Expertise
and insight
for the future

Sachin Shrestha

Comparing Programming Languages used in AWS Lambda for Serverless Architecture

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

19 June 2019

Author Title Number of Pages Date	Sachin Shrestha Comparing Programming Language used in AWS Lambda for Serverless Architecture 45 pages + 1 appendices 19 June 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of Department (ICT) at Helsinki Metropolia University of Applied Sciences Natalia Garnova, Team lead of activation team at Zervant Oy

The primary purpose of this thesis was to explore AWS Lambda service and its features provided by Amazon.com, Inc. The benefits and importance of this service which adapts the principles of serverless computing are discussed. This service has been used in case company to manage infrastructure that hosts and executes codes. The task of such service includes sizing the available provision and scale multiple servers while managing operating system updates, apply security patches and then monitor all the resources for performance and availability.

The secondary purpose is to research in detail about the native programming languages supported by AWS Lambda and their respective implementation methods. They are popular languages in technological field and widely implemented everywhere. These languages are compared under the characteristics of availability of compilers and tools, reusability, efficiency, familiarity, reliability and readability. The specification of each language is studied, and the project report is constructed by providing same test case for all natively supported language separately multiple times.

The analytics result of individual performance under project case was documented and observed to form a general conclusion. It was observed that there is subtle difference in performance among the language when tested in common environment. In terms of speed of execution, interpreted languages performed better than compiled language. While performing the cold start process, C# was slower than other languages, but the difference was in milliseconds. Java had the largest package size compared to others.

This thesis can be useful to get detailed information about AWS Lambda service and know why it is an important realization of serverless architecture. This study also discusses the benefits of serverless architecture over traditional systems on the grounds of efficiency, performance and cost of production and maintenance. The options to integrate AWS Lambda with other services provided by AWS is the main reason for its popularity. The language can be chosen depending upon the requirement of application service model and developer's preference.

Keywords	AWS Lambda, Serverless Architecture, Interpreted and Compiled Programming Languages

Contents

List of Abbreviations

AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
CPU	Central Processing Unit
CSV	Comma Separated Values
CSP	Communicating Sequential Processes
DLL	Dynamic Link Library
DSL	Domain Specific Language
EC2	Elastic Compute Cloud
ENI	Elastic Network Interface
FaaS	Function as a Service
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
JSON	JavaScript Object Notation
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JVM	Java Virtual Machine

.NET	Network Enabled Technologies
OS	Operating System
PaaS	Platform as a Service
RAM	Random Access Memory
RDS	Relational Database Service
REST	Representational State Transfer
S3	Simple Storage Service
SAM	Serverless Application Model
SDK	Software Development Kit
SDLC	System Development Life Cycle
SNS	Simple Notification Service
SQS	Simple Queue Service
VPC	Virtual Private Cloud
XML	Extensible Markup Language

1 Introduction

This thesis explores the AWS Lambda, which is one of many services provided by Amazon.com, Inc. This service is a serverless computing platform based on event-driven architecture. It responds to events provided by the code and efficiently allocates the available resources. The service was first announced and launched on November 2014 at AWS conference. Since then, it has been evolving with continuous development and added features corresponding to market needs.

The focus for the development of this service was to create a platform that handles the infrastructure for running other web services without the need of resources that they individually require to operate. Such client web services only need to maintain transaction of events to remote cloud servers and run without provisions for local hardware resources. In this setup, the local applications will run within milliseconds after being activated by events. Since the remote servers are capable of handling thousands of such events at the same time, various asynchronous functions of application can work in parallel.

This study was carried out for Zervant Oy, later referred to as ZERVANT, with prime purpose to research best possible way to implement AWS Lambda services to manage its already existing services. This research is going to provide insights about pros and cons among several methods of implementation of the service. Only few features use this service currently and there are many areas where it can be implemented for better performance.

Zervant was founded by Mattias Hansson and Tuukka Koskinen in 2010. Since the establishment, the company has experienced and has been focused on making Europe's best invoicing software. Finland, Sweden, the UK, France, Germany, Belgium and Austria are currently the core market and the company has plans to expand it further. This company is focused on only one product, the modern and dynamic web-based invoicing software with primary goal to be financial tool for small scale businesses. The main motto of company is to help the entrepreneurs to succeed.

There are several teams specializing in various fields like marketing, customer support, product development, design and management. Each team must do this part according to their expertise and contribute on over-all collective progress of company. The teams also follow modern organizational practices to constantly give desirable output which will have impact on collective performance. Every employee in the company are experts in their respective field and be motivated to fulfill the goal and objective of the company. Since the company is in the growth phase, it is constantly hiring new people who can fit into the positions required to scale the development.

