation that the statically-typed language runtimes such as .NET and Java would perform poorly compared to the dynamically-typed languages (including Python and NodeJS). Go, although a statically typed language, has certain performance features (such as native binary compilation more like C++) that made it hard to predict it's performance prior to testing.

Initial warm-start results, as shown in Figure 4.1, indicate a surprising winner in terms of performance of .NET Core 2. This was quite unexpected, due to the JIT nature of the compiler. It outperforms even the dynamically typed languages of Python and NodeJS. One possible explanation is that this test performed is an *"empty function"* test. Performance of the language itself in terms of active code execution is not included in this test. The test is more designed to evaluate

AWS Lambda performance in managing the environment required to execute code. Despite this, the results are still very interesting, given the additionally unexpected solid performance of Java 8, which is only slightly (on average) slower than Python. .NET and Java applications, executed at runtime via the .NET CLR (Common Language Runtime) and JVM (Java Virtual Machine), had been expected to take longer to initialise. Finally, the laggard in the first warm-start function test is Golang, at an average runtime performance of more than double some of the top performing runtimes. It will be useful to evaluate Golang in more tests and wider data sets in section 4.3, given the reputation of Golang for fast execution.

4.2 Cold Start Test #1

In total, two separate cold-start test phases were executed to evaluate cold-start performance of the empty functions. The first test run was performed over a single period of 24 hours. At 1 hour intervals between function invocations (to ensure a cold-start scenario), that means 24 individual function executions for each of the five empty functions. These test runs were (like the first warm-start test) designed in order to get an initial impression of respective language framework performance in AWS Lambda under cold-start conditions. The results of this initial cold-start test showed some interesting contrasts to the initial warm-start test discussed previously. Figure 4.2 shows the average execution time of the five language runtimes in this initial cold-start test run. The contrast with the warm-start test is stark. As expected, overall the execution times are longer than warm-start times. The contrast, however, for some of the language runtimes is startling. The *.NET Core 2* runtime shows the largest increase (a massive 38,776%) in cold start time vs. warm-start scenarios. *Java 8* also shows a significant (although not quite as dramatic) increase of 5,026%. The other three runtimes show more consistent performance between cold and warm-start. Some even show a *decrease* in average execution time which is puzzling, especially in the case of Golang with a drop of 58%. This finding indicates a need to run further tests to determine average times over a larger data set. The differences are summarised in Table 4.1.

.NET Core 2 showed unexpectedly strong performance in the initial warm-start test. This makes its dramatically slow performance in this cold-start scenario all the more surprising. Average empty function duration has increased from 6.7ms to 2604.65ms. If subsequent test runs show consistent figures (see sections 4.3 and 4.4) in warm and cold start scenarios, this provides some interesting guidelines in the suitability of *.NET Core* as a language of choice for AWS Lambda. The most

Language Runtime	Warm Start Average (ms)	Cold Start Average (ms)	Warm - Cold Increase (ms)	Warm - Cold Increase (%)
.NET Core 2	6.7	2604.65	2597.95	38776%
Golang	20.8	8.74	-12.06	-58%
Java 8	8.84	453.19	444.35	5026%
NodeJS	13.45	28.55	15.1	112%
Python	7.27	6.62	-0.65	-8.94%

Table 4.1: Difference in Initial Cold and Warm Start Tests: Average Execution Times

obvious conclusion is that, if possible, *.NET Core* should only be used in functions that are frequently accessed and are less prone to significant scale-out events. This will be discussed further in Chapter 6.

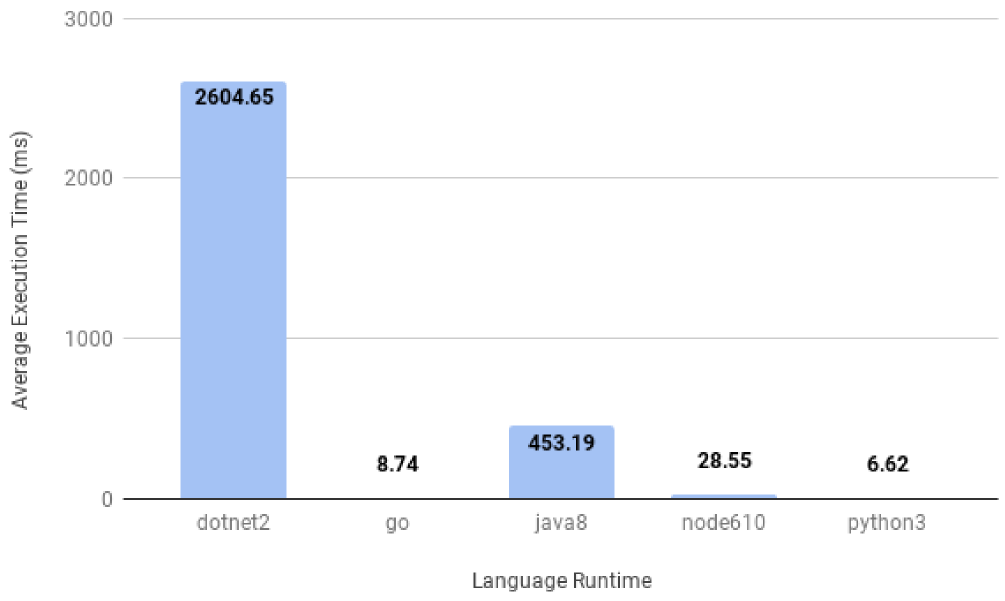


Figure 4.2: Average Execution Time in AWS Lambda Cold Start Test #1

4.3 Further Warm Start Tests

A series of further warm-start tests were conducted, similar in length and interval to the original warm-start test. Three further individual 1-hour tests were conducted (designated "warm-start 2/3/4") in order to provide a wider data-set for analysis, and for comparison with the original warm-start test. A basic average of execution times (similar to that presented in Figure 4.1) is presented in Figure 4.3 for comparison.

There were a total of just over 180 tests for *each* language runtime in this phase.

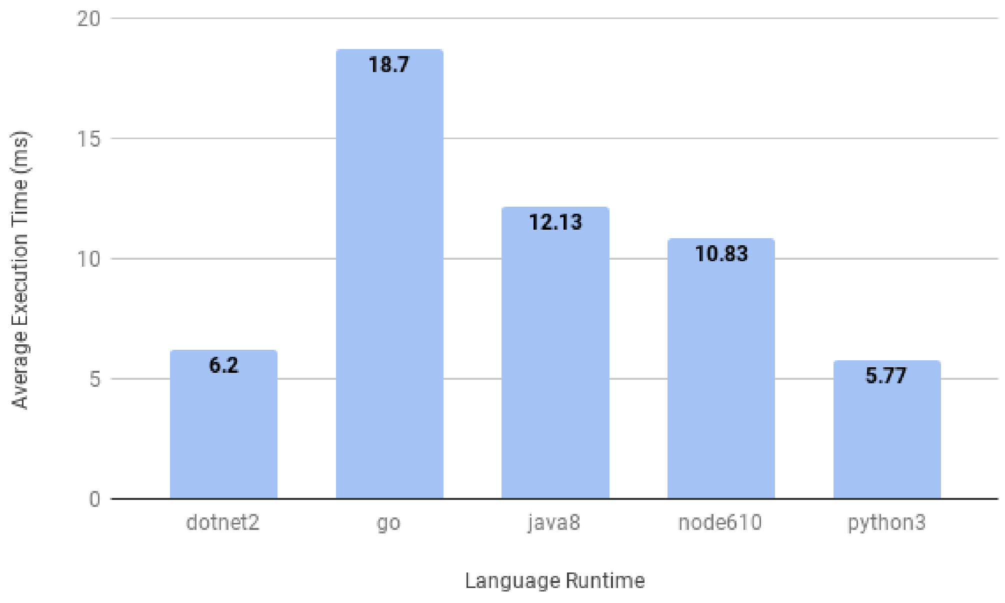


Figure 4.3: Average Execution Time in AWS Lambda Warm Start Test #2/3/4

There is some slight variation in results, with some runtimes (Python, NodeJS) performing, on average, marginally better and some (Java, .NET Core 2) slightly worse. This confirms the results from the initial warm-start test that .NET Core 2 performs considerably well in a warm-start scenario (almost as well as Python, the leader in this test). Golang continues to raise questions, however. When compared with its cold-start results, it seems to perform over 100% **slower** in a warm-start scenario than cold-start (18.7ms against 8.74 in cold-start). This invites further investigation to understand this unexpected result, as logically it does not make sense for cold-start to perform better, given you would expect AWS Lambda to have more internal tasks to perform to setup the runtime container in order to execute the code. Note, however, that Go is unique in that is is a compiled native binary executable (rather than running as an interpreted language). This may offer a clue to this curious performance.

NodeJS 6.10 runtime averages at 10.83ms per request in the warm-start scenario. As mentioned earlier in Section 3.4, AWS released support for a more recent NodeJS runtime version (8.10) during the course of this research. While there not sufficient time to replace the Node6.10 results with full tests based on Node8.10, a sample of 200 warm-start tests were executed against 8.10, showing an average execution time of 17.78ms. This actually hints at a longer execution time by an order of about 70% compared with NodeJS 6.10 runtime but further investigation of the results would be useful future work.

