Honours Project (CCIS)

Final Report

2018-2019

Submitted for the Degree of BSc Computing

Project Title: Amazon Lambda

Name: Christopher Connor
Programme: BSc Computing

Matriculation Number: S1715477

Project Supervisor: Katrin Hartmann

Second Marker: Huaglory Tianfield

Word count :

"Except where explicitly stated, all work in this report, including the appendices, is my own original work and has not been submitted elsewhere in fulfilment of the requirement of this or any other award"

Signed by Student: _____ **Date: 18th April 2019**

Abstract

Serverless computing is the latest evolution of cloud computing architecture and is seeing increased popularity and adoption. Popularised by Amazon in 2014 with the launch of their serverless compute service Lambda – the serverless paradigm has garnered much attention amongst developers and architects alike.

Expanding on the popular microservice architecture pattern adopted by many large companies, serverless takes this idea even further by extracting application logic out into single function calls. These functions are then triggered by an event source – be that an HTTP request from an API end-point or an image upload to an Amazon S3 bucket.

With new technology comes new problems. Due to the underlying architecture of the platform, and its use of containerisation – Lambdas can exhibit a "cold start" after a period of inactivity which presents in the form of a delayed response to the calling event. Initial research showed that this cold start effect could be exacerbated by the choice in runtime. The aim of this project therefore was to develop and test an API using the three most popular runtimes used in Amazon Lambda in order to analyse the performance of each in relation to the cold start problem.

Results showed a demonstrable link between the choice of runtime and its impact on the cold start problem, with interpreted languages generally launching much quicker. However, results also showed that the allocation of memory to each function also has a direct impact on the duration of the cold start but also the execution of warm lambdas.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Katrin Hartmann, for her help and support throughout my project. Your engagement, insights and guidance are greatly appreciated.

I would also like to thank my wife, Michelle, who has been an endless source of encouragement, reassurance and support since my decision to return to higher education in 2015.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

This introductory section aims to provide a brief history on cloud computing and the evolution which has led to serverless computing models.

1.1 Project Background

Cloud computing infrastructure has revolutionised the way in which companies are designing, developing and deploying their software applications. It is thought that by 2021, cloud data centres will be responsible for processing 94% of all workloads and compute instances (*Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper - Cisco*, 2018). This massive shift has been fuelled by innovations in hardware, software and technology. Companies are under increasing pressure to deliver highly available, highly responsive online services to their customers. To do this, they must ensure they are themselves leveraging on the best services and infrastructure available.

Prior to cloud computing, companies had few options and generally relied on their own on-premise server infrastructure. Physical servers were expensive to procure and took a great deal of time and resources to setup. Dedicated, in-house teams were often required to maintain these servers and the underlying network infrastructures. As virtualization become more prevalent, companies could better utilise existing infrastructure by running multiple virtual machines on any given hardware.

1.1.1 Evolution of Software Architecture

Monolithic application architecture is a very common way to develop web applications. Software is written in classes and / or modules, all of which integrate together in a single codebase to produce the working application (Chen, Li and Li, 2017). When using the monolith pattern, all layers of the application - the user-interface, the business logic and the data access layer - will exist on the same environment as shown on Figure 1. Monolithic applications work well and are a good fit for many use cases. However, as applications begin to grow with each release and iteration – the monolithic codebase will also grow and begin to experience problems. As the code increases in complexity, it can become tightly coupled which, in turn, greatly increases the time it takes to test and deploy new features. In addition, large monoliths become gradually harder to scale as the entire monolith must be scaled - horizontal scaling involves additional resource costs as well as adding complexity to the infrastructure with load balancers and redundancy.

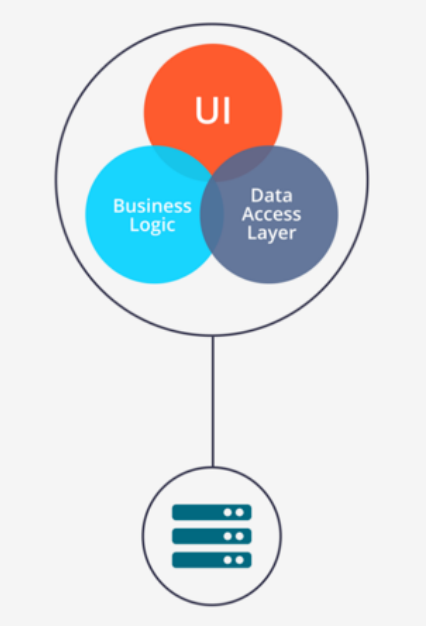


Figure 1 Monolithic Architecture (Bhagwati, 2017)

The microservice architecture is considered by many to be the perfect solution to address the restrictions of the monolithic architecture and has been adopted by many large companies like Amazon and Netflix (Waseem and Liang, 2017).

Microservices involves taking an oversized monolithic code base and decomposing the application down into smaller, self-sufficient applications as shown in Figure 2. Each microservice should have a sole responsibility and should be independently deployable (Newman, 2016). Extracting code into smaller, more manageable codebases means that the infrastructure becomes more flexible and greatly reduces coupling. Services communicate with each other via API calls thus eliminating any direct code dependencies between services. Furthermore, services are not restricted to any one given language or platform, giving developers and architects the freedom to choose the best programming language or environment for the task at hand. Working on and deploying a single, smaller codebase has many benefits – iterations can be fast, with many teams working in two to four-week sprints – however it also means that each service is individually scalable. If there were to be a traffic spike for a particular service, this alone could be scaled as opposed to scaling the entire application infrastructure.

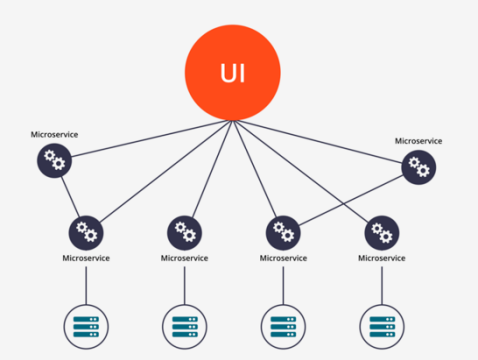


Figure 2 Microservice Architecture (Bhagwati, 2017)

1.1.2 Cloud Technologies and Models

This evolution in application architecture has been facilitated by the advancement in cloud computing technologies, as well as the new service and cost models cloud providers are now offering. Infrastructure as a Service (Iaas) and Platform as a Service (Paas) being two of the most popular.

IaaS offers users complete control over much of the infrastructure provided by the cloud provider. In this model, users are responsible for managing the operating systems, any middleware and their applications and runtime environment. Suited to larger businesses and those who require complete autonomy over their infrastructure – IaaS allows users to choose their operating systems, server environments as well as any library or dependency choices. Scaling is relatively easy but does involve the further provision of additional servers which will in turn increase costs.

PaaS, in contrast, is more limited. With PaaS, cloud companies generally provide a base environment from which to provision, configure and deploy their applications. This is the perfect solution for users who do not wish to have the overhead of dealing with servers at the operating system level – they merely want to provision some resources and deploy the application. Conversely, it may not be the best solution for users who don't wish to be locked down to a specific environment, language or provider.

Serverless architecture is one of the most recent innovations in cloud computing and is being offered by all of the top cloud providers – the most popular being Amazon Lambda. Serverless computing is somewhat of a misnomer, in that there are servers involved in this architecture, but their existence and details are removed from the user experience (Eivy, 2017). One of the main focuses of serverless computing is to abstract this layer from software developers, with the aim of allowing them to concentrate more on code and concern themselves less with the provision of servers and deployment (Lynn *et al.*, 2017). Now commonly referred to as Functions as a Service (FaaS), serverless computing aims to take the concept of microservice even further by decomposing logic into single, self-contained event-

driven function calls (Castro *et al.*, 2017). Depending on the platform, functions can be connected to a wide variety of event-triggers including API end-points, web-hooks, scheduling as well as a post-processing agent. Unlike the existing models where users are charged whilst the instances are running – whether they are in use or not – with the serverless model, charges are only made for compute time and not idle time. One of the biggest attractions however, with regards to serverless computing, is its apparent ability to automatically scale to meet the load requirements of any function at any given time. If a particular function is receiving a large number of requests of traffic, the function will be scaled up to handle these requests and will in turn scale back down again, and even be spun down completely when not in use. There are some performance restrictions with Amazon Lambda, including a maximum execution time of 900 seconds, as well as fixed tier memory – 128 MB to 3008MB, in 64MB increments (*AWS Lambda Limits*, 2018) - that can be allocated to any given function.

From preliminary research carried out into serverless architectures, one consistently reported drawback with Amazon Lambda functions – and indeed all main serverless platforms – occurs when launching from a "Cold Start" (Smith 2017). If a function hasn't been executed for some time, the Amazon Lambda service will completely remove the container on which the function was ran. As a result, the next time a request is received, there can be a marked difference in response times. This issue has become a widely discussed subject in the cloud computing community and will become a significant area of exploration for this project.

1.2 Project Overview

This section of the report will outline the project itself; it will briefly summarise the topics which the literature and technology review intends to investigate as well as an outline of the subsequent primary research tasks which will be carried out in an attempt to answer the research question.

1.2.1 Project Outline

As the internet continues to evolve at an unfathomable rate, businesses grow and develop to accommodate the ever-increasing demand and expectations of their clients. Software led businesses must remain ahead of this curve in order to stay at the top of their respective markets. As mentioned in the introduction, the shift to microservices has led to a reassessment of current cloud models and best practice across the industry.

The serverless paradigm claims to provide a solution to all these problems by abstracting the provision and management of servers away from developers. In addition, instead of paying for an always-on server instance, regardless of traffic – serverless claims to offer cost savings as a result of only paying for the computer time which is actually consumed (*AWS Lambda*, 2017). Finally, automatic scaling means that - regardless of traffic and load on your application - serverless will provide and manage the required resources to ensure a seamless service to users.

Whilst the serverless paradigm has provided a new way to develop and deploy applications and services at scale, so too has it brought new problems – most notably the cold start issue. A cold start occurs when there is no pre-existing container provisioned for a function to run on. As a result, a container must be spun up and the function loaded into the container before it can be executed. This process takes time; as a result, functions starting from a cold start often demonstrate a noticeable delay to the calling service or user on first run. Subsequent

requests received within a fairly short timeframe will execute much faster as this process of provisioning a container can be bypassed thus running the function seemingly instantaneously. As the platform matures and additional runtimes are added, this affords users flexibility in the services they are building. Unfortunately, not all languages are created equal and may not perform in the same way across the board which poses the question:

Does choice of runtime environment have an impact on serverless function cold starts on Amazon Lambda?

1.2.2 Project Aims and Objectives

The aim of this project is to explore the cold start problem which exists in serverless application platforms – namely Amazon Lambda. It will aim to ascertain whether or not runtime environment has an impact on the cold start issue by developing a serverless RESTful API using the most popular languages currently: JavaScript (Node.js), Python and Java (*The State of the Octoverse, GitHub*, 2018). The API will be developed using the three aforementioned languages which will subsequently be deployed as Amazon Lambda functions on the Amazon Web Services platform. Each will be evaluated using a set of semi-automated tests which will be outlined in subsequent sections. Once sufficient performance data has been gathered, it will be analysed in order to present information which will attempt to answer the research question as well as making recommendations for the future.

1.2.3 Examination of the Serverless paradigm

The serverless paradigm will be explored in greater detail. It will outline the key strengths of the architecture and provide a deeper understanding of the Amazon Lambda platform in general.

1.2.4 Investigation into the cold start problem

The cold start problem will be examined in detail to further ascertain the root cause and impact. By exploring and evaluating existing studies and literature, it is hoped that a clear picture of the issue will be gleaned in order to take the research further in order to develop, test and present potential solutions to the problem.

1.2.5 Investigate Amazon Lambda Runtime Environments

Amazon Lambda currently supports a number of runtime environments which may be used for development of serverless applications and services. An investigation will be carried out to evaluate the three most popular runtimes and detail their qualities as well as identifying any potential issues with regards to performance or utilisation within a serverless environment.

2 Literature and Technology Review

The literature and technology review will make up a significant part of the report. It will serve as a solid foundation which this project can build upon. By the end of this section, the reader should have a clearer idea of the project, its related areas and the existing work which has been carried out to date.

2.1 Examination of the Serverless paradigm

Since its launch in 2014 (Janakiram, 2014), Amazon Lambda has proved to be the catalyst for a cloud computing transformation in the form of serverless architecture. Although the ideas and concepts behind serverless architecture are not entirely new (Bardsley, Ryan and Howard, 2018), the Amazon Web Services brand and indeed Amazon Lambda have become synonymous with the technology.

2.1.1 Serverless Benefits

2.1.1.1 No servers

Without doubt, the biggest draw with regards to serverless is the idea of there being no servers to manage. Serverless provides developers with an extremely simplified model which all but completely abstracts the majority of operational concerns with regards to the provision and management of servers (Ishakian, Muthusamy and Slominski, 2018a). The main benefit of this is that it allows developers to focus more time on the business logic and developing quality software solutions (Adzic and Chatley, 2017).

2.1.1.2 Cost implications

One of the main benefits often publicised by Cloud Providers is the reported cost savings involved by switching to serverless. Traditionally in IaaS models, charges are raised on the basis of an always-on service instance – even if your server is idle and processing zero requests or work load. Serverless, on the other hand, only incurs charges for the compute time actually consumed in 100ms blocks. In a recent study into the economic impact of serverless computing (Adzic and Chatley, 2017), Adzic et al. evaluated two case studies in which companies migrated their existing applications to a serverless architecture. Their evaluation of the first case study showed that although the system's users had increased by over 50%, on moving to a serverless architecture their costs dropped by just under 50%. Whilst cost is not something which will be considered in this project, it is important to highlight that this potential cost saving is a big incentive for businesses to consider a switch to serverless.