The software itself has gone through multiple changes to match the requirements of the market and clients. The company is proud to have amazing software and world-class team whose credit goes to efficient planning and execution. The major achievement milestones are receiving international investments of more than 14 million Euros till date and being voted the most promising fintech company in the Nordics. The company still has further plans to add more features to the software and strive for bigger market with the goal to become best invoicing software.

To meet the goals within designated period, the product goes through continuous development and integration. The company also works closely with clients by assisting them and collecting their feedback to plan the improvement of the product. Most of the product development work focuses to improve user experience and usability of the software.

This thesis is structured in five sections and the first one includes introduction to subject of thesis. It contains general insights upon purpose of thesis with introduction to ZERVANT, whose case was used for the research. ZERVANT is a Nordic market-leading company providing invoicing service through web platform. Section two focuses on AWS Lambda service and its detailed structure which provides the background for this research. The basic principal of AWS Lambda service and its role in serverless architecture development is briefed in this section. The third part of this study approaches the information about native languages supported by AWS Lambda and their specific overview of programming patterns. The fourth section is a project work report. In this part, an example test case is formulated and implemented and analytics data is collected. Based on the results, general observations are made which are related to the thesis topic. The last section contains conclusion upon the topic and summary of the overall study.

2 AWS Lambda

AWS Lambda is a computing service that allows developers to operate software without supplying or maintaining servers. It only executes the code when necessary and automatically scales from a few requests a day to thousands a second. Only the computer period is charged, and no fee is applied if the software does not run. AWS Lambda provides the infrastructure to upload the code. It keeps the code and triggers the code whenever the required event occurs. It allows you to select the memory and timeout needed for the code.

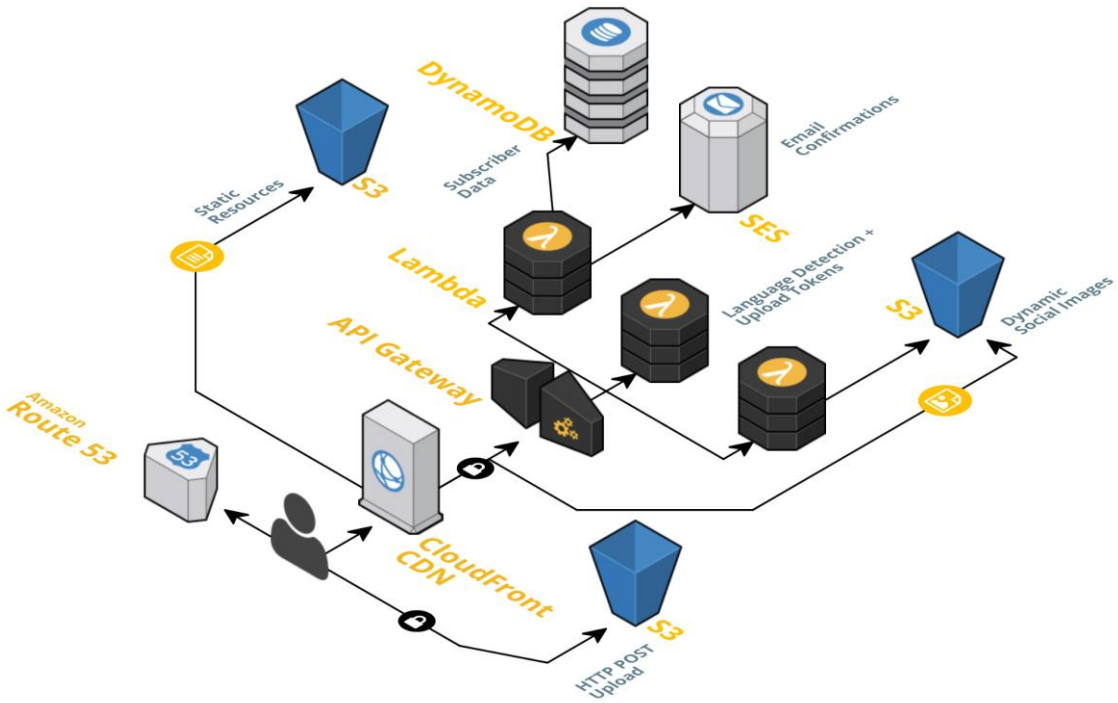

Figure 1. Typical architectural setup for serverless with AWS Lambda.