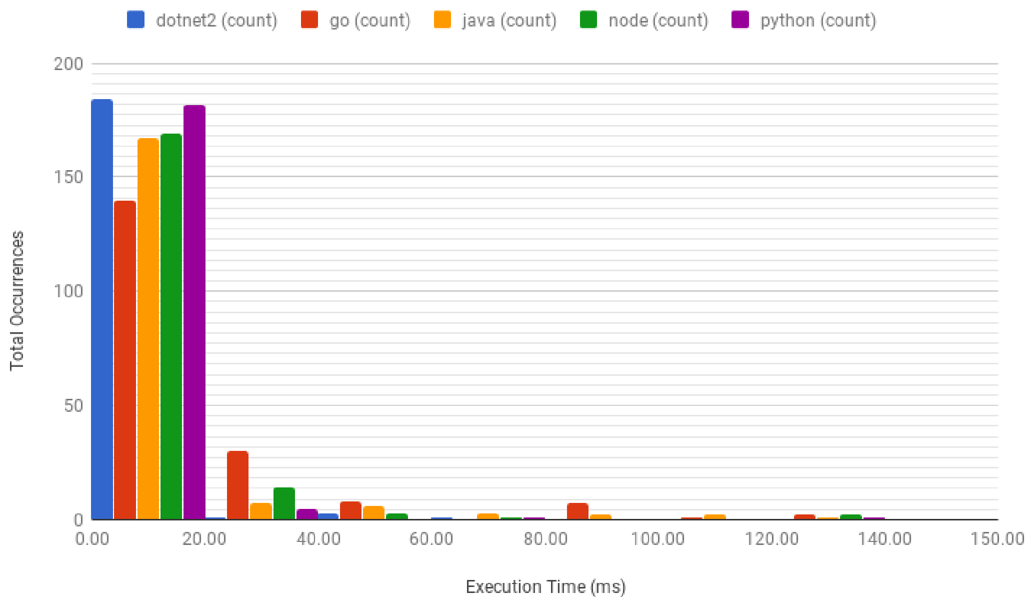


Figure 4.4: Histogram of AWS Lambda Warm Start Test #2/3/4

To analyse the results further, it is necessary to look beyond the simple average (mean) execution time metric. A histogram was created to show the distribution of execution times in buckets of 20ms. This can be seen in Figure 4.4. What this diagram helps illustrate is the reasons for Golang's poor average execution time. Compared to the other four runtimes, it's execution times are far more variable and distributed. This is in fact in stark contrast to it's cold-start distribution, as seen in the box-plot of that test described in section 4.4. The histogram also illustrates the consistency in execution of the top performers, .NET Core 2 and Python. Almost all executions (with the exception of a few outliers) are within the 0-20ms range for these runtimes.

The data revealed by the histogram in Figure 4.4 can be enhanced with a box-plot that shows some other interesting data points (see Figure 4.5). In terms of the distribution and consistency of execution times for each of the five language runtimes, the box plot presents a similar picture to the histogram. The variability of the execution times in Go (and to a lesser extent Java) are 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). It is interesting to observe the relative medians across the five runtimes, which are shown in Table 4.2. The median values are displayed in the table and are represented in the box plot in Figure 4.5 by the dividing line in the inter-quartile range box.

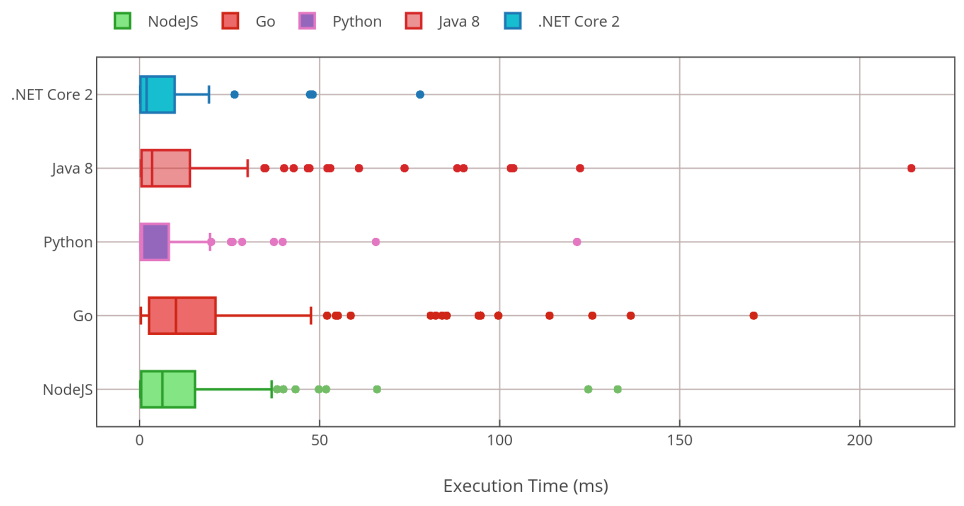


Figure 4.5: Box Plot of AWS Lambda Warm Start Test #2/3/4

These, for all five runtimes, show a much narrower range of values for the lower 50% (1st to 2nd Quartile) of values compared to the upper 50% (3rd to 4th Quartile). This is particularly acute for Python (the median for which is so close to the lower-quartile of the box plot that it is hardly visible in Figure 4.5). For this runtime, the fastest half of the results fall in a range of just 0.21ms, while the slower half of the test results have a relatively wide range of 19.11ms.

Another notable aspect of the median execution time for each language is how much they differ from the average execution times. For Python, they reveal that 50% of tests fall under a very small 0.45ms of execution time. While .NET Core 2 average was very close to Python in terms of performance (6.20ms compared to 5.77ms), their medians differ by a relatively large margin. .NET Core 2 median is 1.96ms, almost five times that of Python. This reveals that although on average they are very similar in performance, Python is far more likely to provide faster performance on a more consistent basis. Similarly, the medians compared with averages for Java and NodeJS have an interesting relationship. While on average, Java is nearly 20% slower than NodeJS, it can be seen from the figures in Table 4.2 that in fact, half

Language Runtime	**Median (ms)**	**Average (ms)**	**1st-2nd Quartile Range (ms)**	**3rd-4th Quartile Range (ms)**
.NET Core 2	1.96	6.2	1.8	17.32
Java 8	3.47	12.13	3.05	26.55
Python	0.45	5.77	0.21	19.11
Golang	10.1	18.7	9.74	37.51
NodeJS	6.34	10.83	6.14	30.34

Table 4.2: Comparison of AWS Warm Start Results Distributions

of Java tests fall under 3.47ms, compared to 6.34ms for NodeJS (nearly double the median for Java). This implies that Java suffers from inconsistency but can deliver very fast execution times in a majority of cases.

4.4 Further Cold Start Tests

A second cold-start test was run over a longer period than the first phase, consisting of 144 hours (6 days). This involved 144 individual invocations of each test function. This was done in order to measure cold-start performance over different overall environment conditions which may occur at different days of the week or times of the day. The results of this second test are generally in line with the findings of the first test, which was over a smaller sample of just 24 hourly invocations over 1 day (see section 4.2). Average execution times for each language runtime across both phases were roughly similar, although all runtimes (apart from Go which was fractionally higher) showed a reduction in average execution time of between 10-20%. Python showed an average execution time of just 2.33ms, a figure 65% lower than the first test. This shows the value of performing a test over a much longer period of 6 days, giving a wider range of values that is less prone to suffering skews from individual busy periods. It is possible the first cold start test was executed at a particularly "busy" time in terms of overall compute demand on the AWS platform in the target region (US-East-1). Perhaps unknown environmental conditions within AWS infrastructure played a part or it was just a result of the smaller sample size in the initial cold-start tests.

An interesting aspect of the larger cold-start results set from this second test is to analyse the distribution of execution times via a histogram, as seen in Figure 4.6. This puts the slow cold-start performance of .NET Core 2 into perspective against the other runtimes. It also shows the relative consistency in execution times of

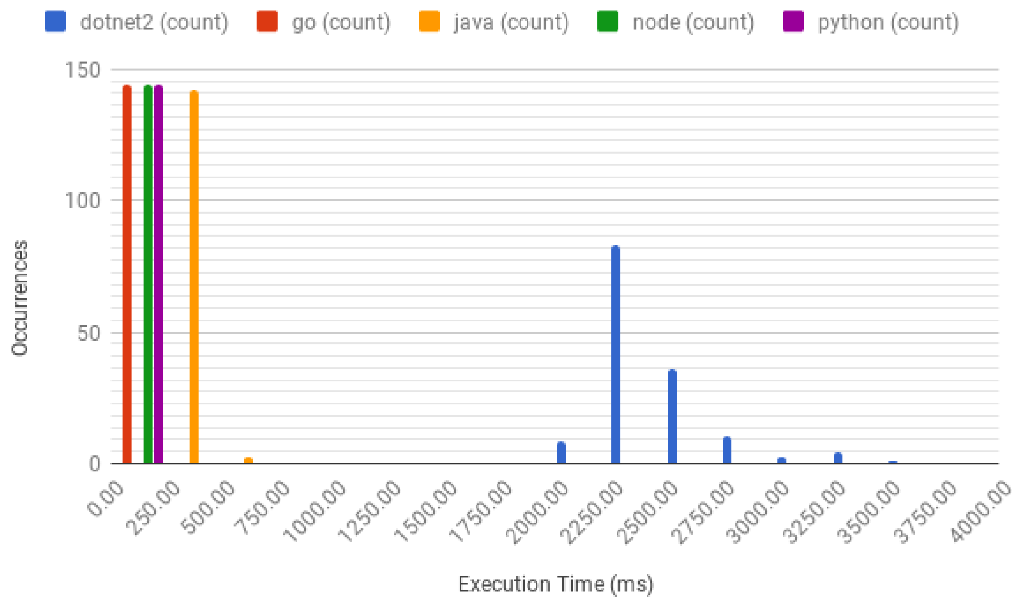


Figure 4.6: Distribution of Execution Times for Cold Start Scenario

the other runtimes, each of which are in a single bucket range, ignoring a small number of Java outliers. This graph also highlights the consistently and relatively fast execution time of Golang in the cold-start scenario (along with Python and NodeJS). This is in contrast to it's laggardly performance in the initial warm-start test.

Overall, Golang shows sub-20ms average warm and cold-start times (overall 19.21ms and 8.97ms respectively). What is difficult to understand is the clearly better performance of Go in the cold-start tests against warm-start. It performs over 100% slower in *warm-start* scenarios. This is counter-intuitive. Indeed, Python shows a similar pattern. In the case of Go, perhaps this is related to its native binary compilation compared to dynamic execution of the other runtimes, but why this would result in much faster cold-start times is unknown. Another possibility is the number of tests in each scenario, with warm-start overall accounting for a total of 250 tests while cold-start consisted of a total of 168. Further investigation is required (see description of future work in section 6.2).

The histogram in Figure 4.6 additionally gives an indication of the distribution of .NET Core (and to a lesser extent) Java 8 test results. The performance of .NET in particular is wildly erratic over a significantly wide range of execution times (2159ms to 3503ms). This is particularly slow and un-reliable performance.