2.1.1.3 Scalability

Last but not least is serverless architecture's elasticity and intrinsic ability to auto-scale on demand without the need to configure load balancers or indeed spin up new service instances which will not be utilised effectively. In a study carried out by Ishakian et al. into the use of serverless architecture along with deep learning models, they noted that the platform seemed to scale well but especially so with large memory allocations (Ishakian, Muthusamy and Slominski, 2018b).

2.1.2 How Does it Work?

As mentioned previously, serverless architecture obviously makes use of servers at its core. Serverless architecture essentially employs container-based technology in order to deploy its functions, albeit on a more granular scale than, for instance, a typical Docker deployment. To date, containerisation has proved to be a popular way in which to deploy microservices at scale (Sill, 2016).

In its basic form, the process is as outlined on Figure 3. When a Lambda Function event is triggered - be it from an end user via an API Gateway endpoint, or another event source within AWS – the platform itself will spin up a new container; all necessary bootstrapping and dependencies for the function will be pulled into the container and when ready, the function will execute. Whilst this whole process happens on the fly and is managed completely by the platform itself; this is a simplistic presentation of the process and in reality, it does present other issues with regards to performance.

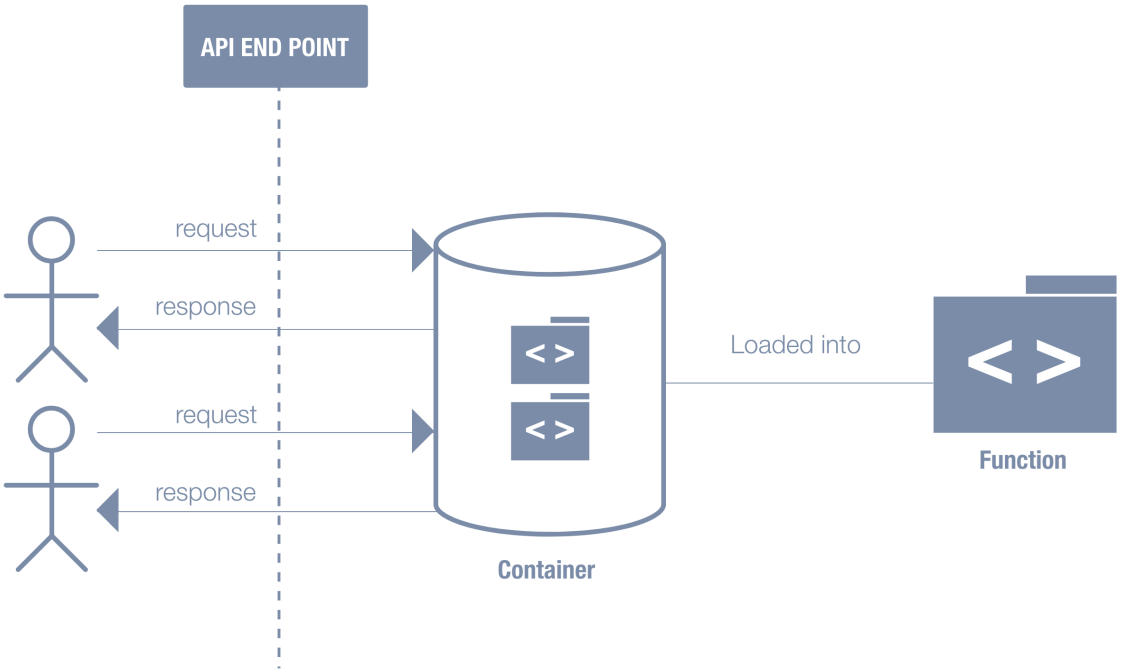


Figure 3 Rudimentary diagram of Lambda Infrastructure

2.2 Investigation into "cold start" problem

The cold start problem is a well-known and widely debated topic in the serverless community. To fully understand the issue, it is important to consider the entire process of how containers are provisioned. As mentioned previously, if an event is triggered a new container needs to be spun up. However, in the first instance, the platform checks to see if there is an existing container on which the function could be executed. If not, a completely new container will be created; again, all bootstrapping for the function will carried out and any dependencies will be pulled into the container and when ready, the function will execute. This is what is known as a cold start. If a container already exists, the function can be loaded and executed almost instantaneously – this is known as a warm start. Should more requests be received within a relatively short time frame the same container may well be re-used. Once there are no more requests, the AWS platform will eventually tear down the container so as to reserve resources and in-turn cost. In order to scale, this same process will be repeated any number of times whenever the AWS platform deems the need for further containers in order to meet the demand of the current load.

In a recent study into serverless architecture and its potential use alongside deep learning models, Ishakian et al. discovered that the cold start issue proved to be a significant factor in their research. They ran a series of tests on both warm and cold starts against various deep learning predication models, but they also took into the consideration the memory allocation for each function and the impact this had on both the latency of the function and the overall all prediction time of their deep learning models. As this project is only interested in the cold start impact, the following results are from a single deep learning model (SqueezeNet) however the model specifics are not required for the purposes of this evaluation.

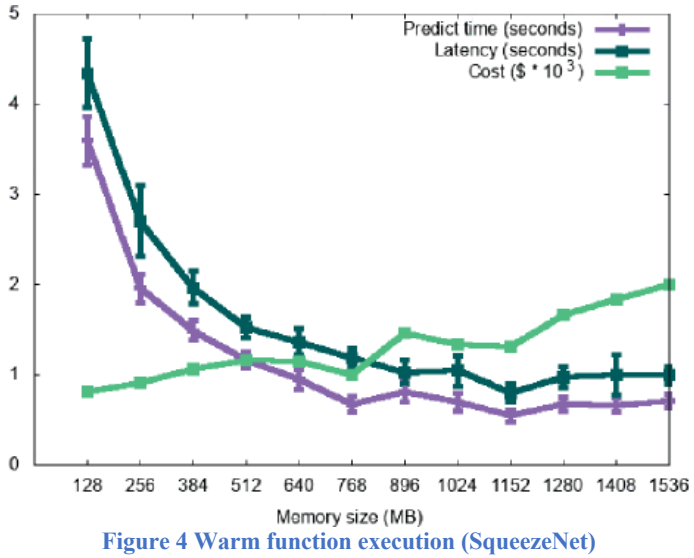


Figure 4 Warm function execution (SqueezeNet)

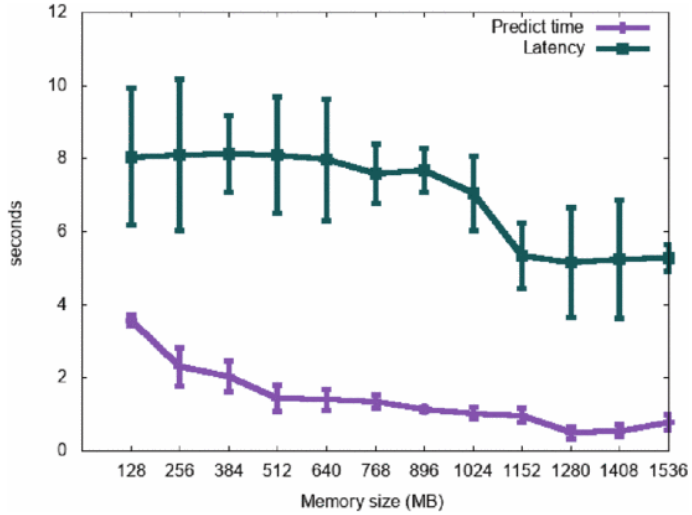


Figure 5 Cold function execution (SqueezeNet)

From the warm tests as show in Figure 4, they discovered that, as memory is increased in the functions, it decreases both the delay time and also the prediction time. In the cold start tests as shown in Figure 5, they noted that whilst the cold start time did begin to decrease as the memory was increased, "it doesn't not follow a similar pattern to that of warm starts".

Another more recent study was carried out by Bardsley et al. earlier this year, exclusively aimed serverless performance in general – including the cold start problem (Bardsley, Ryan and Howard, 2018). They created a rudimentary set of functions which carried out basic string operations, triggered by a request to an AWS API Gateway end point. Whilst they experienced similar issues with regards to the increased cold start time, they also noticed further delays.

#	Average	Median	Deviation	95%	99%	Min	Max
1,000	112ms	96ms	63.45ms	213ms	338ms	46ms	1,002ms

Figure 6 jMeter Test Results (Bardsley et Al. 2018)

Figure 6 shows the test results from one of their test runs. On checking the numbers, they noticed that there were further delays which were "unlikely to be due to cold starting Lambdas"; hinting that they instead could be caused by some form of start-up process taking place in API Gateway. This is certainly something which should be monitored in this project so as to ensure the results are not skewed in any way.

In the evaluation of their study, Bardsley makes special mention of runtime and the potential impact this could have on both cold start and indeed execution time. He goes say that special care should be taken when choosing a runtime and that the nature of the language should be taken into account. He states that "Node.js or Python Lambda's…would reduce latencies since these languages are less susceptible to problems with cold starts." And concludes by saying that tasks where processing performance is an issue, then a compiled language may prove more suitable. Whilst this is not explored in any great depth, it provides a good overview of the part runtime potentially has to play in the cold start issue and will definitely be explored further alongside the potential effect of memory allocation.

2.3 Examination of Amazon Lambda Runtimes

As Amazon Lambda has evolved and become more widely adopted, the list of supported runtimes has also grown. Starting out with Node.js in 2014, Amazon Lambda now supports Python, Java, C# and Go(lang). Whilst having this additional language support makes the service more accessible to a wider audience, it also poses the question about how these languages perform in comparison to one another in the confines of the platform. This project will aim to compare the three most popular languages as of 2018; Figure 7 Top languages over time (GitHub, 2018) shows the rankings of each language on the GitHub platform since 2014 (*The State of the Octoverse, GitHub*, 2018).

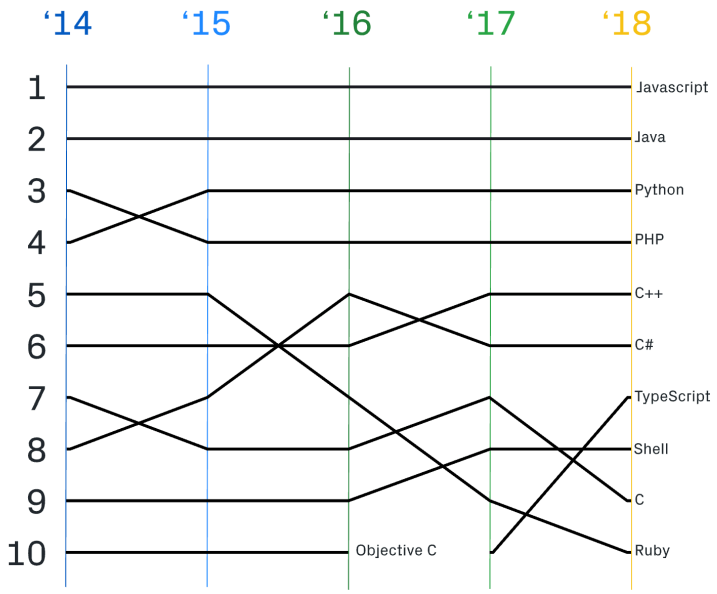


Figure 7 Top languages over time (GitHub, 2018)

2.3.1 Node.js (JavaScript)

2.3.1.1 Overview

JavaScript is a dynamic scripting language which, along with less stringent syntax rules, makes it appealing to a much wider audience (Spinellis, 2005). Although JavaScript has long been considered the de-facto standard for client-side web enhancement (Heo *et al.*, 2016), JavaScript has witnessed a large resurgence and found a new place within the development stack of many in the form of Node.js. Released in 2009, Node.js is a JavaScript runtime environment written by Ryan Dahl which allows developers to write and execute JavaScript code out with the confines of the browser. As well as the traditional standard features of the JavaScript language, Node.js brings with it a number of new system-level API's which facilitate a whole host of additional functionality such as file-system operations and networking capabilities (Liang *et al.*, 2017).

2.3.1.2 Under the hood

At its core, Node.js relies on a single-threaded asynchronous event model instead of multi-threading. The benefit of this is that, even though processes run on a single-thread, the non-blocking nature means that the application can continue to operate using callback functions without the need to wait on a task to completing (Tilkov and Vinoski, 2010).

Node.js is built on the V8 JavaScript engine, developed by The Chromium Project for Google Chrome in 2008. The V8 engine drastically changed the way JavaScript was implemented and consumed on the web. Instead of interpreting bytecode, as was the case up until its launch, V8 compiles JavaScript directly to machine code before being executed thus greatly improving performance (*Chromium V8 Documentation*, 2017).

2.3.1.3 Dependency Management

As well as the built-in modules which ship with Node.js, Node Package Manger (npm) is the default package manager which provides access to a huge collection of pre-existing Node modules and libraries. Launched in 2009, there are now over 800,000 packages available (as of November 2018) via the registry (*About Node Package Manager*, 2018). This vast repository is another reason which makes Node extremely popular.

2.3.2 Python

2.3.2.1 Overview

Python is an object-oriented, multi-purpose programming language which has a very readable syntax (Jing Zhang, Hongxia Luo and Xueqing Zhang, 2011). Python has seen a resurgence in recent years, especially in the academic and data science communities given its very strong mathematical and statistical packages (Silva *et al.*, 2003). Developed by Giudo Van Rossem in 1991, Python has been around for a relatively long time in comparison to some of the newer languages. Whilst Python is dynamically typed, it doesn't have the same loose typing that JavaScript has and will encounter errors on operations where types are unexpected and / or mismatched.