The programmer can upload a software feature to AWS with Lambda and then perform it on AWS. Their own EC2 cases must no longer be provided, they should only concern themselves with the code. AWS Lambda is designed for short run mostly up to 60 seconds, full application web server cannot be generated by Lambda. However, AWS provides many integration hooks: for instance, AWS can trigger a Lambda feature whenever an object is added to S3 or when a new message is received on the SQS, a Lambda function is triggered. Amazon API Gateway allows the execution of a REST API without

any EC2 instances. You can specify that whenever a GET /some/resource request is received, it will trigger a Lambda function. The coder can define that a Lambda feature will be triggered whenever a GET / some / resource request is obtained. The Lambda and Amazon API Gateway blending allows to create strong facilities without having to keep a single EC2 instance. (Wittig 2016: 343)

AWS Lambda is distinct from a physical or virtual server-based traditional strategy. Only the logic of the developer is needed, grouped in functions, and the service itself is responsible for executing the functions, if necessary, by managing the software stack used by the runtime selected, the platform's availability, and the infrastructure's interoperability to sustain the invocation's latency. In containers, functions are performed. Containers are a method of server virtualization where multiple isolated environments are implemented by the OS kernel. Although, physical servers still execute the code, but since no time is wasted handling it, this kind of strategy is commonly defined as serverless. (Poccia 2016)

Serverless architectures are the recent progress in thinking, studying, and adopting by designers and organizations. To achieve a competitive benefit, software architect needs to think about how they can maximize the use of cloud systems. Serverless architectures are the latest advance for developers and organizations to think about, study, and adopt. For IT infrastructure and application software, the cloud has been and remains to be a game changer. This fascinating new paradigm shift in design will develop rapidly as computer services such as AWS Lambda are embraced by software designers. And, in many instances, apps without servers will be easier to operate and quicker to execute. Complexity and expenses connected with operating facilities and developing traditional computer technologies must be reduced. Minimizing costs and time spent on maintaining services and the advantages of optimization are excellent factors to consider serverless architectures for organizations and developers. (Sbarski 2017: 15)

2.1 Background

While EC2 stays one of the most commonly used fundamental AWS facilities, it is not yet intended to manage or react to events; something that is needed in today's apps often. For instance. a normal picture upload activity to an S3 bin, for instance, causes

some type of procedure, such as checking whether the item is a true picture, or whether it includes any bugs or unnecessary malware. There might also be cases where thumbnails of the uploaded image need to be created and placed on the website. If EC2 instance is used for doing all these activities, it would have to program some mechanism for S3 to notify current EC2 instances to periodically perform checks on current S3 bucket. EC2 alone has no way of telling when a new object has been uploaded. AWS specifically launched Lambda to respond to these kind of specific problems (Wadia & Gupta 2017: 8-11)