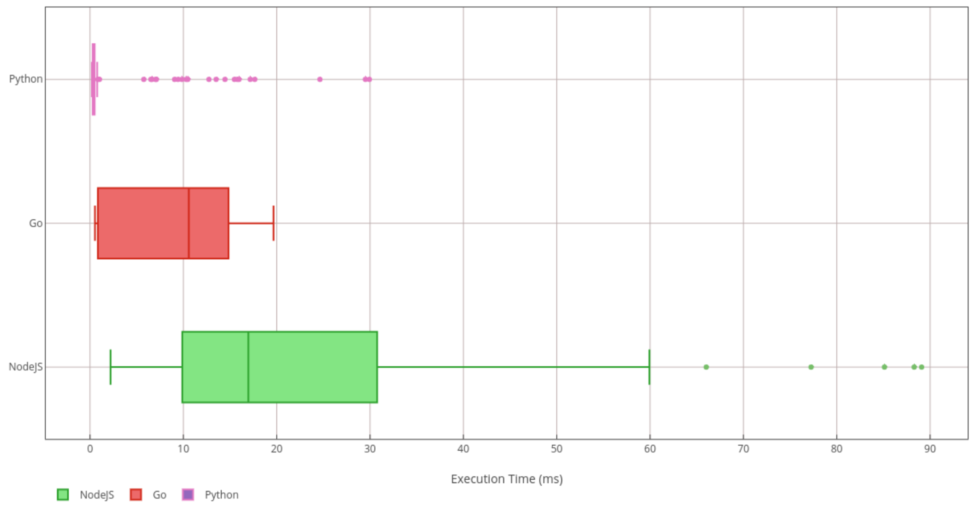


Figure 4.7: Box Plot of Top Three Performing Runtimes in Cold Start Scenario

Given the size of the buckets in the histogram (mainly due to the differential between .NET Core and the other runtimes), it is hard to view the results distribution of the top three performing runtimes - Golang, Python and NodeJS. To dig deeper into the results of these three runtimes, which were all on a similar scale, a comparative box-plot was created, as can be seen in Figure 4.7. The consistent performance of Golang is in evidence from its box-plot, which indicates no outliers outside the upper-whisker - the 1.5 multiplier of its inter-quartile range from the upper quartile (Tukey 1977). Python shows a consistency of its own but in a different way, as can been seen in it's own box-plot. While there are a number of outliers up to a maximum of 29.93ms, the inter-quartile range of Python function executions is very small. Although hard to read from Figure 4.7 due to its relative scale to Golang and NodeJS, it has an inter-quartile range (containing 50% of test results) of just 0.2125ms, with 75% of all tests performing under 0.52ms.

NodeJS shows the largest variance of these top-three performers, with a minimum execution time of just 2.2ms but an inter-quartile range of 21ms. In fact, the difference between the upper quartile (31ms) and upper-whisker (60ms) is a significant 29ms, demonstrating that the upper 25% of test results show a relatively high level of volatility in terms of performance. It is difficult to explain this volatility of NodeJS compared to Golang (Python performs so strongly, it can hardly be compared to the other runtimes). This might be due to the binary executable nature of a Golang application providing a more predictable performance.

As with warm-start tests for NodeJS, a shorter sample of testing against NodeJS 8.10 runtime, released by AWS during this research, were conducted. These revealed comparative performance with 6.10 runtime in cold-start scenario, all tests falling within a similar range with a maximum observed (of just 24 hourly tests) of 64.88ms (compared to 89ms for NodeJS 6.10).

4.5 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. For example, AWS Lambda bills in 100ms blocks for its regional serverless functions (or 50ms blocks for its *"Lambda@Edge"* offering which is NodeJS based only at their network "edge" locations).

For AWS Lambda, there are three main factors in a function's execution cost - execution time (per 100ms), fixed cost per individual function execution and memory allocated to the function. Like other major commercial serverless platforms, AWS Lambda provide a "free-tier" allocation of (at the time of writing) 1 million executions per month and 400k GB/s of execution time[3]. For the purposes of comparison across serverless platforms, these free-tier allocations are excluded from cost calculations. As mentioned in section 2.3 earlier, for high-volume applications, that free allocation is consumed very quickly.

Table 4.3 contains the cost calculations based on the performance data from all combined cold-start and warm-start tests described in the previous section. Costs 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.

As individual function execution costs are so low, to make the costs more relevant, the dollar figure displayed is based on cost for 1 million executions of the function. While this may sound like a lot, for a real-world serverless function of reasonably high throughput, 1 million executions over 1 day equates to 11.574 TPS (Transactions Per Second). High-volume functions, and certainly multiple functions within a micro-service architecture, could expect to have a far higher throughput.

That said, what the cost calculations show is that for individual functions, the effect

[3]https://aws.amazon.com/lambda/pricing

Language Runtime	Test Type	Average Execution Time (ms)	Average Billed Duration (ms)	Average Cost Per Function ($)	Average Cost Per Million ($)
.NET Core 2	Cold	2500.09	2600.00	0.00000561775	5.61775
Golang	Cold	8.97	100.00	0.000000408375	0.408375
Java 8	Cold	391.91	400.00	0.0000010335	1.0335
NodeJS	Cold	23.67	100.00	0.000000408375	0.408375
Python	Cold	2.94	100.00	0.000000408375	0.408375
.NET Core 2	Warm	6.32	100.00	0.000000408375	0.408375
Golang	Warm	19.21	100.00	0.000000408375	0.408375
Java 8	Warm	11.33	100.00	0.000000408375	0.408375
NodeJS	Warm	11.46	100.00	0.000000408375	0.408375
Python	Warm	6.13	100.00	0.000000408375	0.408375

Table 4.3: Performance of Language Runtimes in Cold and Warm Start Tests Mapped to Cost

of language runtime on *cost* is not as significant as is the effect on performance for the top three language runtimes in cold-start scenarios. The long initialisation times for Java and, in particular, .NET Core do have a material effect on cost, however, as can be seen from the figures in Table 4.3. Calculated per million requests, .NET costs significantly higher than all other runtimes due to its comparitively high average execution duration.

The data in Table 4.3 is displayed in graphical form in Figure 4.9. This shows that only .NET and Java have any effect on the cost of function execution, based on the per-100ms billing model of AWS Lambda. In the chart, the economic effect of running .NET functions in a cold-start scenario is quite apparent - for 1 million invocations, the cost is $5.62 compared with just $0.41 for the three runtimes (Golang, NodeJS and Python) which have an average well below the 100ms billing increment. This is a 1,371% increase in cost. For Java, the increase in cost is also significant, given average cost of $1.03 for 1 million invocations. This is 251% higher than the three top performers. To put this into context, Table 4.4 shows the effect of this cost differential based on increasing levels of throughput, measured in TPS.

The table is along the lines of that presented by Eivy (2017) when outlining the effect of rising TPS on the cost of execution a serverless-based solution. To put the TPS used in these calculations in context, a very high throughput web application (as

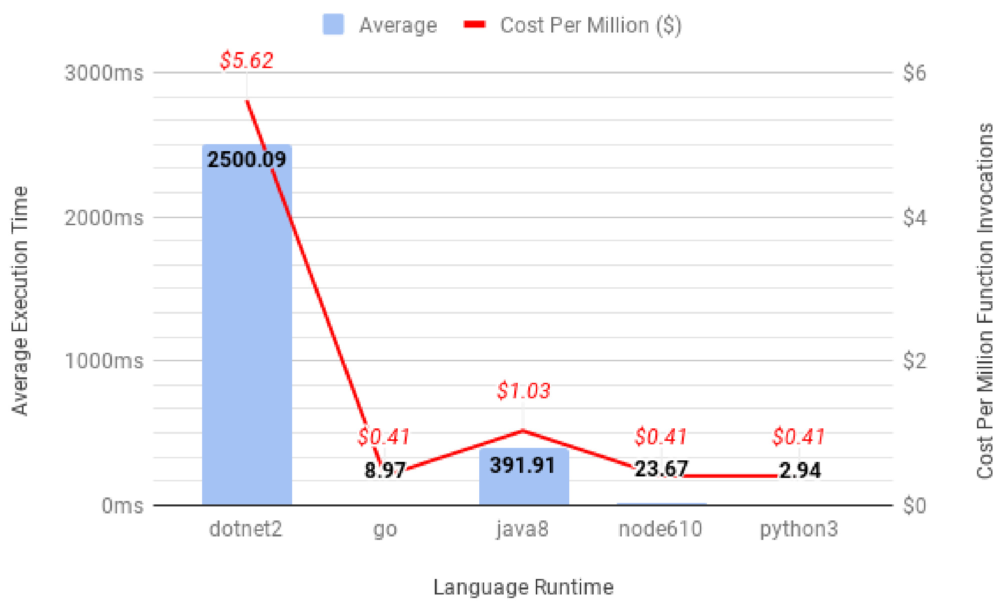


Figure 4.8: Combination Chart Showing Cold-Start Performance and Cost

Language Runtime	Cost Per Day @ 10-TPS	Cost Per Day @ 100-TPS	Cost Per Day @ 30k-TPS	Cost Per Year @ 10-TPS	Cost Per Year @ 100-TPS	Cost Per Year @ 30k-TPS
.NET Core2	$5.03	$50.34	$15,101	$1,837	$18,373	$5,511,980
Golang	$0.35	$3.53	$1,059	$129	$1,288	$386,355
Java 8	$1.07	$10.73	$3,219	$392	$3,916	$1,174,913
NodeJS	$0.35	$3.53	$1,059	$129	$1,288	$386,355
Python	$0.35	$3.53	$1,059	$129	$1,288	$386,355

Table 4.4: Cost of Cold-Start execution at Varying Throughput (TPS)

in Eivy's example) would expect to incur about 30,000 TPS. The data in Table 4.4 shows that as TPS rises, the effect of using .NET or Java becomes significantly costly, although this calculation is based on cold-start figures. It would be reasonable to argue that as TPS rises, the chances of container re-use also rises, leading to warm-start performance and cost rates rather than cold-start. That said, if the traffic patterns are of a highly volatile nature, the chances of cold-start also rise. This is an aspect of cost calculations that would benefit further study based on specific variable TPS scenarios. Also, if multiple functions are invoked as part of a request, a total 30k TPS is more achievable. This is described further via CostHat modelling in section 4.6.