2.3.2.2 Dependency Management

Similar to Node.js, Python has a huge community and has its own large package repository called PyPi. Whilst not as vast as npm, as of March 19, 2017 PyPi had almost 1.3TB worth of package files (Oakes *et al.*, 2017). By making these packages and widely available, it can greatly reduce the amount of development required on projects – developers can simply search for something they are looking for before developing their own code from scratch.

2.3.3 Java

2.3.3.1 Overview

Java is a widely used object-oriented programming language which is said to be running on an estimated fifteen billion devices today (Oracle Corporation, 2018b). Developed by James Gosling in 1991, Java had originally been created for use in interactive television technology but ultimately was never implement for this purpose (Oracle Corporation, 2018c). From servers to printers and everywhere in between, Java can run almost anywhere. Figure 8 shows an extended history of the TIOBE Index of programming languages which illustrates just how popular Java has been in the past fifteen years and beyond. Java is a static, strongly typed language and is hugely popular amongst larger enterprise organisations due to its proven stability and scalability.

Programming Language	2018	2013	2008	2003	1998	1993	1988
Java	1	2	1	1	17	-	-
C	2	1	2	2	1	1	1
C++	3	4	3	3	2	2	4
Python	4	7	5	10	24	18	-
C#	5	5	7	8	-	-	-
Visual Basic .NET	6	11	-	-	-	-	-
PHP	7	6	4	5	-	-	-
JavaScript	8	9	8	7	22	-	-
Ruby	9	10	10	18	-	-	-
Swift	10	-	-	-	-	-	-
Objective-C	14	3	40	49	-	-	-
Perl	16	8	6	4	3	11	22
Lisp	29	12	16	13	8	6	2
Ada	31	19	19	15	12	5	3
Pascal	187	14	14	96	4	3	13

Figure 8 TIOBE Index, Long Term History (2017)

Java code is compiled and runs, for the most part, on a Java Virtual Machine (JVM) which makes programs extremely portable. Once Java is compiled, it performs particularly well and can be run on any JVM without the need to recompile (Oracle Corporation, 2018a).

2.3.3.2 Dependency Management

Whilst Node.js and Python have npm and PyPi respectively with regards to package and dependency management, Java is somewhat more complex. Maven is widely considered the industry-standard dependency resolution tool in the Java eco-system however Maven is more than just a package manager. Essentially, Maven is a complete build automation, project management, reporting and documentation tool which aims to centralise and simplify the entire build process for Java projects (Xiong Zhen-hai and Yang Yong-zhi, 2014).

2.3.4 Existing work on runtime performance

Whilst some academic studies have been carried out in the area of serverless performance, there is little to no consideration of runtime environment. There are however several prominent figures in the serverless community who have carried out experiments with a focus on runtime.

In 2017, Yan Cui - Principal Engineer at DAZN, carried out a range of tests in order to compare the performance of Node.js, Java, C# and Python on the AWS Lambda Platform (Cui, 2017). The function code itself was extremely rudimentary – a fairly common "Hello World" – which was written and deployed in each of the respective languages using the Serverless Framework, making use of AWS API Gateway to link the functions to an HTTP endpoint. Cui then tested each of the functions over a period of one hour and gathered the results. His initial observation was that, although each of the languages appeared to perform fairly consistently, "C# was sticking out like a sore thumb" with an average execution time almost double the others as illustrated in Figure 9.

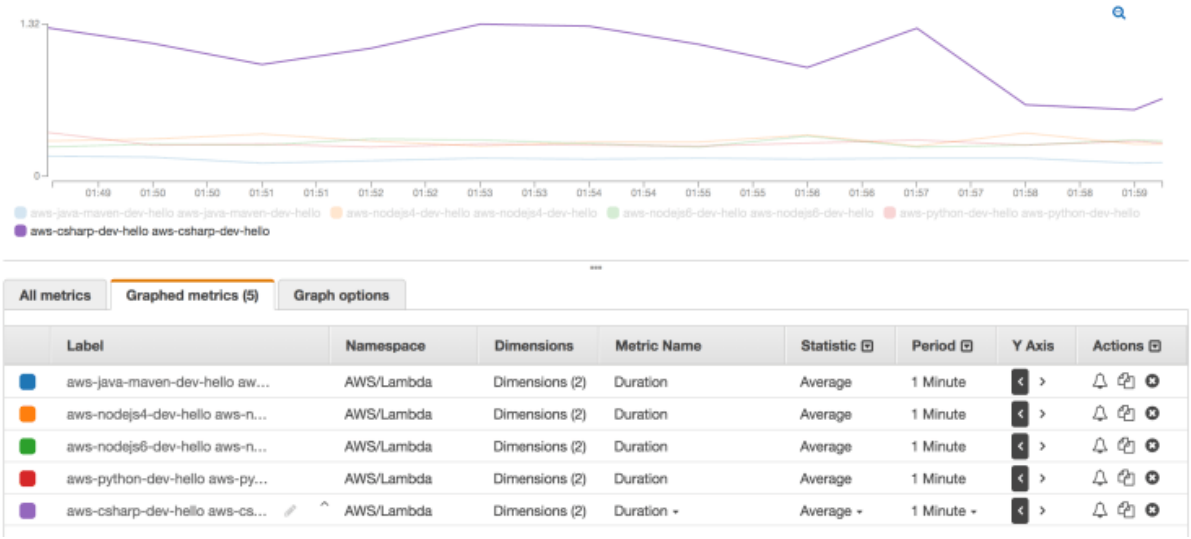


Figure 9 Average Execution Time (Cui, 2017)

Commenting further on the results, he states that both Java and C# seemed to generate the most consistent performance overall, hypothesising that this could be as a result of both being compiled languages.

In 2018, based on the previous work carried out by Yan Cui, Yun Zhi Lin – Vice President of Engineering at Contino – re-ran the original tests whilst taking into consideration any new or updated runtimes including Go(lang). Zhi Lin makes mention of memory in his test, detailing that all functions are ran with an allocation of 1024MB memory; Yan Cui makes no mention of memory in his original tests in 2017.

Overall, the test results were fairly similar with some improvements – probably as a result of platform improvement by Amazon – however Zhi Lin re-stated the apparent performance link between compiled languages and their consistent results "the compiled languages of Java and .NET Core 1.0 were clearly more consistent compared to the dynamic languages – Python and Node.js" (Zhi Lin, 2018).

Whilst these tests produced some good results and highlighted some key areas for further investigation, it should be noted that both Cui and Zhi Lin deliberately excluded cold start results from the tests. They also make note of the disparity in package size between the compiled languages and dynamic languages where, in some cases, the compiled packages can be almost nine times larger. This, along with the cold start data, is something that will be explored explicitly as part of this project.

2.3.5 Dynamic / Interpreted vs Strongly Typed / Compiled

From the literature evaluated so far in both this section of the literature review and the section on the cold start problem, it is evident that there is a link between the typing of the language as well as the means with which is it converted ultimately to machine code. The dynamic, interpreted languages would appear to reduce the severity of a cold start when compared to a compiled language such as Java or C#. Conversely, the compiled languages display better performance once the Lambda's are warm.

3 Execution

This section of the report will outline the tasks carried out in order to create the API prototypes and the subsequent semi-automated tests which were developed to investigate the cold start problem and its relation to runtime environment. An incremental approach was adopted in this section in order to hone and refine the prototypes and tests as the project evolved.

3.1 Design

As outlined in previous sections – the aim of this project was to develop a small API which could be used to test how the choice of runtime affects the impact of a cold start in a Lambda function. The API would be developed in the three different runtimes as detailed in the literature review – NodeJS, Python and Java. Each would subsequently be deployed to Amazon Web Services using the Serverless framework which would facilitate the commissioning of all necessary resources including AWS API Gateway and AWS Dynamo DB.

The decision was taken to develop a simple API which would provide the functionality of a basic "to-do list" style back-end. This is a popular use-case prototype used when learning new technologies given the relatively simple yet effective create, retrieve, update and delete (CRUD) operations required throughout. Although somewhat rudimentary in design and complexity, it was thought that this design would provide the necessary functionality – with the subsequent testing artefacts and resultant data providing the real bulk of the analytics for investigation.

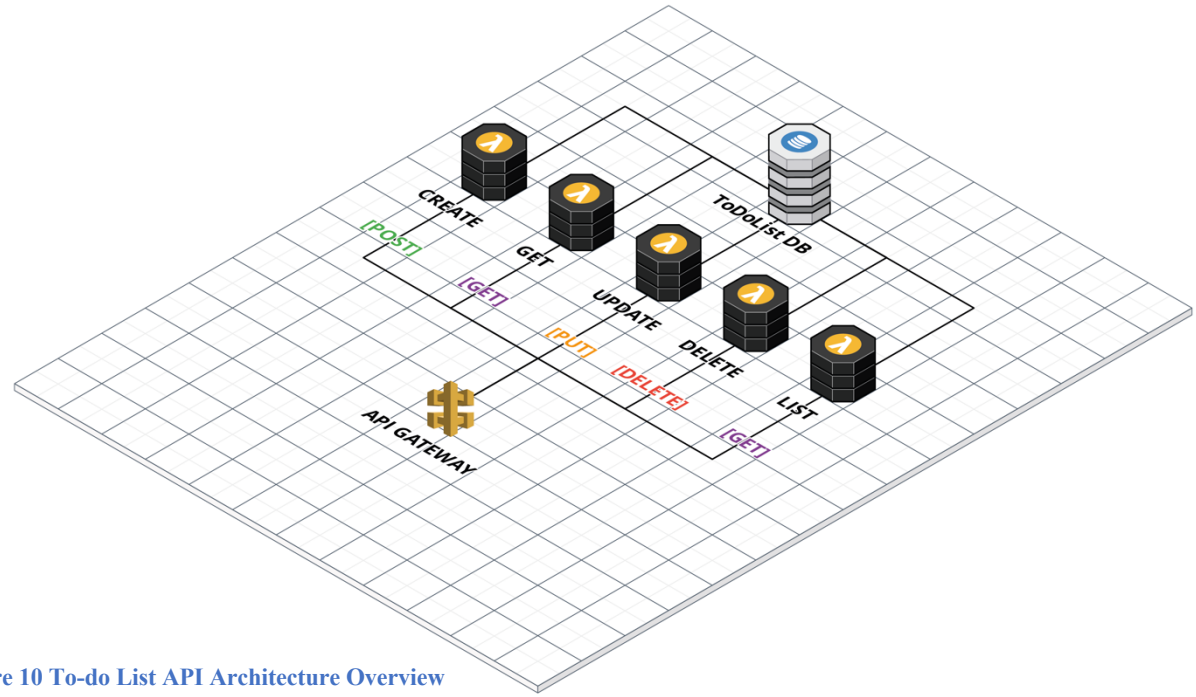


Figure 10 To-do List API Architecture Overview

The API scenario was chosen in order to simulate a more realistic use-case development for Lambda functions. To date, the existing research has involved little to no actual logic within the lambda function themselves – often simply outputting timestamps as was the case in Yan Cui's test from the literature review. Existing data on the topic, therefore, may not necessarily

be representative of the potential overheads a cold start problem could introduce in a real-world production environment such as a public facing API. Figure 10 outlines an overview of the API architecture and the services involved within AWS. This architecture is the same across all three runtimes; requests enter via the API Gateway and are routed, depending on the request and HTTP Method, to the associated Lambda function. The functions themselves then interact via Dynamo DB to carry out the appropriate operation. A response is then returned from the Lambda and routed back to the user via the API Gateway.

3.1.1 Requirements

As a result of the relative simplicity of the to-do list prototype, the requirements for the API's themselves are equally modest. Table 1 features a breakdown of the Lambda functions from the Architecture overview in Figure 10 – each function will be responsible for its own actions and will have a dedicated API Gateway route which it will be triggered by.

Table 1 Lambda function outlines

Lambda Function	API Gateway Routing	Action Description
Create	**POST** `/todos`	Should take a JSON request body and create a to-do list item in the Dynamo DB.
Get	**GET** `/todos/{id}`	Should retrieve a to-do list item from the Dynamo DB based on the id passed via the URL.
Update	**PUT** `/todos/{id}`	Should take a JSON request body and update a to-do list item in the Dynamo DB.
Delete	**DELETE** `/todos/{id}`	Should delete a list item if it exists in the Dynamo DB based on the id passed via the URL.
List	**GET** `/todos`	Should retrieve a list of all to-do list items stored in the Dynamo DB.

3.2 Serverless Framework

Having outlined the design of the API and its requirements, it had to be developed in each runtime in order to be deployed to Amazon Web Services for testing. The Serverless framework was chosen, in part, to help with reducing the time it would take to develop the API in each runtime and deploy.

The Serverless Framework provides a number of boiler-plate templates to get started in any given language. These were leveraged in order to speed up the development process whilst ensuring each runtime was being developed from a similar foundation. When creating a project, the template sets up the basic folder structure, configuration files and language files. Almost all setup is managed from the serverless.yml configuration file located in the root of each project. From defining the functions and memory allocation, to API Gateway configuration and Dynamo DB resources the yaml file is responsible for all configuration. The configuration for each runtime is almost identical making it extremely easy to replicate once the first instance had been created.

3.2.1 NodeJS API

The NodeJS implementation was developed using the Amazon Web Services Software Development Kit (AWS SDK) - this provided a set of classes and methods for interacting

with the wider AWS platform - including Dynamo DB. This was acquired via the Node package manager NPM which managed the dependencies of the project. The logic for each function was separated out into five individual files – one for each Lambda each action in the API.