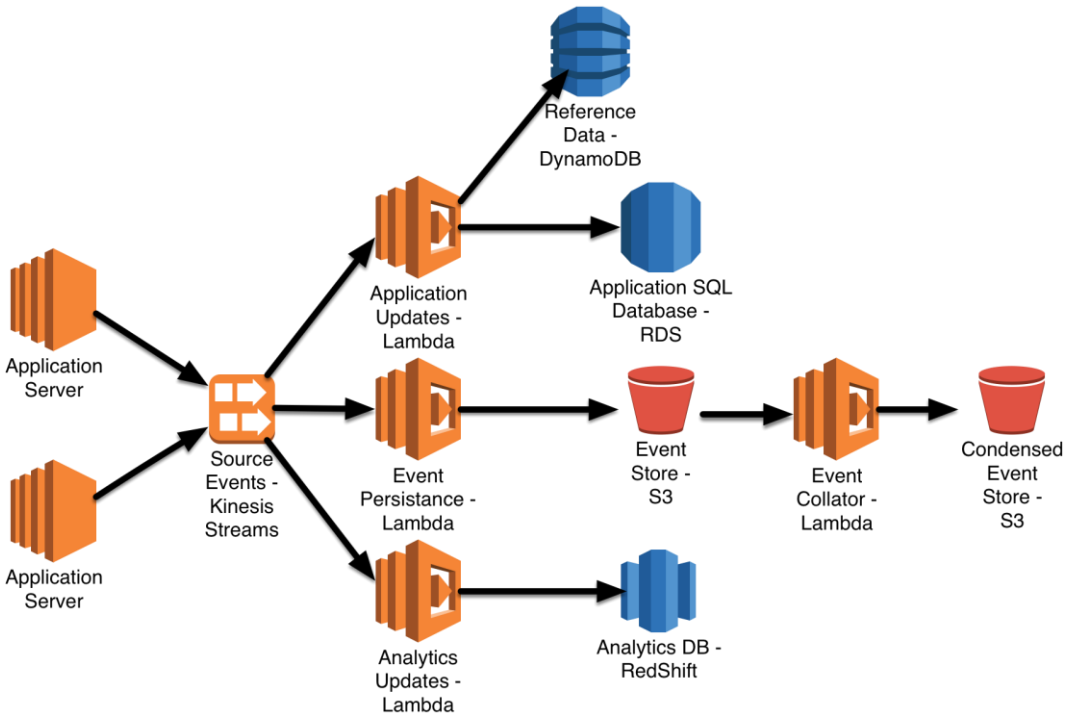

Figure 2. Infrastructures services provided by AWS for serverless architecture.

Serverless architecture enables developers to concentrate instead of infrastructure on computer design and code. It is simpler to attain scalability and elevated accessibility. The distribution is often fairer because the user of this service only pays for what they use. Importantly with serverless, by minimizing the number of nodes and quantity of code needed, the developer has the ability to decrease some of the system's complexity. (Sbarski 2017: 14-15)

This service can also be used to build services with flexible scaling and fault tolerance. Deployments of an application are allocated by default across various available areas. Since a single error point is redacted, the application becomes more tolerant of such errors. No Idle Capacity. You pay only for the total invocations of your function; plus, the time the function is running. Capacity management and elastic scaling are automatically handled by the service which will reduce the cost of implementation of features in the application. These benefits directly effect on reduced infrastructure & operational costs, shorter time-to-market, better service stability, less waste, and increased flexibility.

One benefit of the serverless strategy is that it is uncomplicated to gradually covert current apps to serverless architecture. If a developer faces a foundation of monolithic software, they can gradually separate it and generate Lambda features with which the software can interact. Initially, the easiest strategy is to build a prototype to test developer expectations as to how the scheme would work if it were to be partially or completely serverless. Legacy schemes tend to have exciting limitations that involve innovative alternatives; and there will certainly be tradeoffs as with any architectural refactors on a big scale. The scheme may end up being a hybrid, but it may be better to have some of its parts use Lambda and third-party infrastructure instead of continuing with an untouched heritage design that no longer scales or needs costly services to function. It may take time to get correct from a classic server-based implementation to a scalable serverless design. It requires to be approached cautiously and smoothly, and before they start designers need a strong sample scheme and a great DevOps approach in place. (Sbarski 2017: 12)

When writing code for Lambda, it's important to understand the fundamental precept: the code can't create state assumptions. This is because when a new function container is first developed and activated, Lambda completely handles this. A container may be activated for various purposes such as events causing the Lambda function increase in concurrency beyond the number of containers initially produced for the feature, an event triggers your Lambda function in several minutes for the first time, etc. While Lambda is accountable for scaling up and down the service containers, the software requires to be prepared to operate correctly. While Lambda is accountable for scaling up and down the service containers the software requires to be prepared to operate correctly. The code cannot create any expectations that state from one invocation to the next will be

maintained. However, it stays active and accessible for later invocations for at least a few minutes before it is terminated every time a feature container is generated and invoked. When consecutive invocations take place on a container that has been active and invoked at least once before, it can be claimed that invocation runs on a warm container. Whenever there is an invocation for a Lambda feature requiring the creation and first invoking of your feature application bundle, the invocation is having a slow beginning. (d1.awsstatic.com 2017: 8)

2.2 Basic Principles of AWS Lambda

AWS Lambda is an optimal computation platform for many implementation situations, as long as the application script is written in languages endorsed by AWS Lambda and operate within Lambda's normal runtime setting and assets. It is a computing service that allows the programmer to operate software without supplying or maintaining servers. AWS Lambda only executes the code when necessary and automatically scales from a few demands per day to thousands per second. For virtually any type of application or backend service, anyone can run code with AWS Lambda without any administration. AWS Lambda operates the code on a server infrastructure with wide accessibility and conducts all computer resources management including server and working system servicing, power provisioning and automatic scaling, code tracking and logging. (docs.aws.amazon.com 2019: 1)

The cloud has been and remains a game changer for the growth of IT infrastructure and applications. To achieve a competitive benefit, software designers need to think about how they can maximize the use of cloud systems. Serverless architectures are the recent progress in thinking, studying, and adopting by designers and organizations. This interesting new change in architecture will develop rapidly as software services such as AWS Lambda are embraced by software designers. And, in many cases, it will be cheaper to run serverless applications and quicker to implement. Intricacy and expenses connected with operating facilities and developing traditional software technologies also need to be reduced. Cutting down costs and time spent on maintaining infrastructure and the advantages of scalability are excellent factors to consider serverless architectures for companies and developers. (Sbarski 2017: 15)