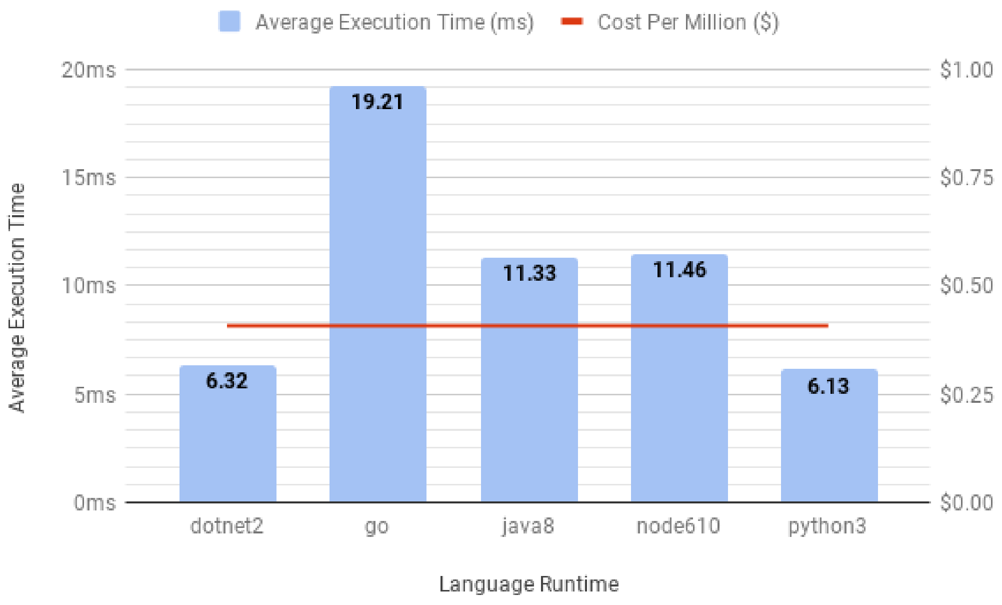


Figure 4.9: Combination Chart Showing Warm-Start Performance and Cost

The warm-start scenarios show an important result in terms of the cost implications of language runtime. Figure 4.9 shows the combined average execution times across all four individual warm-start test runs. Overlaid on those averages is a line indicating the associated cost of execution for each language runtime based on that average execution time. Given the current 100ms billing increments of AWS Lambda, and the maximum average execution time (for Golang) of 19.21ms, it can be seen that for warm-start scenarios, there is currently no material impact on cost for the language chosen. This is despite the fact that, for example, at an average execution time of 6.13ms for Python, it is three times faster than Golang (19.21ms) and almost twice as fast as NodeJS (11.46ms). However, this conclusion comes with some caveats, which are discussed further in Chapter 6.

4.6 Cost Hat Model

The "CostHat" model is a microservices cost-modelling algorithm developed by Leitner et al. (2016). This was discussed in detail earlier in section 2.4. One of the purposes of this cost model is to understand the implications of changes to individual services on the costs of a wider eco-system of services or functions. The model takes into account compute, API, I/O and any other associate costs for each service or function. Leitner et al. (2016) describe the application of this model to both traditional instance-based (e.g. VM) service implementations as well as serverless function architectures. This makes it a useful model to investigate the costs of a more complex set of serverless functions rather than the single function tests performed and analysed in this thesis.

Cost Hat Analysis of Serverless Performance Framework

The first application of the CostHat model is to the Serverless Performance Framework (SPF) which was created to perform the testing for this research. The architecture in terms of individual functions, I/O (via DynamoDB database interactions) and API Gateway interactions is displayed in Figure 4.10.

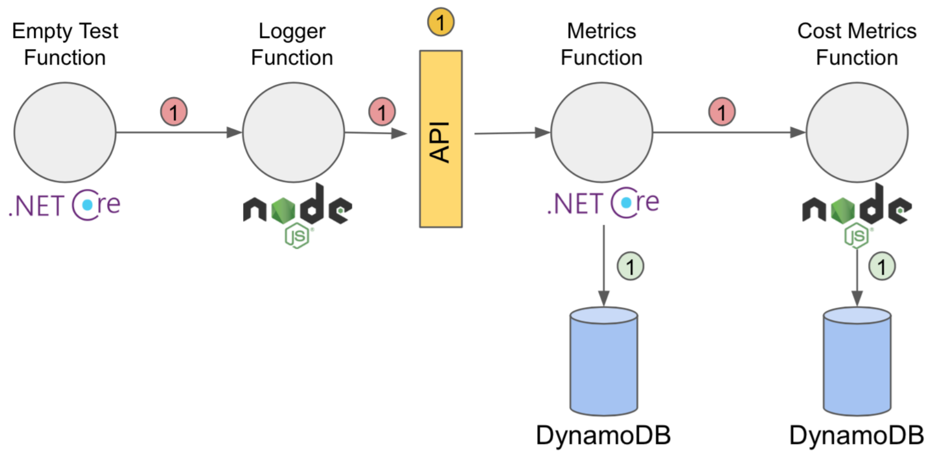


Figure 4.10: CostHat Model of single test function in Serverless Performance Framework

This is a fairly simple model - showing the chain of function calls involved in the measurement and storage of performance data for a single test function, implemented in .NET Core. As described in the overall system design in Chapter 3, each test function invocation results in an execution of a logger function to parse logged metrics. This logger function makes a single API invocation of the metrics function, which writes the metrics once to a DynamoDB table. This triggers the cost-metrics function which makes a single write to another DynamoDB table. The CostHat model created for this via the reference Python implementation[4] is shown in Appendix C.

Using figures based on cold-start performance results and calculated "per million requests" (for clarity of numbers), the applied CostHat model shows that to measure

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

Function	**Runtime**	**Cost ($)**	**Alternate Scenario Runtime**	**Alternate Scenario Cost ($) (*Estimated)**
Test Function	.NET Core 2	5.62	.NET Core 2	5.62
Logger Function	NodeJS	22.7	NodeJS	2.96*
Metrics Function	.NET Core 2	24.89	NodeJS	6.55*
Cost Function	NodeJS	2.96	NodeJS	2.96
Total:		**56.17**		**18.09**

Table 4.5: Cost Hat Details of SPF Single Test Function - Cold Start (Per Million Requests)

the performance of a single .NET Core test function will, in total, cost $56.17 for 1 million invocations. Like all the costs calculated in this research, this excluded any free-tier allocations. This figure is mostly composed of the cost of the .NET-based metrics service which takes an average billed duration of 10,800ms to execute. In addition, given the logger function which calls the metrics function via the API is making a synchronous call (it waits for completion), this function also takes a similar length of time, despite being implemented in NodeJS (a runtime shown to be faster in the results presented earlier in section 4.4). The full breakdown of this CostHat model's costs can be seen in Table 4.5.

The figures in Table 4.5 also show the potentially lower costs generated by the same CostHat model, but where the .NET metrics function is re-implemented as a NodeJS function. Given the faster performance in AWS of NodeJS (see section 4.4), this results in a significant cost saving (total of $18.09 compared to $56.17), which includes a knock-on effect of reducing the logger function synchronous wait-time for API call completion. NodeJS is chosen as an alternative runtime for metrics function in this example as we can estimate the cost based on the similar existing NodeJS function for Cost Metrics, although in this case with additional fixed cost of API calls and data-out transfer costs. This provides a realistic example of the cost implications described in this research of choosing the most appropriate language runtime for function implementation.

Cost Hat Analysis of Modified SPF

A second CostHat model was created to represent a scenario of a high throughput system with a total of 30k TPS. This aims to show the knock-on effects of language runtime cost differences on a high-volume microservices architecture. See Figure 4.11 for a graphical representation of this model. Appendix C contains the simple SPF CostHat model which is of similar structure. The full content of this more complex model is available on GitHub[5].

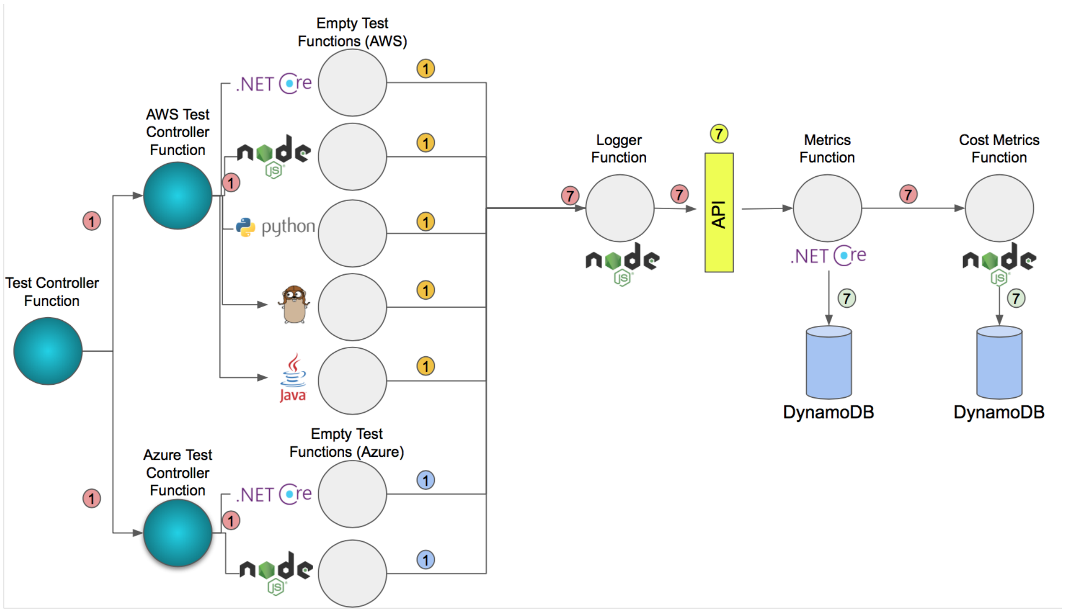


Figure 4.11: CostHat Model of Modified Serverless Performance Framework

This modification of the SPF seen in Figure 4.10 represents the full set of test functions across both AWS and Azure (seven in total). The additions which are not part of the SPF are "controller" functions which could in theory be implemented to programmatically manage test runs in a scalability scenario, where a main controller delegates to each of the AWS and Azure controllers. In this example scenario, we may want to use SPF to measure performance at scale, where each test function is invoked by the controllers at a far more significant rate than was used in the warm or cold-start testing in this research. In this sample architecture, each single call to the Test Controller Function (far left of the diagram) results in a total of 30 calls to the other functions. This consists of 2x AWS/Azure controller functions, 7x test functions, 7x logger function calls (one per test call), 7x metrics function calls and 7x cost function calls. This implies that a scalability test running at a rate of 1,000 TPS on the main Test Controller would result in an overall system throughput of 30,000 total function executions. The current implementation of the metrics function using .NET Core now has hugely significant cost implications. At a rate of 1,000 TPS

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

on the main Test Controller function, based on the performance and costs measured as part of this research, CostHat calculates an overall cost of \$31,463 per day. If the Metrics function was to be changed to NodeJS, as in the previous example, this reduces significantly to \$8,716 per day. This represents a reduction in cost of about 72%.