3.2.2 Python API

The approach to the Python implementation was fairly similar to the NodeJS version of the API. Python instead makes use of the Boto3 library – a python implementation of the AWS SDK. This was managed via the PyPi dependency manager which served a similar purpose to NPM.

3.2.3 Java API

The Java implementation of the API was somewhat more involved given the more complex nature of both the Java language and the fact that the serverless framework requires a compiled artefact in order to deploy to AWS. As with the other two runtimes, the Java API made use of the AWS SDK which was managed by Maven – a Java package manager and build tool. Maven was also responsible for building the jar file artefact which the Serverless framework would package and upload to AWS for deployment.

3.3 Testing

Once the API had been developed and deployed to AWS across all three runtimes, the tests had to be designed and developed. This section will outline the testing tools as well as the test themselves and the data being gathered.

3.3.1 Testing Tools

The original intention was to utilise Artillery – a Node based testing tool – in order to carry out the majority of testing against the API's. However, on the first set of exploration tests it was apparent that Artillery wasn't going to provide sufficient metrics with enough control to satisfy the needs of the project. Instead of providing individual test results for each request, Artillery simply returns averages from each test run. The mathematics being used to calculate these averages were unclear - as a result, it was decided that completely bespoke semi-automated tests would be written from scratch using a combination of bash scripts and the Linux curl library.

To ensure that each test could be evaluated with a shared baseline, all tests would be carried out on the same machine with a wired local area connection. This primarily to reduce the potential impact of any wireless connectivity issues through network interference or packet loss.

3.3.1.1 Test Metrics

Given the nature of this project, it was important to consider the data which would be gathered at the testing stage. Data gathered at this point would ultimately serve as the basis for analysis and evaluation. The cURL library provides a number of timing metrics which can be captured for each request however the most suitable in this case was the ***time_total*** field which logs the time the entire request and response to be received.

In addition to the total time, it was also important to log the HTTP Status code of each request. Given that these tests were being ran in an automated fashion – i.e. the script would simply loop requests over a period of time – it was vital to have a way to identify requests that failed; the status code would give a fair idea of the problem. It would enable quicker cross-reference with the AWS should the need arise.

3.3.2 Lambda Timeout Tests

Whilst the main focus of this project is the cold start itself - and how this is exhibited across different runtimes – it was first important to ascertain the point at which lambdas go "cold". As discussed previously, after a period of inactivity the Lambda container will be destroyed to free up resources – this means when the next request is received, a new container must be spun up and bootstrapped with the necessary code and dependencies to handle the request. Amazon do not publish the details for the criteria which needs to be met for a container to be destroyed – as a result it was key to ascertain this information to proceed with the subsequent cold start testing.

From the literature review, and the previous work done by Yun Zhi Lin, it was thought a container could exist from anywhere between a few minutes up to forty-five minutes of inactivity. With that information, a test was written which would call the API in each runtime then wait five minutes. After each subsequent API hit, an extra five minutes would be added to the sleep timer to increase the intervals. On review of the results, it is hoped that there will be a discernible difference between the times to pinpoint a reasonable accurate idea of what the containers are destroyed.

3.3.3 Lambda Cold start Tests

Once the point at which containers are being destroyed has been determined, it will be possible to design a test to gather results for cold starts across all three runtimes. As this was going to form a significant part of the research – it was important to consider this test carefully.

Initially, the functions deployed at this stage would utilise the default Serverless framework deployment settings which set a memory allocation for each function at 1,024 MB. In this test, an end point for each runtime would be called to create a new to-do list item within each API. The test would then pause for the desired internal – gleaned from the previous test – before sending another request to create an additional to-do list item. This test would be repeated 100 times in order to gain a broad range of results for more accurate evaluation.

At each iteration of the test, the data gathered – response time, http status code, date/time – would be written out to a results log file for retrospective evaluation. Given that the timeout for containers could be anywhere between a few minutes up to forty-five minutes, it's important that this test is robust in well written in order to run for long periods of time. Going on the worst-case scenario this test may have to be ran for up to seventy-five hours (100 x 45-minute intervals).

3.3.4 Lambda Memory Tests

In the literature review, it was noted that the allocation of memory seemingly had an impact on the performance of a cold start. In their investigation into serverless architecture and its potential use with deep learning, Ishakian et al discovered that not only did increasing the memory have an impact in reducing the cold start time – it also improved their overall prediction times. From this it was evident that this is something which needed to be explored in this project.

In order to establish a suitable test for this scenario – each runtime was subsequently deployed with varying memory allocation. The Serverless framework was extremely useful in facilitating this with a simple configuration update and no extra code. The following memory brackets were chosen for this test; 512 MB, 1024 MB, 2048 MB. Having studied the logs from the previous test run – which allocated a default tier of 1024 MB – the functions

themselves were utilising merely a fraction of that memory. Table 2 summarises the average memory usage for each function from the previous test. By testing at both double and half the default memory allocation, it was hoped that there would be a demonstrable correlation between these changes in the results. Given the relatively low memory usage – testing any higher than 2048 MB was deemed to be a potential waste of resources.

Table 2 Average memory usage from initial cold start test

Runtime	Python	NodeJS	Java
Average Memory Usage	69 MB	70 MB	140 MB

The test itself would be similar in structure and logging as the previous cold start test with one minor addition. At each iteration of the test in sending a request to each runtime – this test would introduce a sub-iteration which send 2 additional requests to the 512 MB and 1024 MB functions respectively before waiting for the next cold start. Each iteration would be logged as before whilst this time including the memory allocation in the logs for differentiation in the evaluation stage.

3.3.5 Lambda Regular Execution Tests

For the final set of HTTP API tests, it was hoped to gather a set of results for the normal execution of a "warm" Lambda function. Whilst this project is focused primarily on the impact the choice of runtime has on the cold start – it is also important to consider and evaluate the impact the runtime has on regular execution times. With that in mind, the final test took a similar form to the memory tests but, instead of waiting on a cold start, there would only be a small one second pause. The reason for keeping a small interval was to reduce the potential for any further cold starts. As outlined in the introduction, one of the main benefits of serverless computing and Amazon Lambda is its ability to seemingly auto-scale to meet the demand of traffic spikes. The downside of this is that in doing so, this introduces further cold starts as new containers are provisioned. The one second interval in this final test would ensure that the Lambdas would not be forced into provisioning further instances thus skewing the results of the regular execution test.

The inclusion of varying memory levels from the previous memory allocation test would be maintained in this test with the hope of identifying any potential correlation between not only memory allocation and a cold start – but also memory allocation and its impact on runtime and regular execution. Results would, again, be logged to text files as outlined in previous tests for posthumous evaluation.

3.3.6 Direct Invocation Lambda Tests

During the literature review it was noted that, in a recent experiment into serverless performance, Bardsley et al. had observed an unexpected delay in some of their results. They hinted that this may not be due to a cold start and went on to speculate that this could be as a result of some form of API Gateway delay. Given that, it was important to explore this further in order to eliminate a potential skew in the findings of this project.

On that basis, a test was devised on order to attempt to bypass the API Gateway factor completely. Within the Serverless Framework it provides the ability to invoke Lambda functions directly from the command-line interface. In addition, it also facilitates the export of logs for each invocation. Again, using bash, an automated test was developed which would invoke each runtime function via a Serverless Framework command at the interval based on

the cold start time from the initial timeout tests. In addition, the test would invoke each memory tier as outlined in the earlier memory tests. Any vast deviation between these results and the earlier memory tier tests via API Gateway could confirm the original theory of Bardsley et al.

As with the previous tests – a final set of “warm” results would be logged using the pure Lambda however the test would not pause to wait on a cold start.

4 Evaluation and Discussion

This section will outline the test results from the execution stage and will seek to present and critically analyse the finds of the results in-depth.

4.1 Timeout Test

The first of the tests carried out was the timeout test. This had been devised in order to ascertain the point at which Lambda functions go "cold" – that is to say at which point AWS deems the resources no longer required and thus destroys the containers. Any subsequent requests will trigger the commission and bootstrapping of a new container – the resultant delay in this process is known as the cold start.

The test was an instrumental part of the research as it ultimately formed one of the important metrics which the subsequent tests would rely on in order to accurately trigger a cold start. Conducted across all three runtimes, the test essentially sent an HTTP request to API gateway which would trigger the Lambda function thus rendering it warm. The test would then send subsequent requests after a cumulative wait of five minutes at each iteration. The chart in Figure 11 outlines the results of this test. The x-axis illustrates the time interval at which each runtime is called. At zero minutes, the cold start is quite evident in comparison to the results at the five-minute marker. The fall in duration can be seen across all three runtimes. The durations themselves remain fairly steady until the thirty-minute marker where there is a noticeable jump in duration. This trend continues there after which would support the idea that containers are being destroy between the twenty-five and thirty-minute point.

It should be noted that a slight increase in the NodeJS runtime can be seen at the twenty-five-minute point however this may not necessarily hint at a short life-space but could also be network congestion and the fact that all three tests were being ran concurrently. Whilst the evidence of all three runtimes being cold at the thirty-minute marker was enough to satisfy the requirements of this test and its use in subsequent tests – this could be explored in finer detail in order to gain a more accurate picture of the exact timeout point. It may also ascertain whether or not NodeJS Lambdas do indeed have a slightly shorter lifespan than the others.

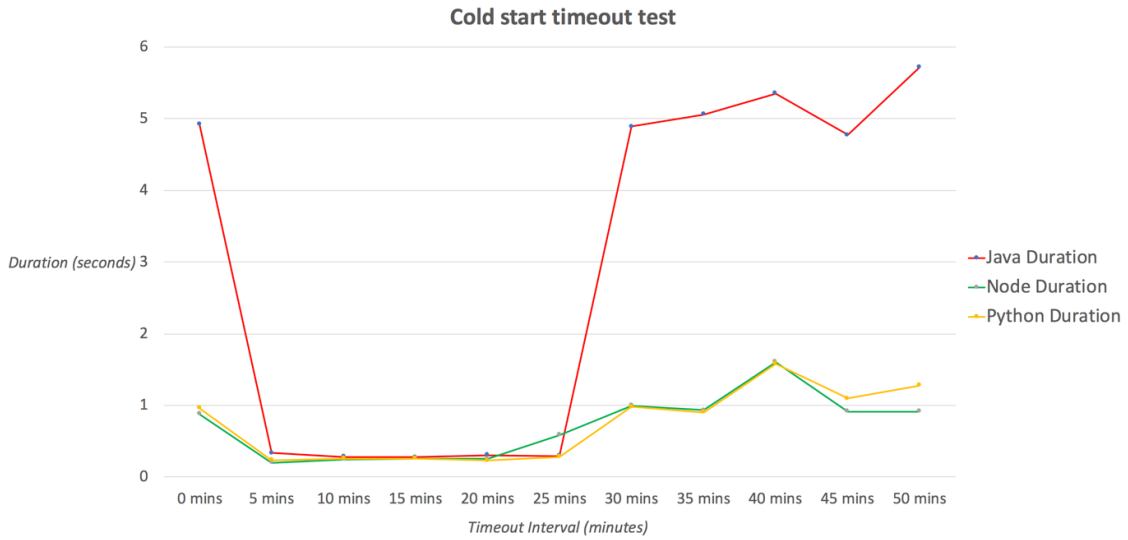


Figure 11 Cold start Timeout Test

What is immediately evident from this test is stark contrast in cold start times between the Java runtime and the NodeJS and Python runtimes. Whilst Figure 11 would seem to suggest they execute fairly similarly warm (this will be explored in greater detail in the subsequent tests) the difference between them when cold is over four seconds in some cases.

After the suspected cold start point of thirty minutes, there does appear to be some fluctuation in durations. In particular, all three runtimes seem to spike at the forty-minute point.

4.2 Cold Start Test

Having established the point at which containers were being disposed of as between twenty-five and thirty minutes – the next run of tests could be initiated. Given the timescales involved with the fairly lengthy intervals, this part of the project took over fifty hours in order to gather results alone. These tests form the most significant part of the research and are intended to ascertain the impact that runtime environment has on the cold start. As a result, each runtime required thorough exploration before wider evaluation.

It should be noted that there was some conflict when this test took place – the API's were accidentally triggered out with these tests which meant that for some of these tests runs the Lambdas were warm when they should have been cold. Figure 12 shows a range of outlier points highlighted in red which appears to be when the conflict took place. As a result, these outlier points will be excluded from the subsequent evaluation for this to so as not to skew the results.

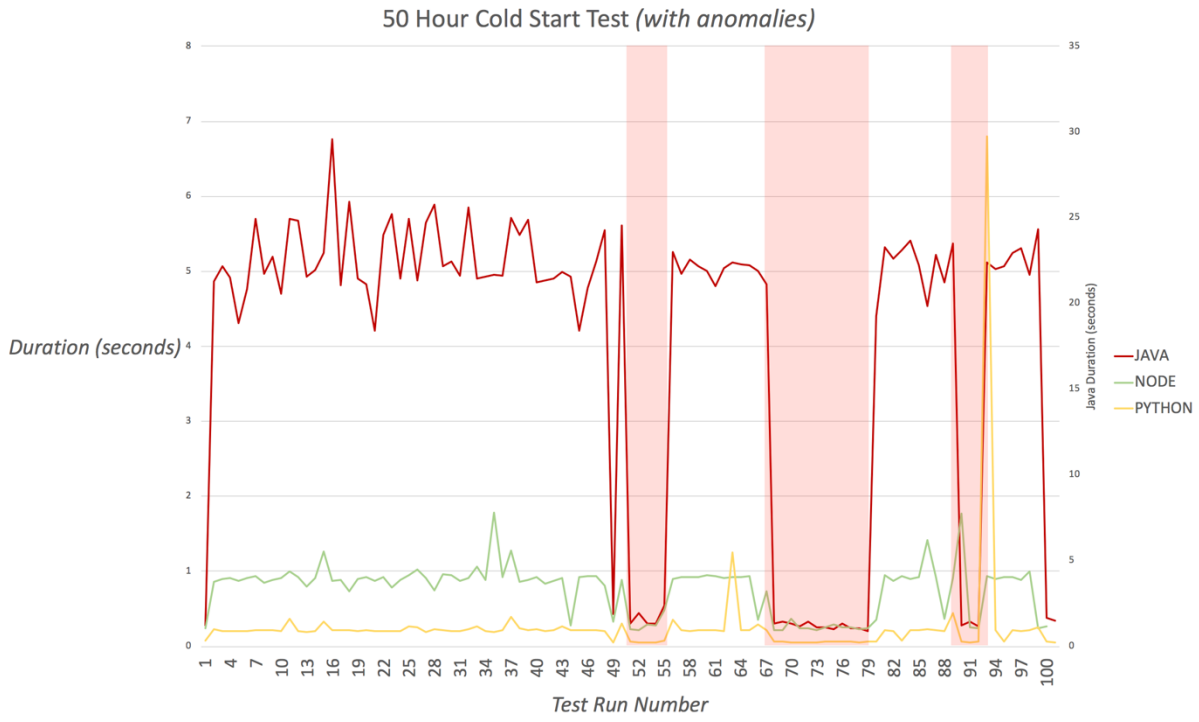


Figure 12 50 Hour Cold Start Test (with anomalies)

4.2.1 Java

Overall, the Java runtime displayed a consistently high cold start time. Table 3 details the overall statistics from the Java test run at shows that both the mean and median averages of the cold starts above five seconds which is quite a significant delay. In the worst case, there was a difference of over two seconds between the minimum and maximum duration.

Table 3 Java Cold Start Statistics

Runtime	**Average**	**Median**	**Min**	**Max**	**Std. Dev.**	**99p**
Java	5136ms	5068ms	4208ms	6766ms	417ms	6129ms

4.2.2 Node

In stark contrast, the Node times were overall significantly lower than Java – an overview of which can be seen in Table 4. Even comparing the maximum execution time Node with the minimum execution time of Node, there is almost a two and a half second gap. Furthermore even with the removal of the outlier points mentioned earlier, there does seem to be some unusual duration spikes in the data. This could be the result of further cross-lines between the tests but given that the result is higher spikes, this could perhaps be the result of network latency or interference.

Table 4 Node Cold Start Statistics

Runtime	**Average**	**Median**	**Min**	**Max**	**Std. Dev.**	**99p**
Node	943ms	917ms	732ms	1789ms	173ms	1773ms

4.2.3 Python

The numbers gleaned from the test of the Python runtime are extremely similar to the results demonstrated by the Node tests. The statistics from the Python cold start tests as shown in Table 5 are evidently similar to the Node tests with one difference - it would seem that Python ran slightly slower in these tests. There is a marginally higher standard deviation in the overall spread of results in Python. This could be one of several things – Python could be slightly less stable than Node accounting for the larger spread, or network latency in sending or receiving the request and subsequent response is skewing the results slightly.

Table 5 Python Cold Start Statistic

Runtime	**Average**	**Median**	**Min**	**Max**	**Std. Dev.**	**99p**
Python	998ms	924ms	837ms	1926ms	204ms	1761ms

4.2.4 Conclusion

Overall, from this test is it quite evident that there is a clear distinction between the cold start performance of the interpreted languages – Node and Python – versus the compiled language Java. The difference is fairly stable across the entire test – this is also consistent with the initial theory of the project which was that the Java runtime would indeed take much longer to commission and bootstrap the resources required to launch the Java container.

Whilst removal of the outlier points did seem to level the results somewhat - looking at the graph of the average results per hour, outlined in Figure 13, this shows that there are still some unusual performance spikes. Whilst these could be a result of a spike in the cold start, it may also be that some other ambient interference is the cause – possibly with network connectivity or network load on the testing local area network.

What is plainly clear from the initial analysis of this first test run is that there is a clear difference in the cold start performance of these runtimes.

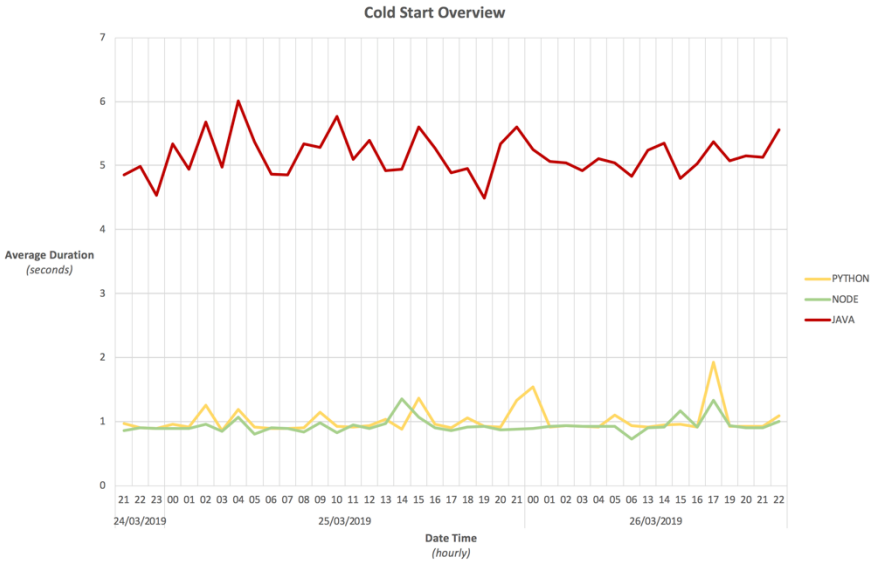


Figure 13 Cold Start Overview Graph (All runtimes)

4.3 Cold Start Memory Tests

Whilst the cold start tests evaluated in the previous section do show a strong correlation between the compiled and interpreted languages, these tests were carried out using the default memory allocation as inherited from the Serverless Framework – 1024 megabytes. In the literature review, it was noticed that in previous work by Ishakian et al. it had been observed that there was a relationship between warm execution times and memory allocation. On that basis, this test aimed to explore that theory further by deploying the same functions not only across the three different runtimes but across three memory tiers. It was hoped that there would be demonstrable link which could be drawn between memory allocation and a reduction of the cold start time.

4.3.1 Java

As with previous tests, the Java runtime continued to display comparatively large cold start times. There was however a clear visible difference across the different memory tiers as displayed in Figure 14. Both the mean and median averages were almost identical with both displaying the same trend. By doubling the memory from 512 megabytes to 1024 megabytes there was on average a decrease of 20% in responses times; doubling this value again to 2048 megabytes so a further decrease of an average 37%.

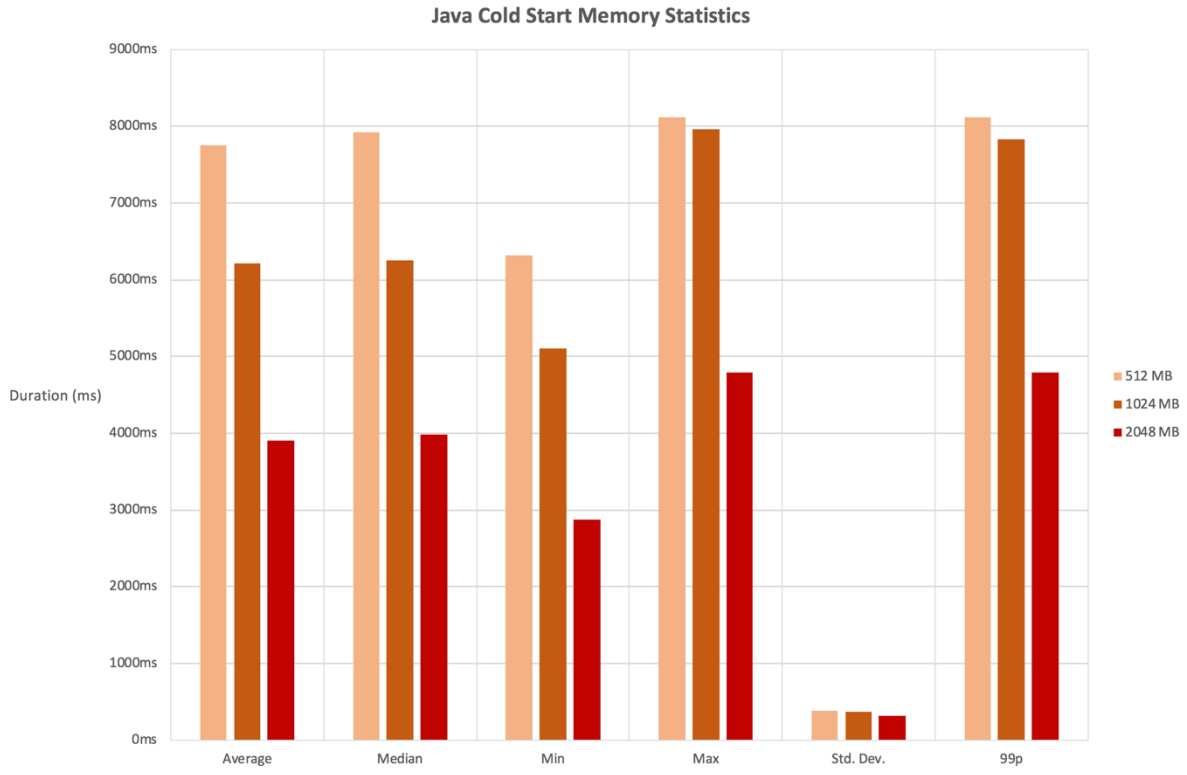


Figure 14 Java Cold Start Memory Test Statistics Chart

There did however seem to be a slight deviation from that trend with regards to both the minimum and maximum times on the 1024 megabyte memory tier with the minimum slightly elevated and the maximum almost on par with the 512 MB tier maximum time. Having cross-reference this with a graph of the 1024 megabyte tier, there were two spikes in duration which could account for this anomaly but it's again, there is the possibility of some interference with regards to network latency.

One important thing which should be noted is with regards to the HTTP status codes. Whilst all requests to the 1024 megabyte and 2048-megabyte tiers executed successfully – the 512

megabyte tier lambdas were all responding with an HTTP status code of 502; bad gateway. On further excavation of the logs in AWS itself, it was discovered that each request to the 512-megabyte lambdas were all timing out at 6000ms which was interesting on two fronts. Although the function was timing out, presumably as a result of lack of memory, further inspections revealed that the memory used by the function never exceeded 130 megabytes. Similarly unusual was that whilst the logs were timing out at 6000ms, results from the tests were showing times between six and eight seconds – in the worst instance, this was an additional 2,115ms unaccounted for. Whilst the latter is most likely a result of network latency between the both location of the testing equipment and Amazon's servers, it could also indicate a delay within Amazon's own platform as was hypothesised by Bardsley et al. in the literature review. This will be explored further in a subsequent test.

4.3.2 Node

Whilst the Java runtime displayed some good findings in terms of the relationship between runtime and the effect on cold start times, the Node results were not as conclusive. Where Java displayed a fairly significant difference, the Node results were in fact quite close – even across the memory brackets. There was a variation across the mean and median averages, with a slightly wider spread on the maximum times as displayed on Figure 15 however the numbers still seemed far from what was expected.

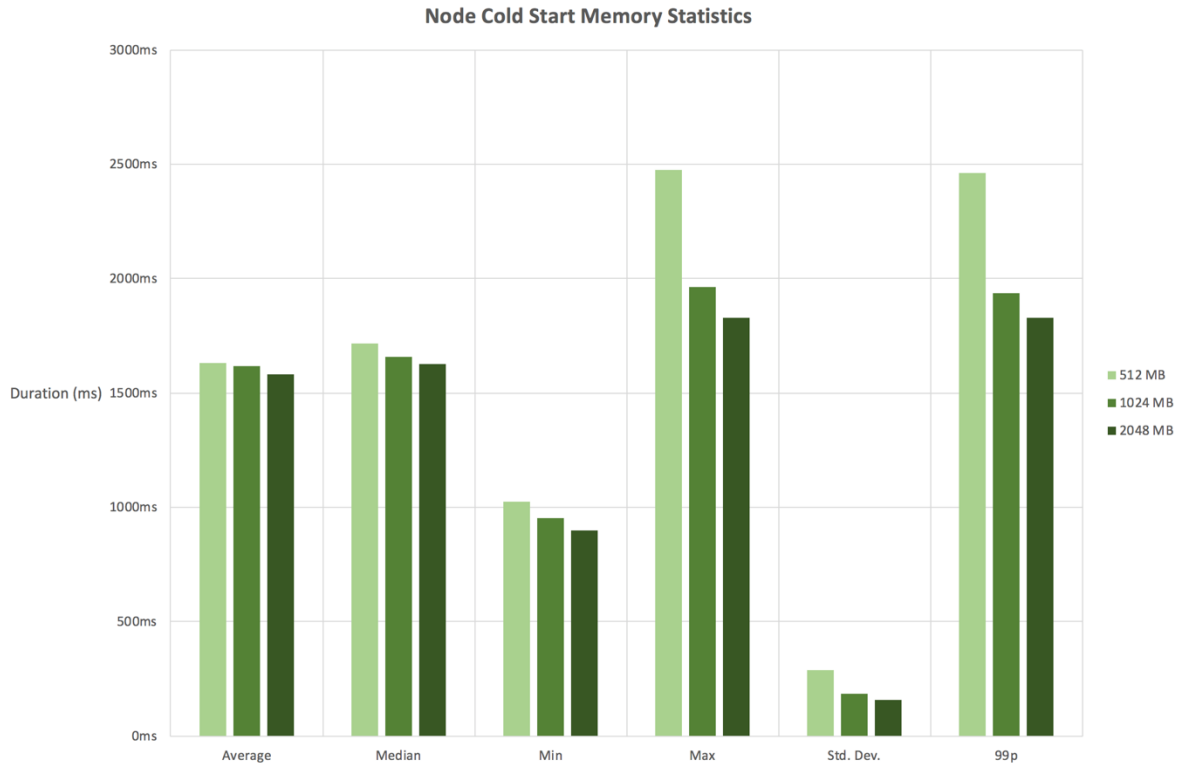


Figure 15 Node Cold Start Memory Statistics Chart

Furthermore, on comparison of the 1024-megabyte memory tier with the results from the initial cold start tests, it is clear there is some discrepancies with the results. Both the mean and median averages are exhibiting significant increase in times, with the median showing an increase of over eighty percent as shown in Table 6.