Chapter 5

TEST RESULTS - AZURE FUNCTIONS

The aim of this thesis, as described in Chapter 1, is to evaluate the performance and cost implications of the choice of language runtime for serverless function implementation. This includes the assessment of two Azure Functions runtimes: NodeJS and C# .NET. The test approach and implementation was discussed in detail in Chapter 3. Chapter 4 discussed separately the results of AWS Lambda testing and the application of the calculated costs to a CostHat model (Leitner et al. 2016) to understand cost implications for a full serverless architecture.

This chapter will discuss the tests performed on the Microsoft Azure Functions and describe the results produced. A comparison is also provided against results for similar runtimes in AWS Lambda. Further discussion is then provided in Chapter 6. Test result data was analysed using the same tools as described for AWS Lambda in Chapter 4.

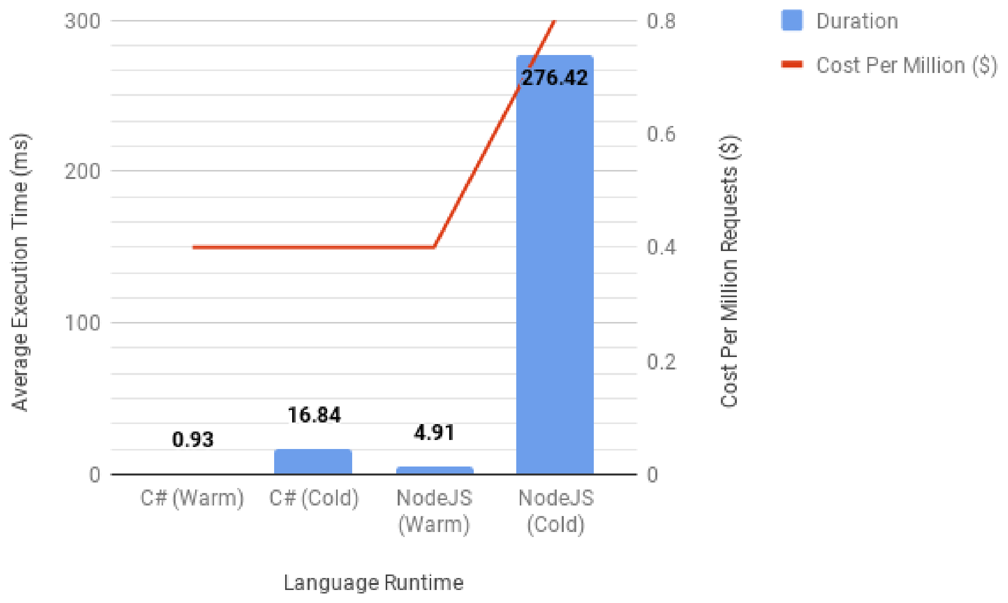


Figure 5.1: Performance and Cost For Azure Functions in Warm and Cold Start Scenarios

This round of testing was similar in nature to the AWS Lambda testing. The purpose was to measure Azure Functions performance in setting up the internal container environment to run a target function. This was again achieved by creating completely empty test functions in two of the main supported Azure Functions languages: 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. See section 3.5 for example of the CLI command used.

Testing was performed in the same two scenarios as the AWS Lambda testing presented in Chapter 4. Cold-Start tests evaluated Azure Functions performance when it needed to allocate a fresh container to execute the function. These were performed at the same 1-hour intervals as in AWS testing. There were a total of 144 Cold-Start tests (for each runtime) over a 6-day period. Warm-Start tests were performed at 1-minute intervals where an existing container would be re-used by the Azure Functions platform. There were a total of 273 Warm-Start tests over a single 4.5-hour period for each of the two Azure runtimes tested. See Chapter 3 for details of how the test environment was created.

5.1 Azure Test Results Analysis

Figure 5.1 shows the combined average performance and cost of both cold and warm-start scenarios for NodeJS and C# Functions in Azure. C# performs much the faster of the two runtimes measure in Azure, showing an *average* sub-millisecond performance of 0.93ms in warm-start scenario. This compares to 4.91ms for NodeJS. From a cost perspective, both are equivalent due to the 100ms billing increments charged by Azure Functions.

In the Cold-Start scenario, C# again performs better than NodeJS. In this case, however, it performs significantly better - 16.84ms average compared to 276.42ms. The relatively poor performance of NodeJS runtime in Azure in these cold-start scenarios has a significant cost implication. Functions, on average, implemented in NodeJS cost 200% of the C# function cost in the cold-start scenario ($0.80 per million requests compared to $0.40). The histogram displayed in Figure 5.2 provides more detail on the reasons for this disparity in performance between C# and NodeJS. The C# runtime shows considerable consistency in terms of execution time, 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

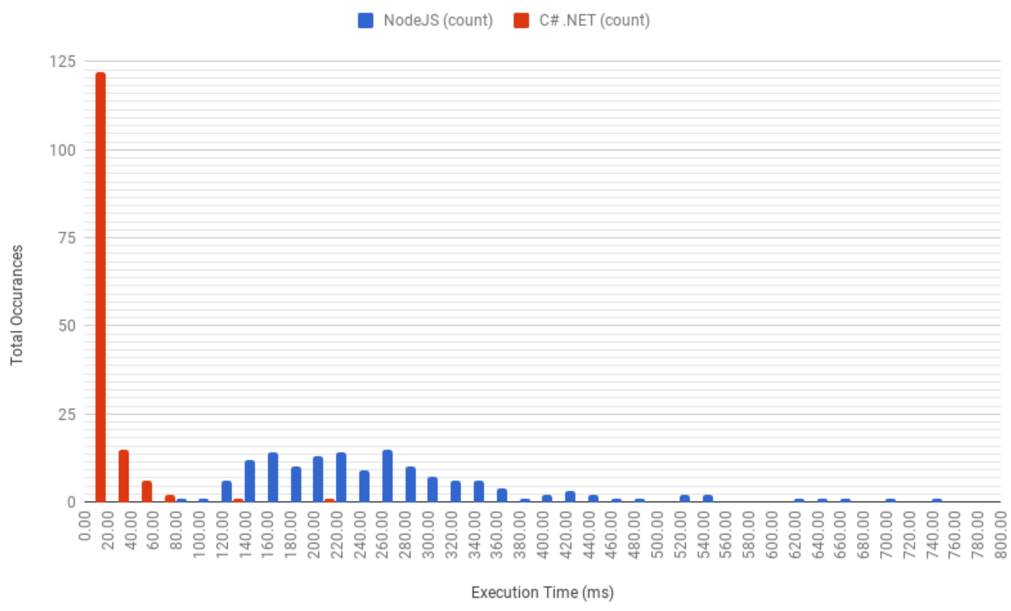


Figure 5.2: Histogram of Azure Functions Cold Start Performance

the 20ms buckets between 120ms and 340ms. Note that a single outlier for NodeJS of just over 1100ms was omitted from the histogram to aid clarity of the chart.

5.2 Cost Analysis

Azure Functions are billed similarly to AWS Lambda. The similarities include the fact they they are billed in GB-seconds and that billing is applied in 100ms increments. The difference is that unlike AWS Lambda, Azure Functions are not pre-assigned a memory allocation. Instead, they are dynamically assigned memory based on their execution needs. 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 assumed. Therefore, the minimum billing for a function is 0.1 seconds at 128MB of memory. Note that Azure Functions can be billed via a pay-per-use pricing model (labelled "consumption plan") and also via an "app service plan" model which is a more traditional style of billing based on pre-configured compute resources. This research assumes the "consumption" pay-per-use billing model which is more typical of the usual pay-per-use serverless billing style.

The cost calculations presented in Table 5.1 are based on all the performance data gathered in the Azure warm and cold-start testing described in the previous sections.

Language Runtime	Test Type	Average Execution Time (ms)	Average Billed Duration (ms)	Average Cost Per Function ($)	Average Cost Per Million ($)
.NET (C#)	Cold	16.84	100.00	0.0000004	0.4
NodeJS	Cold	276.42	300.00	0.0000008	0.8
.NET (C#)	Warm	0.93	100.00	0.0000004	0.4
NodeJS	Warm	4.91	100.00	0.0000004	0.4

Table 5.1: Performance of Azure Functions in Cold and Warm Start Tests Mapped to Cost

Language Runtime	Cost Per Day @ 10-TPS	Cost Per Day @ 100-TPS	Cost Per Day @ 30k-TPS	Cost Per Year @ 10-TPS	Cost Per Year @ 100-TPS	Cost Per Year @ 30k-TPS
.NET C#	$0.35	$3.45	$1,036.80	$126.14	$1,261	$378,432
NodeJS	$0.69	$6.91	$2,073.60	$252.29	$2,523	$756,864

Table 5.2: Cost of Cold-Start execution at Varying Throughput (TPS) for Azure Functions

Note that all functions were running within maximum of 128MB memory. Similar to AWS Lambda, there are three factors in the cost of a function execution: invocation cost, execution time and (maximum) memory allocated to the function. There is also an almost identical free-tier model in Azure[1], with 400k GB-seconds and 1 million invocations allowed for free for all accounts. This free-tier is *not* included in cost calculations, as was the case for AWS Lambda calculations also. Costs were calculated using latest Azure pricing of $0.20 per million function invocations and $0.000016 per GB/s of execution time. Note that the only current base-price difference between Azure and AWS Lamdba is the rounding of the GB/s execution time - AWS charges a more specific $0.00001667 per GB/s, which is a 4.2% higher cost per GB/s.