Table 6 Comparison on Node Results with 1,024 MB memory allocation

1024 MB Memory	**Average**	**Median**	**Min**	**Max**	**Std. Dev.**	**99p**
1st Node Cold Start Test *(Cold Start Test)*	943ms	917ms	732ms	1789ms	173ms	1773ms
2nd Node Cold Start Test *(Memory Tier Test)*	1617ms	1660ms	951ms	1962ms	187ms	1939ms
Percentage Change (%)	**+71.5%**	**+81%**	**+29.9%**	**+9.7%**	**+8.1%**	**+9.4%**

4.3.3 Python

On evaluation of the Python results, it unfortunately seems as though this test has suffered the same distortion of test results as the Node runtime. Looking at the graph in Figure 16 there is no real discernible trend which would support the link between memory allocation and the effect it has on runtime from there results. This is an increased standard deviation in the 512-megabyte tier which may go some way to explain why the 512 megabyte and 2048-megabyte tier have an extremely similar maximum duration of 1,923 milliseconds and 1,919 milliseconds respectively.

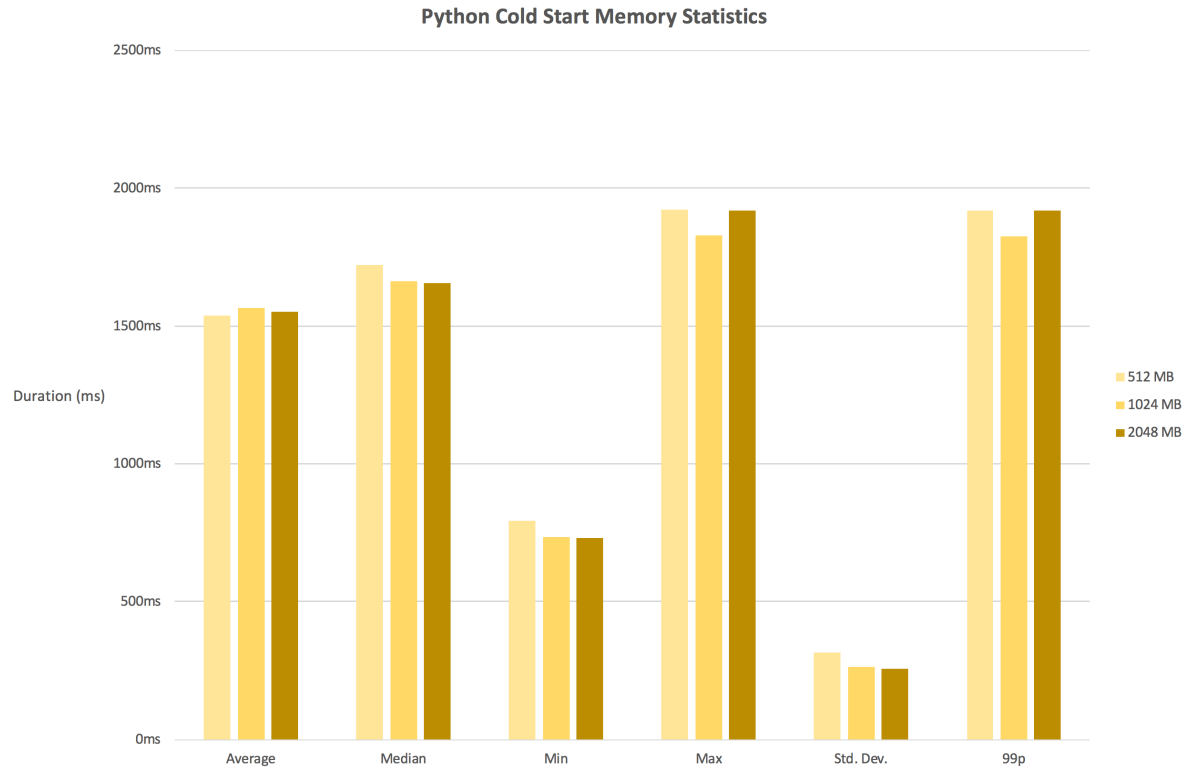


Figure 16 Python Cold Start Statistics Chart

4.3.4 Conclusion

Whilst the initial results from the Java runtime test seemed promising – they certainly exhibited the linkage expected between memory allocation and the impact this reportedly has on reducing cold start times. Unfortunately, as is becoming quite clear, the testing setup and approach would appear quite susceptible to result variations as a result of network congestion or connectivity issues. On that basis, whilst the Java data was encouraging, there can be no conclusive assertions drawn from the subsequent Node and Python tests as yet.

4.4 Direct Invocation Lambda Tests

It was noted earlier in the literature review that several studies had come across some unexpected delays in their results in different guises. This project too has observed varying results when triggering the functions across HTTP via API Gateway.

As mentioned in the execution section, the Serverless Framework, whilst enabling the configuration of API Gateway and other AWS resources to trigger Lambda functions – it also facilitates the ability to invoke Lambda functions directly from the command-line interface. The subsequent sections will discuss the results of those tests.

4.4.1 Direct Invocation Cold Start

The first direct lambda test was to re-investigate the cold start issue. A test was carried out similar to the test in section 4.3 - Cold Start Memory Tests – this time excluding any HTTP request/response latency and eliminating the need for API Gateway completely.

4.4.1.1 Java

The Java runtime again showed a similar correlation between memory allocation and the cold start times – the higher the memory allocation, the shorter the cold start time. It should be noted here that, in a similar fashion to the Java test carried out in section 4.3.1 – the 512 megabyte tier timed out as a result of lack of memory, even though only around 130 megabytes was consumed according to the AWS logs. For this reason, the 512-megabyte tier will be excluded from any charts.

The statistics in Figure 17 show the very consistent results of the test. There is very little deviation from the mean and median averages which suggest the results of this tests are far more stable than in previous sections.

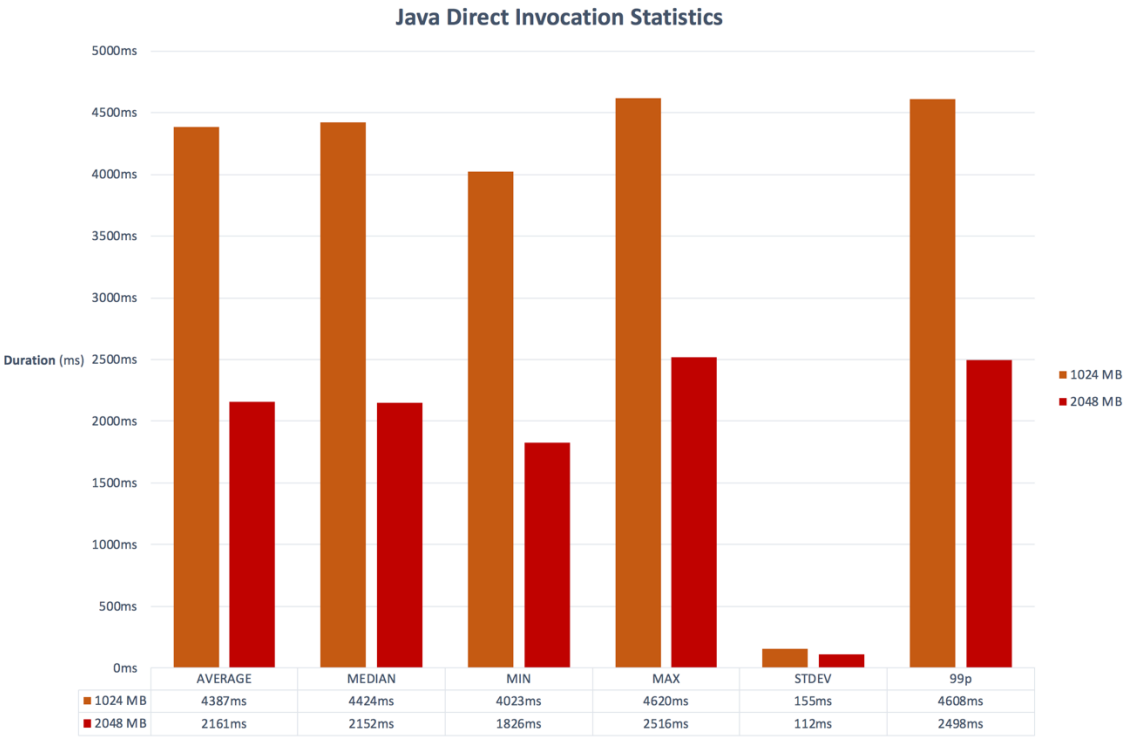


Figure 17 Java Runtime Direct Invocation Statistic (Cold Start)

In addition to this, the times themselves are generally smaller across the varying memory tiers. A comparison of the memory tiers against the first cold start memory test via API Gateway in Table 7 shows a significant percentage decrease across the board. The 2048 megabyte tier presented a forty-five percent reduction in cold start time by eliminating API Gateway alone.

Table 7 Comparison of Java results from original memory test

Memory Tiers	Average	Median
1024 MB	-29.38%	-29.26%
2048 MB	-44.66%	-45.89%

4.4.1.2 Node

Closer inspection of the Node runtime results of the direct invocation test shows an almost identical trend across all memory tiers including the 512-megabyte tier. This evidence further supports the strong link between memory allocation and cold start times.

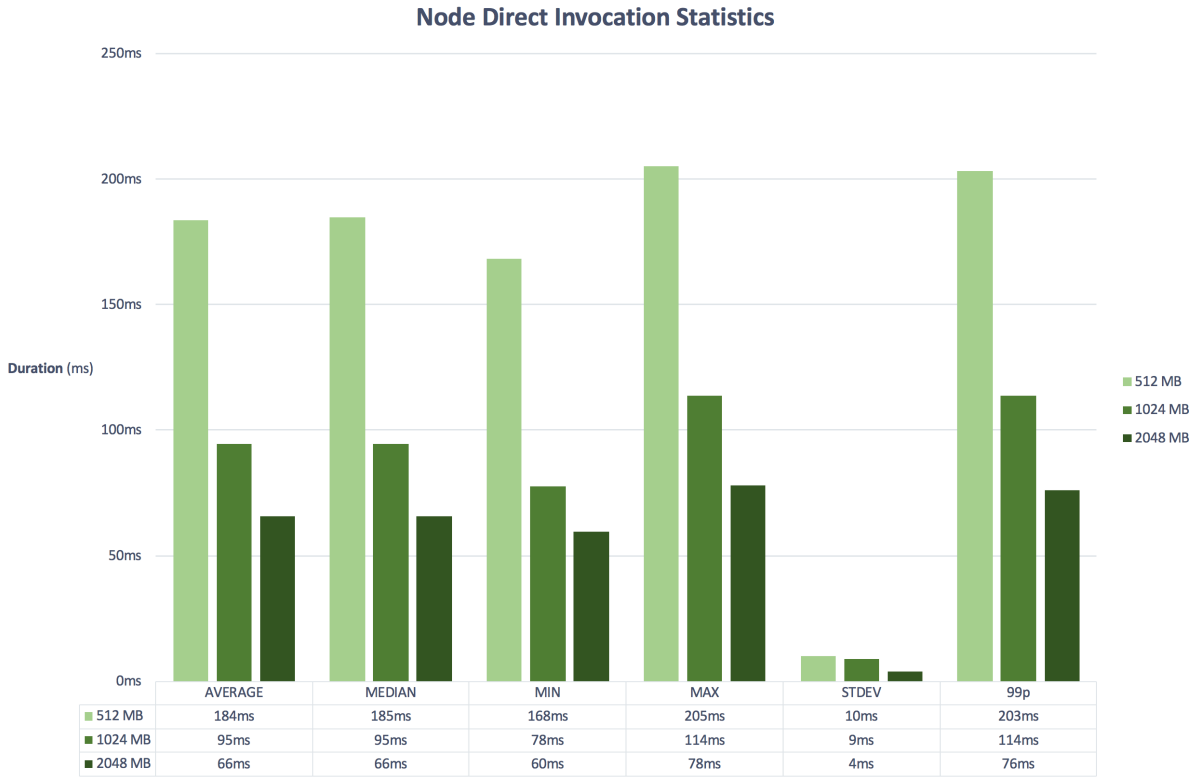


Figure 18 Node Direct Invocation Statistic (Cold Start)

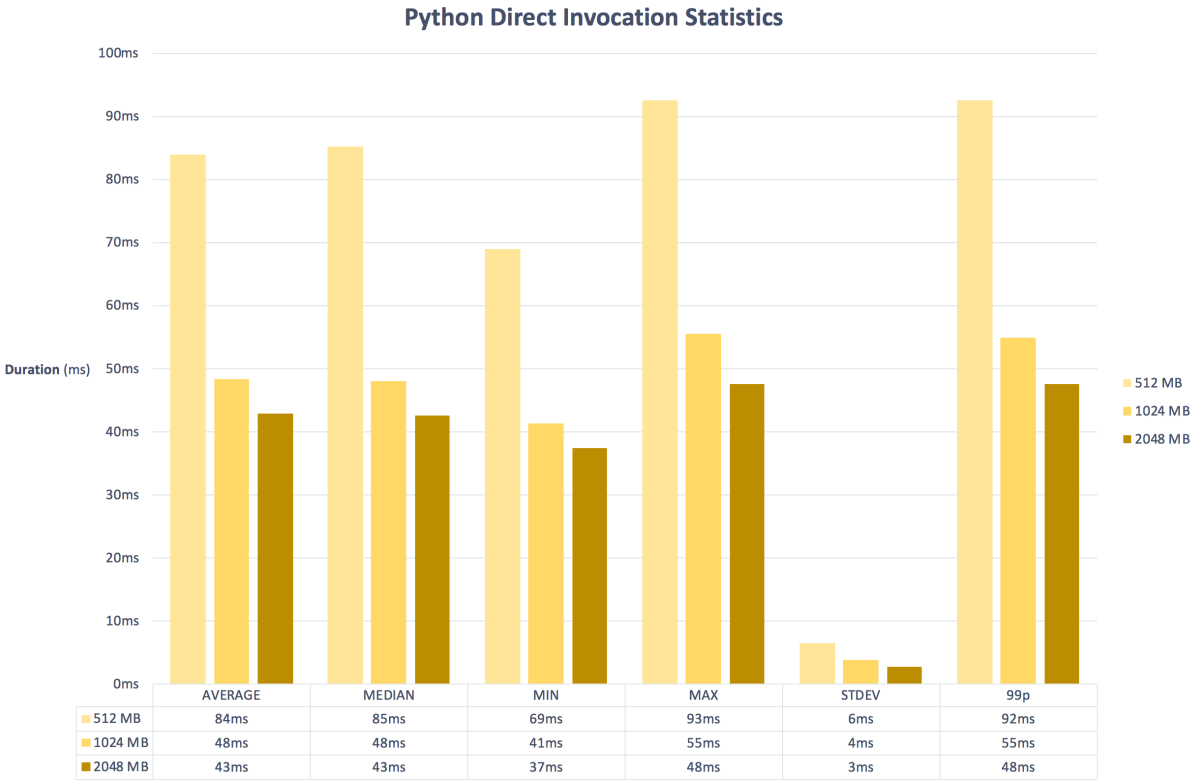
Evaluation of the statistics in Figure 18 reveals an almost identical trend to the Java results. There is very little deviation between the mean and median average as well as a relatively low standard deviation which would hint at a much more even distribution of results.

Instantly, there is a stark contrast between the results of Java, a compiled language, and Node an interpreted language. Even comparing the minimum Java cold start time (4023ms) with

the maximum Node cold start time (114ms) at the default memory tier of 1024 megabytes reveals a fairly substantial **97% decrease** in cold start time.

4.4.1.3 Python

Finally, Python exhibits an extremely similar trend to both Java and Node. There is a clear link between memory and cold start time with the time decreasing as the memory is increased. One thing gleaned from the chart in Figure 19 is that performance does seem to plateau a bit more than Node. This however is most likely to the further improvement on performance between Node and Python.



	AVERAGE	MEDIAN	MIN	MAX	STDEV	99p
512 MB	84ms	85ms	69ms	93ms	6ms	92ms
1024 MB	48ms	48ms	41ms	55ms	4ms	55ms
2048 MB	43ms	43ms	37ms	48ms	3ms	48ms

Figure 19 Python Direct Invocation Statistics (Cold Start)

4.4.1.4 Direct Invocation Cold Start Conclusion

It is clear from this test that direct invocation of Lambda functions not only provides more consistent test results, but that there is an unequivocal increase in performance across all three runtimes regardless of memory tier. That said, there was a clear difference of performance across the runtimes with Python coming out significantly faster than both Java and Node with seemingly the smallest variance of results – graph comparing the two can be viewed in Appendix A.

Whilst there was undoubtedly some skewing of the test results in the earlier tests – owing to network latency or connectivity issues – it is also quite evident that there is indeed some overhead involved with the addition of API Gateway to the architecture. Although it seems logical that the addition of an extra step in any process would increase overheads, it is unclear from the research carried out in this project to what extent it adds to the overall performance of a serverless API.

4.4.2 Pure Lambda Warm Execution

Having explored the cold start times which much greater success in the previous tests? – it was becoming clearer that there is certainly a clear relationship between the memory allocation and the cold start times. It is however also important to consider the impact the choice of runtime has in general with regards to serverless architecture. As a result, this section will evaluate the results of a similar test to the previous one – a direct invocation of lambdas across all three runtimes except evaluation the warm execution times as opposed to the cold start execution times. Further graphs relating to these tests can be located in Appendix A.

4.4.2.1 Java

In the warm test, Java performed exceptionally well in contrast to its cold-start results. One result was recorded as executing in under nine milliseconds which is incredible performance given the results witnessed up until now. Unfortunately, even with a warm container the Lambda could not execute on the 512-megabyte memory tier. It should be observed that whilst memory allocation had a drastic impact on the cold start time – this isn't as apparent in a warm container given the comparatively low times. Table 8 below provides and overview of the test results but it's clear the when running in a warm container, Java exhibits excellent performance.

Table 8 Warm Execution Times Java (Direct Invocation)

Memory Tier	**Average**	**Median**	**Min**	**Max**	**Std. Dev.**	**99p**
1024 MB	18ms	14ms	9ms	68ms	10ms	45ms
2048 MB	14ms	12ms	10ms	39ms	5ms	30ms

4.4.2.2 Node

Node again performed much better in a warm container and saw similarly low times to Java in terms of average duration and execution. It clear that at these extremely low times, the impact of the allocation of memory isn't realised as much as it is with cold starts. Table 9 illustrates just how close the durations become at these extremely units of measurement – an average difference of on seven milliseconds between the largest and smallest memory allocations.

Table 9 Warm Execution Times Node (Direct Invocation)

Memory Tier	**Average**	**Median**	**Min**	**Max**	**Std. Dev.**	**99p**
512 MB	29ms	27ms	12ms	199ms	16ms	65ms
1024 MB	23ms	23ms	7ms	90ms	9ms	46ms
2048 MB	22ms	21ms	7ms	73ms	8ms	39ms

4.4.2.3 Python Warm Execution

Python provided the most unanticipated results in this test, exhibiting extremely similar low results to Java and, in some instances, outperforming Java altogether. This is all the more surprising given the fact that Python, in theory, should be slower given its interpreted nature. On the contrary, Python performed extremely well as outlined in Table 10. Whilst there are only a few milliseconds in these times, this has a larger impact when taken into consideration with the wider cold start times – with Python showing a drastically faster cold start time versus Java. It has become evident that Java also has some overheads with regards to memory which haven't been witnessed in Python – as clearly demonstrated by the test results.

Table 10 Warm Execution Times Python (Direct Invocation)

Memory Tier	**Average**	**Median**	**Min**	**Max**	**Std. Dev.**	**99p**
512 MB	15ms	12ms	8ms	81ms	9ms	50ms
1024 MB	12ms	9ms	8ms	45ms	5ms	25ms
2048 MB	11ms	11ms	10ms	46ms	3ms	24ms

5 Conclusions and further work

The aim of this project was to ascertain the impact the choice of runtime has on cold starts within Amazon Web Services' serverless compute product, Lambda. An API was developed across the three most popular runtimes as identified in the literature review. These API's were developed and subsequently deployed to AWS using the Serverless Framework. A varying range of semi-automated tests were designed and developed in order to explore the correlation between runtime, cold starts, warm starts and how this all can be affected by memory allocation.

5.1 Memory Allocation

It is self-evident from the statistics outlined in the evaluation section that there is a clear and demonstrable link between memory allocation and the impact this has on diminishing cold start times. That is to say that, as more memory is added to a function, the cold start times are in turn proportionally reduced.

This is important to note on a number of levels. Firstly, whilst developers must also consider the memory their functions will consume for regular execution, they must also consider the potential impact any cold starts would have on their wider architecture depending on the particular domain and use-case. In the tests carried out by this project, it was clear that although the functions were reported, from the logs, to be consuming much less memory than the allocated memory tier – this isn't necessarily indicative of the operation of the function itself. The Java runtime seemed to suffer from this issue the most – especially with regards to the 512-megabyte tier which consistently timed out across all tests. Lastly, it is quite evident that increasing memory also increases performance. On the direct invocation tests, there was a discernible improvement of performance as memory was increased and having already alluded to the fact that functions were reportedly consuming much less memory than allocated to them, this may suggest some other factor other than memory.

Work in this area could be done to establish the link further, not only with memory, but perhaps in the wider context of container resources. As memory is increased, could there also be an increase in processing power or performance. There may be other resource upgrade too.

5.2 Triggering Services

As outlined in the introduction, Lambda functions can be triggered by an increasing number of services within the AWS ecosystem. From S3 bucket image uploads to queue message – triggers can vary depending on the particular need of the wider project and domain.

In this project's initial design the testing artefact was a RESTful API which would receive requests initially via API Gateway and route these appropriately to the correct function. Whilst this worked well in a normal scenario with warm lambdas, in a cold start environment this appeared to cause a significant overhead to request's in general. It should be noted that there was potentially some network congestion in the LAN used to carry out the tests but it is thought that API Gateway certainly contributed to these unexpected results. A more controlled environment with which to monitor the environment variables would certainly improve the quality of research, especially with regards to network latency and congestion.

Given the way in which the underlying Lambda platform seems to be architected, with the use of on-demand containerisation, it could be said that the API Gateway is architected in a

similar way – destroying containers when idle in order to optimise performance in the wider platform. There could be the potential for a double cold start in our design – the first as a request reaches API Gateway, and another as the request is routed to a cold Lambda function.

This could be explored further in order to ascertain if API Gateway exhibits the same behaviour. If so, identifying key factors in its cold start could be used in order to streamline the whole process end to end through each service.

5.3 Runtime

Throughout the test phase of this project, it has become obvious that the choice of runtime clearly has an impact on a number of areas within serverless performance. The most obvious being the impact this has on the cold start times. Whilst the interpreted languages – Node and Python - still had an overhead with regards to cold start, it was nowhere near as large as the compiled Java cold start. Even at the lower end of the scale on the default tier of 1024 megabytes, four seconds would be serious cause for concern in a number of use-cases. In particular anything which involves direct interaction with users – API's, time sensitive image processing etc. Whilst this would potentially only affect a small number of users, the issue would be compounded as and when the application scales. In a situation where responsiveness is paramount, based on the research by this project, a compiled language like Node or Python would be a more sensible choice.

Whilst the warm results for Java and Python were comparable in this project, the Lambda's themselves are not consuming a great deal of computational power as such – at most, some string manipulation and interactions with other services like Dynamo DB. Further research into both Python and Java would be extremely beneficial for this space in order to see how the performance fairs as the computational load and complexity increases. It would be a fair hypothesis to predict that given the compiled nature of Java, it would begin to outperform Python as complexity increases. In these instances, there would be relevant use-cases where the cold start becomes less of an inhibiting factor. Post-processing or archiving of audio or video for example, which is triggered on a schedule basis, would not be compromised by a cold start and conversely would benefit from a faster processing time. Cost has not been a consideration in this project but the aforementioned scenario would be a prime example of where faster processing power at the cost of a slower cold start could actually reduce cost over a period of time.

Whilst Node, Python and Java were used in this project – there are a number of available runtimes in AWS Lambda which is ever-increasing. Broadening this project to include a wider balance of both compiled and interpreted language would no doubt yield further insights.

6 References

About Node Package Manager (2018). Available at: https://www.npmjs.com/about (Accessed: 15 November 2018).

Adzic, G. and Chatley, R. (2017) 'Serverless computing: economic and architectural impact', in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. New York, New York, USA: ACM Press, pp. 884–889. doi: 10.1145/3106237.3117767.

AWS Lambda (2017). Available at: https://aws.amazon.com/lambda/ (Accessed: 15 November 2018).

AWS Lambda Limits (2018). Available at: https://docs.aws.amazon.com/lambda/latest/dg/limits.html (Accessed: 15 November 2018).

Bardsley, D., Ryan, L. and Howard, J. (2018) 'Serverless Performance and Optimization Strategies', in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, pp. 19–26. doi: 10.1109/SmartCloud.2018.00012.

Castro, P. *et al.* (2017) 'Serverless Programming (Function as a Service)', in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 2658–2659. doi: 10.1109/ICDCS.2017.305.

Chen, R., Li, S. and Li, Z. (2017) 'From Monolith to Microservices: A Dataflow-Driven Approach', in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, pp. 466–475. doi: 10.1109/APSEC.2017.53.

Chromium V8 Documentation (2017). Available at: https://v8.dev/docs (Accessed: 15 November 2018).

Cisco Global Cloud Index: Forecast and Methodology, 2016–2021 White Paper - Cisco (2018). Available at: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html#_Toc503317522 (Accessed: 9 April 2018).

Cui, Y. (2017) *Comparing AWS Lambda performance when using Node.js, Java, C# or Python*. Available at: https://read.acloud.guru/comparing-aws-lambda-performance-when-using-node-js-java-c-or-python-281bef2c740f (Accessed: 14 November 2018).