The data presented in Table 5.2 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. This is intended to indicate how the relative performances could *potentially* affect an application's cost at various throughput levels, up to an extreme example of 30k TPS. Overall, as described in the previous

[1]https://azure.microsoft.com/en-gb/pricing/details/functions/

Serverless Platform	Language Runtime	Warm Start Average (ms)	Cold Start Average (ms)
AWS Lambda	.NET Core 2 (C#)	6.32	2500.09
AWS Lambda	NodeJS	11.46	23.67
Azure Functions	.NET C#	0.93	16.84
Azure Functions	NodeJS	4.91	276.42

Table 5.3: Summary of Average Performance Between Azure and AWS

section, 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.

5.3 Azure vs AWS Results Comparison

This section describes a comparison between Azure Functions and AWS Lambda based on the two runtimes tested in Azure: NodeJS and .NET C#. Performance in both cold-start and warm-start scenarios is compared separately.

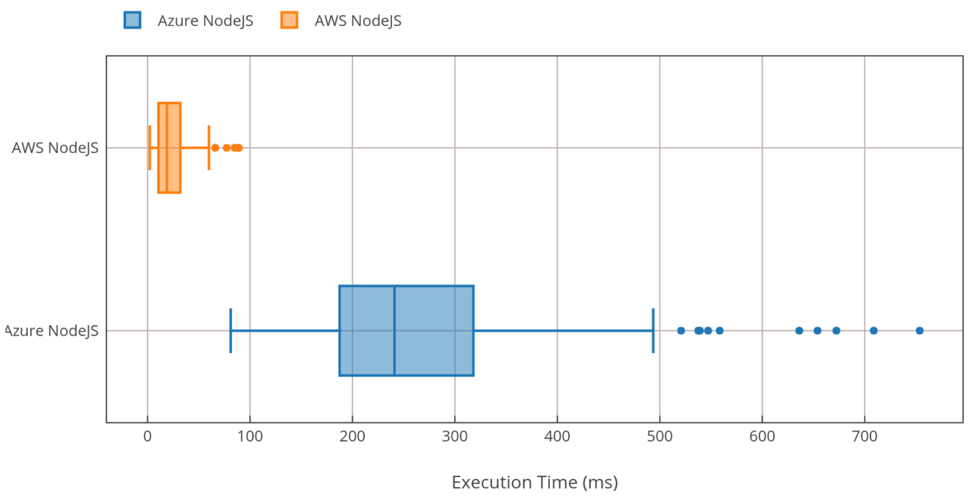


Figure 5.3: Box Plot of Cold Start Performance for NodeJS on Azure and AWS

A summary of the average execution times in each scenario is shown in Table 5.3. It shows that each serverless platform has advantages in different runtimes. 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 5.3

adds more detail as to the spread of test results in this cold-start scenario for NodeJS. It is interesting to note in the box-plot that the lower whisker (the lowest recorded value at 81.14ms) for Azure is actually almost as high as the upper whisker of AWS at 89.09ms. There is no comparison between the two NodeJS runtimes - 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 (although linux environment options are being introduced[2]), versus the linux-based containers in use on AWS[3].

While Azure NodeJS performance is over 10x slower than AWS, it is a lower 3x multiple in terms of billed duration (300ms against 100ms). Relative cost and average execution times can be seen in the combination chart shown in Figure 5.4. Average cold-start cost (per million requests) for NodeJS in Azure is $0.80 compared to $0.41 on AWS.

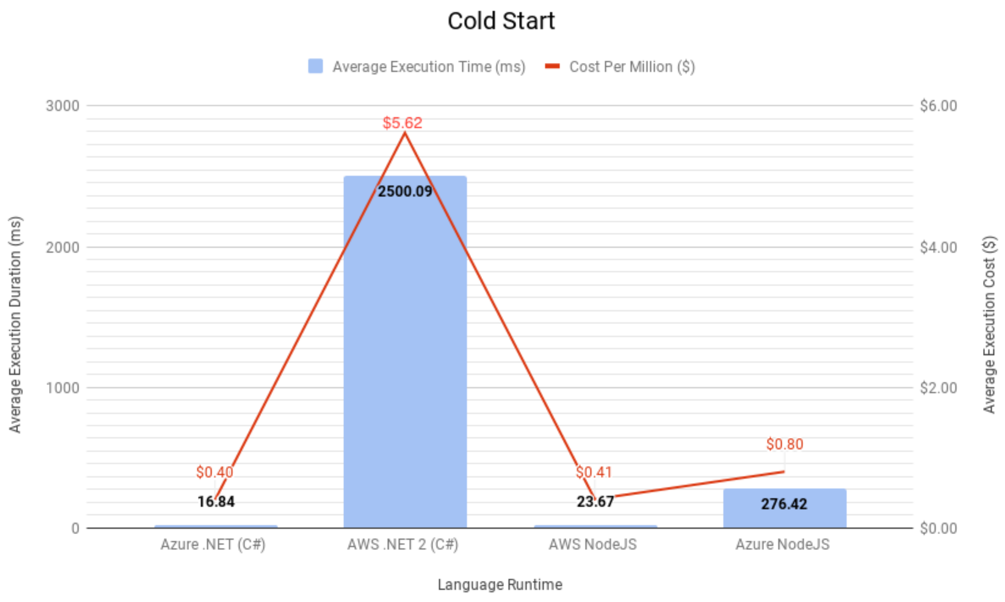


Figure 5.4: Comparison of Cold Start Performance and Cost Between Azure and AWS

For C# .NET performance, the situation is reversed. Table 5.3 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

[2]https://blogs.msdn.microsoft.com/appserviceteam/2017/11/15/functions-on-linux-preview/
[3]https://docs.aws.amazon.com/lambda/latest/dg/current-supported-versions.html

average of just 6.32ms per execution and both platforms are equivalent from a cost perspective. However, the poor cold-start performance of AWS Lambda compares badly with Azure in both performance and cost. This can be seen in Figure 5.4. While the costs of warm-start functions are roughly equivalent across both platforms, the cold-start figures are quite different: AWS averages at $5.62 compared with $0.40 on Azure (per million requests).

For warm-start tests in C#, the box plot in Figure 5.5 shows a comparison of performance between Azure and AWS. It shows interesting results where Azure significantly out-performs AWS. This is perhaps not unexpected. 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 standalone c-sharp "script" (.CSX extension) files running on windows containers. This is different to the AWS support of .NET Core 2 which is based on traditional .NET project structure and runs on the open-source .NET CLR (Common Language Runtime) on linux containers.

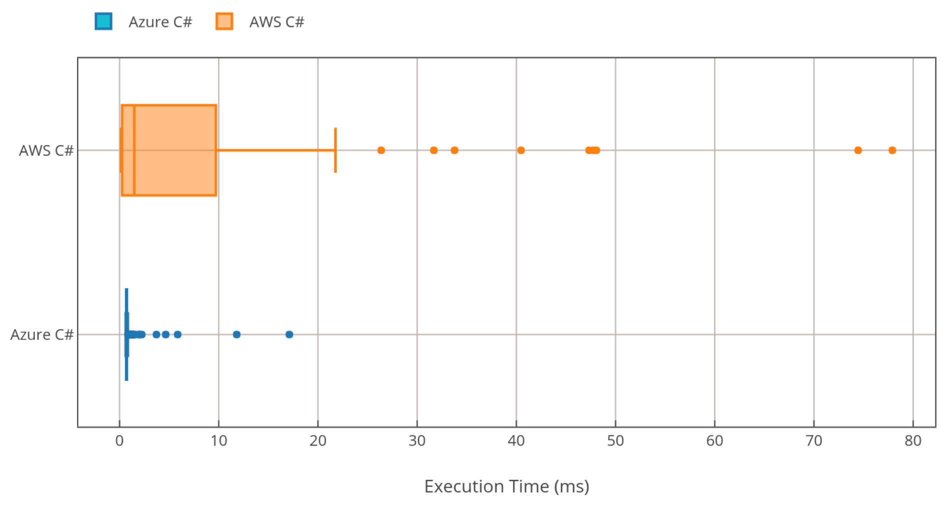


Figure 5.5: Comparison of Warm-Start C# Function Performance Between Azure and AWS

Azure shows remarkably fast and importantly consistent execution of C# functions. As the box plot shows, there are few outliers outside the sub-millisecond performance shown. AWS performance is still largely very fast in warm-start conditions, with 75% of all tests were in the sub-10 millisecond range. From a billing perspective, they are the same, however, as both are consistently well within the 100ms billing increment.

Chapter 6

DISCUSSION

This chapter discusses the highlights of the findings presented in Chapters 4 and 5 and their implications.

The tests performed in this research showed 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, C# .NET is the top performer for Azure Functions. Similarly, Python is a clear choice on AWS Lambda, 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.

Choosing the appropriate language runtime is crucial. Factors influencing this decision include the expected throughput of the function and the target cloud platform. The results obtained for some runtimes vary significantly based on the initialisation scenario (e.g. cold vs. warm start). C# .NET is a particular example. On AWS Lambda, .NET functions showed completely contrasting performance characteristics based on whether the internal container was already initialised (warm-start) or not (cold-start). The cold-start performance was so slow that use of .NET should be avoided on AWS Lambda unless it is expected to experience a consistent and high throughput, thus avoiding too many cold-start executions. This is because its performance in a warm-start scenario was both fast and consistent. On the other tested serverless platform, Azure Functions, C# showed incredibly fast and very consistent performance in the sub-millisecond range (for warm-starts). The variety in results for .NET (C#) across the tested platforms and scenarios emphasises an important conclusion - the choice of language runtime and serverless platform can have a potentially huge effect on response times and hosting costs. This conclusion is not limited to the C# runtime. NodeJS also showed significant differences in performance and cost between AWS and Azure. However, in this case, AWS Lambda functions implemented in NodeJS out-performed similar Azure Functions in cold-start scenarios.