Eivy, A. (2017) 'Be Wary of the Economics of "Serverless" Cloud Computing', *IEEE Cloud Computing*, 4(2), pp. 6–12. doi: 10.1109/MCC.2017.32.

Heo, J. *et al.* (2016) 'Improving JavaScript performance via efficient in-memory bytecode caching', in *2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*. IEEE, pp. 1–4. doi: 10.1109/ICCE-Asia.2016.7804810.

Ishakian, V., Muthusamy, V. and Slominski, A. (2018a) 'Serving Deep Learning Models in a Serverless Platform', in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, pp. 257–262. doi: 10.1109/IC2E.2018.00052.

Ishakian, V., Muthusamy, V. and Slominski, A. (2018b) 'Serving Deep Learning Models in a Serverless Platform', in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, pp. 257–262. doi: 10.1109/IC2E.2018.00052.

Janakiram, M. (2014) *Amazon EC2 Container Service and Lambda Service Launched at AWS re:Invent 2014*. Available at: https://www.infoq.com/news/2014/11/aws-ecs-lambda (Accessed: 15 November 2018).

Jing Zhang, Hongxia Luo and Xueqing Zhang (2011) 'Application of python language and arcgis software in RS data management', in *2011 International Conference on Remote Sensing, Environment and Transportation Engineering*. IEEE, pp. 96–99. doi: 10.1109/RSETE.2011.5964225.

Liang, L. *et al.* (2017) 'Express supervision system based on NodeJS and MongoDB', in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*. IEEE, pp. 607–612. doi: 10.1109/ICIS.2017.7960064.

Lynn, T. *et al.* (2017) 'A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms', in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, pp. 162–169. doi: 10.1109/CloudCom.2017.15.

Newman, S. (2016) *Building microservices*. O'Reilly Media. Available at: https://books.google.co.uk/books?hl=en&lr=&id=jjl4BgAAQBAJ&oi=fnd&pg=PP1&ots=_AIQUrf0fQ&sig=EkfgsbGo3gZ5cdqYEY1ad_K0VN4&redir_esc=y#v=onepage&q&f=false (Accessed: 20 April 2018).

Oakes, E. *et al.* (2017) 'Pipsqueak: Lean Lambdas with Large Libraries', in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, pp. 395–400. doi: 10.1109/ICDCSW.2017.32.

Oracle Corporation (2018a) *Design Goals of the Java TM Programming Language*. Available at: https://www.oracle.com/technetwork/java/intro-141325.html (Accessed: 18 November 2018).

Oracle Corporation (2018b) *Go Java*. Available at: https://go.java/index.html (Accessed: 16 November 2018).

Oracle Corporation (2018c) *History of Java Technology*. Available at: https://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html (Accessed: 18 November 2018).

Sill, A. (2016) 'The Design and Architecture of Microservices', *IEEE Cloud Computing*, 3(5), pp. 76–80. doi: 10.1109/MCC.2016.111.

Silva, A. G. *et al.* (2003) 'Toolbox of image processing using the Python language', in *Proceedings 2003 International Conference on Image Processing (Cat. No.03CH37429)*. IEEE, p. III-1049-52. doi: 10.1109/ICIP.2003.1247428.

Spinellis, D. (2005) 'Java Makes Scripting Languages Irrelevant?', *IEEE Software*, 22(3), pp. 70–71. doi: 10.1109/MS.2005.67.

The State of the Octoverse, GitHub (2018). Available at: https://octoverse.github.com/projects#languages (Accessed: 17 November 2018).

Tilkov, S. and Vinoski, S. (2010) 'Node.js: Using JavaScript to Build High-Performance Network Programs', *IEEE Internet Computing*, 14(6), pp. 80–83. doi: 10.1109/MIC.2010.145.

Treat, T. (2015) *Everything You Know About Latency Is Wrong*. Available at: https://bravenewgeek.com/everything-you-know-about-latency-is-wrong/ (Accessed: 16 November 2018).

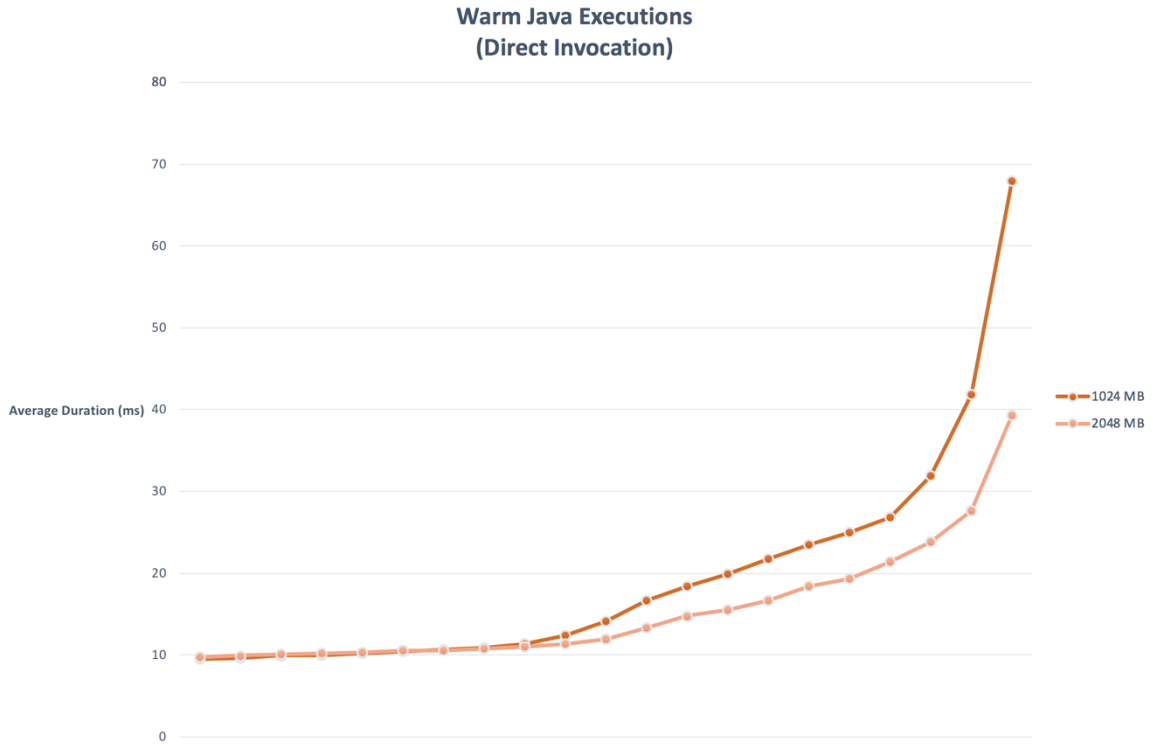
Waseem, M. and Liang, P. (2017) 'Microservices Architecture in DevOps', in *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*. IEEE, pp. 13–14. doi: 10.1109/APSECW.2017.18.

Xiong Zhen-hai and Yang Yong-zhi (2014) 'Automatic updating method based on Maven', in *2014 9th International Conference on Computer Science & Education*. IEEE, pp. 1074–1077. doi: 10.1109/ICCSE.2014.6926628.

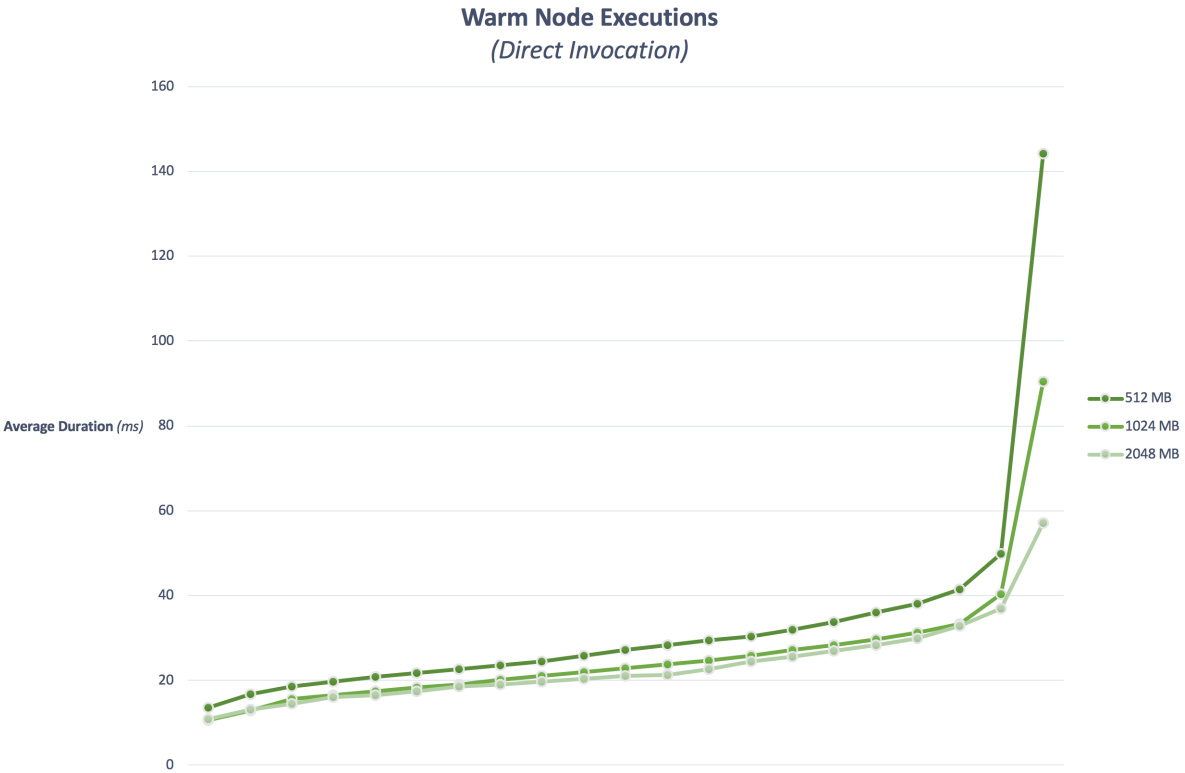
Zhi Lin, Y. (2018) *Comparing AWS Lambda performance of Node.js, Python, Java, C# and Go*. Available at: https://read.acloud.guru/comparing-aws-lambda-performance-of-node-js-python-java-c-and-go-29c1163c2581 (Accessed: 25 October 2018).

Appendix A

Java – Sorted warm execution times

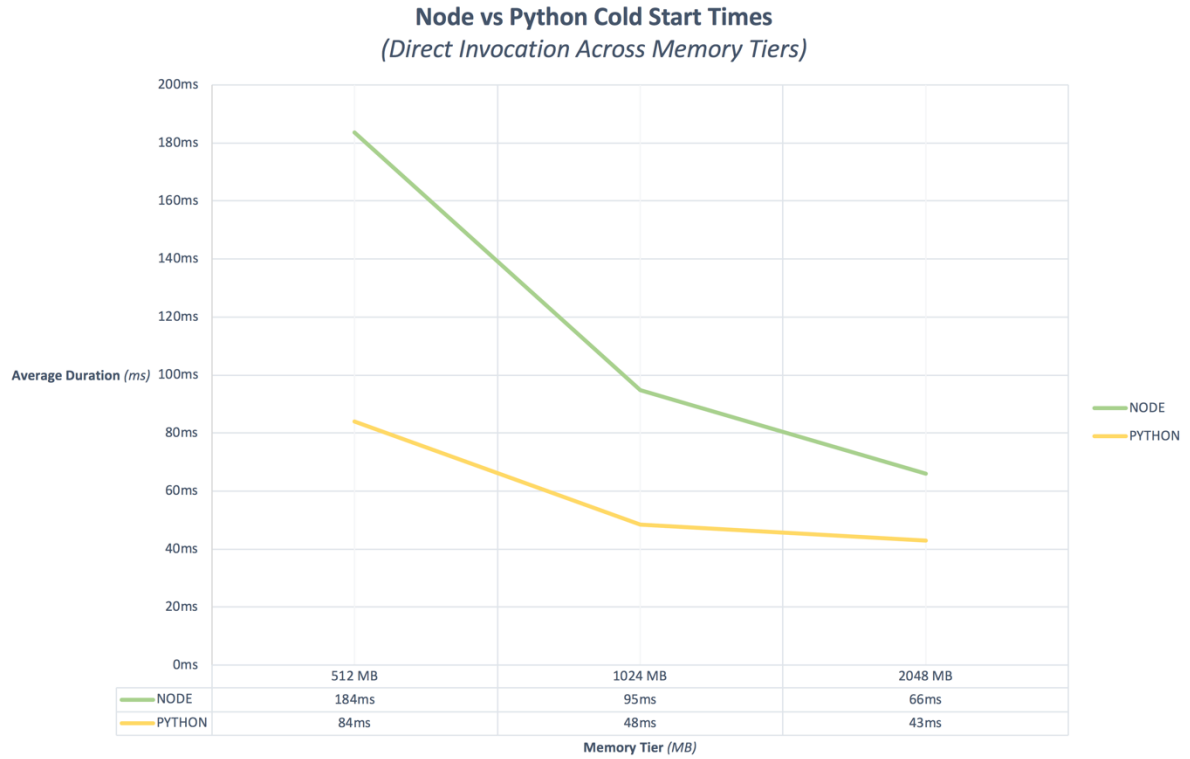


Node – Sorted warm execution times



Python – Sorted warm execution times

Node vs Python Cold Start Times



Appendix B
Code Listings

GitHub Repository

Full code listings and result logs can be found on GitHub
https://github.com/cjconnor24/Amazon-Lambda-Honours-Project

OneDrive Shared Link

Full codebase can be downloaded from OneDrive via the following link
http://tinyurl.com/S1715477