In warm-start testing, despite some languages being 2-3 times faster than others, the cost is uniform across the board. This reflects the advice of Eivy (2017) in relation to performance tuning discussed earlier in section 2.2. Due to the billing increments of 100ms, relative function performance does not always mean higher costs of execution. However, there are some caveats to these findings. This includes the fact that the tests performed in this research were all based on an empty function and the actual execution duration may be closer to 100ms depending on the task of a real-world function. While in general, the warm-start (and many of the cold-start) empty function tests are nowhere near the current 100ms billing increment, two things might affect this - changes in the billing model (quite possible in future given Lambda@Edge already bills in 50ms increments[1]) and the type of task the function is performing. If the average execution time might be pushed close to the billing increment line, this would cause a more material effect on cost. In this scenario, the difference in average execution times may cause significant cost differentials. For example, on average there is a 13.8ms warm-start difference for empty functions between Golang and Python on AWS Lambda. If a Python function implementing some real-world task were to average 90ms, a Golang function might take 103.8ms, costing double at a billed duration of 200ms. The pace of change in serverless computing is extremely high - both in features offered, usage patterns and cost models. This constantly shifting environment requires regular review to ensure serverless applications are designed for optimum performance and cost benefit.

One of the design principles behind the SPF framework created for this research was to enable full automation of test execution, metrics collection and also analysis of the data. The analysis would be through an expanded API allowing search and retrieval of performance testing results. This would allow the creation of an automated dashboard, with the potential for ongoing monitoring of serverless functions' performance. In a serverless architecture composed of very clearly defined, discrete functions, individual high-volume functions could be re-implemented easily if it was found in future (via the dashboard) that the performance of one language runtime had significantly improved relative to another - e.g. perhaps the cold-start issues of .NET C# on AWS Lambda might be resolved, making it a more attractive option. The conclusion here is that 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.

Overall from a cost perspective, the cold-start scenarios present the greatest dif-

[1]https://aws.amazon.com/lambda/pricing/#Lambda.40Edge_pricing_details

ferentials between the evaluated serverless platforms and language runtimes. The results presented in Chapters 4 and 5 showed cost calculations based on average execution times. These showed the effects on cost in cold-start scenarios 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). For an individual function, 1 million requests per day equates to a fairly moderate throughput of just over 10-TPS. However, as the extrapolated figures presented earlier in Table 4.4 show, higher throughput results in significantly higher costs for .NET and Java on AWS (and similarly for NodeJS on Azure Functions). The figure of \$5.5m (per year) for an extremely high-volume AWS C# function (compared to \$0.38m for Python, NodeJS and Go) is an extreme example, and perhaps would realistically contain many more warm-start times depending on the nature of the spikes in the traffic. However, considering an overall enterprise-level eco-system of many serverless functions (rather than a single function example), an overall combined throughput of 30k TPS is far more realistic and indeed a much higher rate of cold-start scenarios are probable. An example of such a system was presented via the CostHat (Leitner et al. 2016) model in section 4.6. This showed the increased cost caused by a downstream function implemented in .NET was an extra \$22,747 per day (or 361% of the cost if this function was implemented in NodeJS).

6.1 Limitations

This research measured both cold-start and warm-start performance of serverless function execution. To determine when a function is in a warm or cold start, research was conducted to understand the expected lifetime of a function container (as described in section 3.1). This provided the appropriate intervals in which to invoke a test function to achieve either warm or cold start scenario. This was set at 1-minute for warm-start and 1-hour for cold-start. This achieved satisfactory results, as the results in warm and cold-start test batches were consistent and in particular Azure functions were verified via manually executed Azure CLI commands to assess memory usage at specified times, showing a 0MB memory usage when in a "cold-start" state. However, deterministic detection of warm vs cold start would be ideal. There are references to methods of detecting whether a function was invoked from a warm or cold state. These include using the local file cache of the function execution environment (Wagner 2014). A check could be performed on the existence of a "fingerprint" file, the presence of which would indicate a warm-start of the function. Another method might use in-memory "counters" as described by Cui (2017*c*).

For cost calculations, the pricing information for both AWS and Azure were set as static environment variables on the cost-calculation Lambda function which performed that task (see section 3.3). This was a sufficient approach for the purposes of this research as the pricing model did not change in either cloud platform. Ideally, however, pricing base units (e.g. cost-per-GB-second or cost-per-invocation) would be updated automatically as pricing changes were made. This is possible with AWS (Barr 2017) based on the SNS (Simple Notification Service) topic they provide for price update notifications. This provides a JSON structure of all updated prices by AWS Region, allowing cost-calculations to be dynamically (and perhaps retrospectively if necessary) updated.

In AWS Lambda, there are five different runtimes available for implementation of a function. These include NodeJS and Python, for which there are actually multiple versions available. At the beginning of this research, NodeJS had two versions (4.3 and 6.10). During the course of the research, a third version (8.10) became available (see section 3.4). For simplicity, only one version (6.10) was chosen for the study as being the latest version available (at that time). Similarly for Python, both 2.7 and 3.6 were available. 3.6 was the version used as being the latest version of the newer "branch" of python. However, subsequently usage data provided by NewRelic (Smith 2017) was reviewed, indicating that Python2.7 accounted for 48.9% of AWS Lambda functions and therefore might have proven a more useful target runtime for study. That said, it will be a minor effort in future to add Python2.7 testing support into the framework.

6.2 Future Work

The research conducted for this thesis has revealed many potential avenues of further investigation. Some of these are as a result of time constraints on this research while others are potential further avenues of investigation prompted by the results achieved.

This thesis limited itself in scope to testing empty functions in order to concentrate on the performance of a serverless platform in creating the execution environment for each language runtime. Other tests could be considered in future, such as the common use-case of database access (such as DynamoDB in AWS) given the fact that serverless functions are stateless in nature. Additionally, standard programming language performance benchmarking tests could reveal useful aspects of runtime performance. For example, in performing language performance analysis between Java and Go, Togashi & Klyuev (2014) used a technique of matrix-based multiplication, applied in single and multi-threaded scenarios. Additionally, Lei et al. (2014) measured relative performance as a web framework of NodeJS, PHP and Python by testing two scenarios - an essentially empty "hello world" web endpoint and a Fibonacci sequence calculation.

The Serverless Performance Framework (SPF) created for this thesis has been designed to allow any number of serverless platforms to be "plugged-in", in order to allow cross-platform comparisons of performance and cost. In this research, AWS Lambda and Azure Functions were tested. These were chosen due to their respective popularity and market-leading positions. However, there are a number of other serverless platforms already available or emerging. These include major cloud platforms such as IBM OpenWhisk and Google Cloud Functions. There are also open-source projects in development such as OpenLambda (Hendrickson et al. 2016) and Iron.io (Lynn et al. 2017). Plugging these platforms into the test framework via the existing API layer would allow for further cross-platform comparisons.

An intriguing aspect of the results of this research centres around the results for AWS Lambda's implementation of Golang and Python. Unexpectedly, they showed better performance when a function was called in a cold-start scenario (i.e. once per hour) than in a warm-start scenario (once per minute invocation). It would be of interest to investigate this further to understand whether this was an anomaly caused by insufficiently large data sets or whether there was a consistent underlying cause.

This research measured serverless function performance in two distinct scenarios: cold-start and warm-start. The *"cold-start effect"* has potentially significant impact on the running costs of serverless-based applications. There are techniques that have been created, such as that described by Neves (2017) to reduce this effect by keeping functions "warm" through scheduled "heartbeat" invocations. This would be an interesting area of research to understand the optimum frequency, or indeed create the ability to dynamically adjust the frequency of such "heartbeats".

Finally, there is significant potential in measuring the performance and cost of serverless platforms against other cloud computing architectures such as basic VMs (Virtual Machines) and containers. Docker and Kubernetes are emerging as industry standard container technologies, reflected in their support by all the major cloud platforms, including AWS, IBM, Azure and Google. Cloud providers are rapidly creating new docker-based services. For example, AWS announced in 2018 native support for Kubernetes via their EKS (Elastic Kubernetes Service) service in addition to their new "Fargate" service (Hunt 2017) which provides a CaaS (Container-as-a-Service) model. On Azure, there is a similar service, "Azure Container Instances", which offers another CaaS option[2].

[2]https://azure.microsoft.com/en-us/services/container-instances/

BIBLIOGRAPHY

Adzic, G. & 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. & Suter, P. (2017), 'Serverless computing: Current trends and open problems', *CoRR* .
URL: *http://arxiv.org/abs/1706.03178*

Barr, J. (2017), 'Aws price list api update – regional price lists', `https://aws.amazon.com/blogs/aws/aws-price-list-api-update-regional-price-lists/`. Accessed: 2018-04-17.

Conning, J. (2017), 'Aws lambda: Faster is cheaper', `https://medium.com/@jconning/aws-lambda-faster-is-cheaper-6bf32f58d741`. Accessed: 2018-03-13.

Conway, S. (2017), 'Cloud native technologies are scaling production applications', `https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/`. Accessed: 2018-05-04.

Crane, M. & Lin, J. (2017), 'An exploration of serverless architectures for information retrieval'.

Cui, Y. (2017*a*), 'Comparing aws lambda performance when using node.js, java, c# or python', `https://read.acloud.guru/comparing-aws-lambda-performance-when-using-node-js-java-c-or-python-281bef2c740f`. Accessed: 2018-03-13.

Cui, Y. (2017*b*), 'How does language, memory and package size affect cold starts of aws lambda?', `https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76`. Accessed: 2018-03-13.

Cui, Y. (2017*c*), 'How long does aws lambda keep your idle functions around before a cold start?', `https://read.acloud.guru/how-long-does-aws-`

lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810. Accessed: 2018-04-17.

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

Fielding, R. T. & Taylor, R. N. (2000), *Architectural styles and the design of network-based software architectures*, Vol. 7, University of California, Irvine Doctoral dissertation.

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

Fox, G. C., Ishakian, V., Muthusamy, V. & 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. & Arpaci-Dusseau, R. H. (2016), ‘Serverless computation with open-lambda’, *Elastic* p. 80.

Hoang, A. (2017), Analysis of microservices and serverless architecture for mobile application enablement, PhD thesis, California State University, Northridge.

Hunt, R. (2017), ‘Introducing aws fargate’, https://aws.amazon.com/blogs/aws/aws-fargate/. Accessed: 2017-12-03.

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997), Aspect-oriented programming, *in* ‘European conference on object-oriented programming’, Springer, pp. 220–242.

Lei, K., Ma, Y. & Tan, Z. (2014), Performance comparison and evaluation of web development technologies in php, python, and node.js, *in* ‘2014 IEEE 17th International Conference on Computational Science and Engineering’, pp. 661–668.

Leitner, P., Cito, J. & 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.

Lima, E. (2018), ‘Node.js 8.10 runtime now available in aws lambda’, `https://aws.amazon.com/blogs/compute/node-js-8-10-runtime-now-available-in-aws-lambda/`. Accessed: 2018-04-14.

Lin, Y. Z. (2018), ‘Comparing aws lambda performance of node.js, python, java, c# and go’, `https://read.acloud.guru/comparing-aws-lambda-performance-of-node-js-python-java-c-and-go-29c1163c2581`. Accessed: 2018-03-13.

Lynn, T., Rosati, P., Lejeune, A. & 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.

Malawski, M., Gajek, A., Zima, A., Balis, B. & Figiela, K. (2017), ‘Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions.’, *Future Generation Computer Systems* .

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

McGrath, G. & Brenner, P. R. (2017), Serverless computing: Design, implementation, and performance, *in* ‘Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on’, IEEE, pp. 405–410.

Meyerson, J. (2014), ‘The go programming language’, *IEEE Software* (5), 104–104.

Nasirifard, P., Slominski, A., Muthusamy, V., Ishakian, V. & Jacobsen, H.-A. (2017), A serverless topic-based and content-based pub/sub broker, *in* ‘Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos’, ACM, pp. 23–24.

Neves, G. (2017), ‘Keeping functions warm - how to fix aws lambda cold start issues’, `https://serverless.com/blog/keep-your-lambdas-warm/`. Accessed: 2018-04-17.

Smith, C. (2017), ‘Aws lambda in production: State of serverless report 2017’, `https://blog.newrelic.com/2017/11/21/aws-lambda-state-of-serverless/`. Accessed: 2018-04-14.

Togashi, N. & Klyuev, V. (2014), Concurrency in go and java: Performance analysis, *in* '2014 4th IEEE International Conference on Information Science and Technology', pp. 213–216.

Tukey, J. W. (1977), *Exploratory data analysis*, Vol. 2, Reading, Mass.

Varghese, B. & 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. & 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.

Wagner, T. (2014), 'Understanding container reuse in aws lambda', `https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/`. Accessed: 2018-04-17.

Wampler, D. & Clark, T. (2010), 'Guest editors' introduction: Multiparadigm programming', *IEEE Software* (5), 20–24.

Appendix A

SOFTWARE PACKAGES AND TOOLS

The list of software, packages and tools presented in Table A.1 is a comprehensive list of what was used to create the Serverless Performance Framework used to execute, store and analyse results for this thesis. This appendix lists the steps to setup a development environment (assuming Mac OS X) to reproduce the test environment. Detailed commands and steps for the actual deployment of the functions themselves are included on the GitHub repository[1].

A.1 Pre-Requisites

A valid AWS (Amazon Web Services) account and an Azure Portal account are required for testing against the respective serverless platforms.

A.2 AWS Test Environment Setup Instructions

The following setup steps assume running on Mac OS X. See Table A.1 for version details. Note that these steps are required to enable build and deploy of both the AWS Test Target functions but also the common components for collecting and storing the metrics and costs data, such as the API, Metrics Function and Cost Function (see details in Chapter 3).

1. Install Brew (a package manager for MAC OS)
2. Install Node (via *"brew install node"*)
3. Install AWS CLI
4. Configure AWS Credentials for AWS CLI
5. Install Serverless Framework (via *npm install -g serverless*)
6. Configure AWS Credentials for Serverless Framework
7. Install .NET Core 2.0.5
8. Install Java JDK 1.8

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

Package	Version	URL
MacOS	10.12.6	
Brew	1.5.4	`https://brew.sh`
AWS CLI	1.14.32	`https://aws.amazon.com/cli`
Serverless Framework	1.26.1	`https://serverless.com/framework/docs/getting-started/`
Node	9.5.0	`https://nodejs.org/en/`
NPM	5.6.0	`https://www.npmjs.com`
.NET Core Framework	2.0.5	`https://www.microsoft.com/net/learn/get-started/macos`
.NET SDK / CLI	2.1.4	`https://www.microsoft.com/net/learn/get-started/macos`
Java	Oracle jdk1.8.0_101	`http://www.oracle.com/technetwork/java/javaee/overview/index.html`
Apache Maven	3.5.2	`https://maven.apache.org/`
Golang	1.10	`https://golang.org/doc/install`
Azure CLI	2.0.29	`https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest`
Azure Functions Core Tools	2.0.1 beta.24	`https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurefunctions`
Plot.ly	N/A	`https://plot.ly`
Google Sheets	N/A	`https://sheets.google.com/`

Table A.1: List of Software Packages Used

9. Install Maven 3.x

10. Install Golang (1.x)

11. Note the *"LatestStreamArn"* for the "metrics" table generated by the first dynamo-db create CLI command in step below. This is needed when specifying the trigger for the cost-metrics calculation function.

12. Deploy DynamoDB tables used by this framework for persisting data

    a) Create Metrics Table

    ```
    aws dynamodb create-table --cli-input-json
    file://create-table-metrics.json --region us-east-1
    --profile <aws cli profile> --stream-specification
    StreamEnabled=true,StreamViewType=NEW_IMAGE
    ```

    b) Create Costs Table

```
aws dynamodb create-table --cli-input-json
file://create-table-costs.json --region us-east-1
--profile <aws cli profile>
```

A.3 Azure Functions Test Environment Setup Instructions

The following setup steps assume running on Mac OS X. See Table A.1 for version details. Note that these steps are required to enable build and deploy of Azure Test Target functions (see details in Chapter 3).

1. Install Azure CLI (via *"brew update && brew install azure-cli"*)
2. Install Azure Serverless Framework Plugin via *"npm install -g serverless-azure-functions"*
3. Install VSCode Azure Functions Plugin
4. Install Azure Core Tools via *"npm install -g azure-functions-core-tools@core –unsafe-perm true"*
5. Follow instructions to setup Azure CLI credentials[2] to use Serverless Framework for deployment.
6. Follow instructions to setup Serverless Framework for Azure[3].

[2]https://serverless.com/framework/docs/providers/azure/guide/credentials/
[3]https://serverless.com/framework/docs/providers/azure/guide/quick-start/

Appendix B

SAMPLE TEST FUNCTION IMPLEMENTATION

This appendix contains the full listing of an example test target function from each of the two serverless platforms tested as part of this research: AWS Lambda and Azure Functions. For a full description of the implementation of the Serverless Performance Framework (SPF), see Chapter 3. All testing in this research was against completely empty functions. The purpose of this was to measure the performance (and cost) of the serverless platform to construct the environment for execution of the target function. Full code is available on GitHub[1].

B.1 AWS Lambda - NodeJS Empty Function Example

handler.js:

```
'use strict';

module.exports.emptytestnode810 = (event, context, callback) => {

  // just an empty function
  callback(null, {});
};
```

B.2 Azure Functions - C# Empty Function Example

run.csx:

```
using System;

public static void Run(TimerInfo myTimer, TraceWriter log)

{

// Completely Empty Function

}
```

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

Appendix C

COST HAT MODEL

This appendix contains the details of the Cost Hat model (Leitner et al. 2016) used for creation of cost modelling of microservice architectures based on the performance and cost calculations generated as part of this research. The full code listing is available on GitHub[1].

C.1 Single Function Sample CostHat Model Definition

This section contains the Python definition of the CostHat model used to demonstrate costs of a single test function in the Serverless Performance Framework created for this research. See Figure 4.10 for a graphical representation of this model.

```
# Overall costs listed based on "per million requests"
def test_aws_spf_coldstart():

  test_netcore = LambdaEndpoint('test_netcore')
  test_netcore_service = LambdaService
   ('test_netcore_service', [test_netcore])
  test_netcore_costs = {'capi' : 0, 'cio' : 0, 'ccmp' : 5.62, "coth" : 0}
  test_netcore.configure_endpoint(test_netcore_costs)

  aws_logger = LambdaEndpoint('aws_logger')
  aws_logger_service = LambdaService('aws_logger_service', [aws_logger])
  aws_logger_costs = {'capi' : 0, 'cio' : 0, 'ccmp' : 22.7, "coth" : 0}
  aws_logger.configure_endpoint(aws_logger_costs)

  # API Calls \$3.50 per million requests + data transfer out
  # (\$0.09/GB for first 10TB)
  # Traffic is within 1GB, so \$0.09 total for transfer and \$3.50
  # for the API calls
  common_metrics = LambdaEndpoint('common_metrics')
  common_metrics_service = LambdaService
```

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

```
 ('common_metrics_service', [common_metrics])
common_metrics_costs =
 {'capi' : 3.59, 'cio' : 0.47, 'ccmp' : 20.83, "coth" : 0}
common_metrics.configure_endpoint
 (common_metrics_costs)

# IO costs based on DynamoDB cost \@ \$0.47 per month
# for 1 WCU (enough for 1 write per second or 2.5m per month)
# This is the minimum and is enough to cover the 1 million
# requests being priced
# https://aws.amazon.com/dynamodb/pricing/
common_cost_metrics = LambdaEndpoint
 ('common_cost_metrics')
common_cost_metrics_service = LambdaService
 ('common_cost_metrics_service', [common_cost_metrics])
common_cost_metrics_costs =
 {'capi' : 0, 'cio' : 0.47, 'ccmp' : 2.49, "coth" : 0}
common_cost_metrics.configure_endpoint
 (common_cost_metrics_costs)

test_netcore_cg = [(aws_logger_service, aws_logger, 1)]
test_netcore.set_callgraph(test_netcore_cg)

aws_logger_cg =
 [(common_metrics_service, common_metrics, 1)]
aws_logger.set_callgraph(aws_logger_cg)

common_metrics_cg =
 [(common_cost_metrics_service, common_cost_metrics, 1)]
common_metrics.set_callgraph(common_metrics_cg)

model = CosthatModel([test_netcore_service, aws_logger_service,
 common_metrics_service, common_cost_metrics_service])
```