Lambda is a function-based, high-scale, provision-free server-free computation service. It offers your request with the cloud logic layer. A range of events occurring on AWS or promoting third-party facilities can trigger Lambda functionalities. It enables the developer to build reactive, event-driven systems. Lambda effectively performs more copies of the process in parallel when there are numerous, concurrent events to react to. Lambda features scale up to the client requirement with exactly the magnitude of the workload. Thus, there is an exceptionally small probability of getting an inactive server or container. Using Lambda features, architectures are intended to decrease waste ability. Lambda can be defined as a Function-as - a-Service (FaaS) serverless form. FaaS is one strategy to constructing computational systems driven by events. It depends as the execution and integration unit on features. Serverless FaaS is a type of FaaS where the programming model does not include virtual machines or containers and where the vendor provides provision-free scalability and built-in reliability. (d1.awsstatic.com 2017: 2)

2.2.1 Cloud Computing

Cloud computing or cloud computation is a synonym for IT resources production and usage. The cloud's IT resources are not immediately apparent to the consumer; abstraction levels are in between. The cloud's proportion of abstraction can differ from virtual systems to complicated distributed structures. Resources are accessible in vast quantities on request and are charged for each use. Amazon Web Services (AWS) is a web services platform that provides alternatives for computing, storing, and networking across distinct abstraction levels. These services can be used to host websites, operate business apps, and mine huge quantities of information. The term web service implies that a web interface can be used to control services. Systems or individuals can use the web interface via a graphical customer interface. The most prominent services are EC2, delivering virtual servers, and S3, providing storage ability. AWS services function well together; replicating the currently existing configuration or designing a fresh configuration from scratch can be done using them. Services are paid on a sales system for pay-per-use. (Wittig 2016: 3-4)

The official definition from the National Institute of Standards and Technology:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. (Wittig 2016: 5)

Cloud computing is one of the technologies that are redefining application development and delivery. Many teams and initiatives strive to utilize fresh technology and sometimes fail. The main reason for inability is to apply a radically distinct technique to the present implementation design and programming model. Well-designed, deployed, and supplied cloud-based apps differ radically from traditional implementations. Technologies such as platform as a service (PaaS) and containers have emerged and discussed over the previous few years as prospective alternatives to the headache of incompatible storage settings, disputes, and server governance. PaaS is a cloud computing method that offers people with a platform to operate their applications while hiding some of the infrastructure underlying it. (Sbarski 2017: 12)

Clouds can be differentiated in many types. One of them is public cloud that is managed by an organization and accessible to the general public. Another type is called private cloud which virtualizes and shares the IT resources within a single organization Combining these both type of cloud forms a hybrid cloud. AWS cloud is one of the examples of such hybrid cloud. It offers basic resources such as computing, storing, and networking capabilities, using virtual servers such as Amazon EC2, Google Compute Engine, and Microsoft Azure Virtual Machines and platforms for deploying custom cloud applications. (Wittig 2016: 5)

The AWS Cloud offers many distinct facilities that can be serverless implementation elements. The major capabilities include computation, storage, inter-process messaging, orchestration, serverless architectures and analytics. (d1.awsstatic.com 2017: 1)

2.2.2 Serverless Architecture

The direct descendants of service-oriented design are microservices and serverless architectures. They maintain many of the values and thoughts mentioned above while

trying to address the complexity of old-fashioned service-oriented architectures. A latest trend has been in the implementation of microservices systems. Developers tend to perceive microservices as tiny, autonomous, completely autonomous facilities constructed around a specific company function or capacity. Theoretically at least, it should be simple to substitute microservices with each service published in a suitable structure and language. An attention drawing point for many developers is the absolute reality that microservices can be published in various general-purpose or domain-specific languages (DSL). Using the correct language or a specific library for the work can provide benefits. However, it can often also be a trap. Having a combination of languages and frameworks can be difficult to sustain and can lead to confusion on the long run without rigorous discipline. (Sbarski 2017: 6-7)

The big distinction between traditional cloud computation and serverless computation is that the developer — the consumer who needs such computing — does not pay for unused resources. In the past, the engineer had to anticipate and prepare for capability and resource demands, whether in local data center or in the remote cloud. However, with the use of serverless setup, the developer only directs the cloud provider to spin some time of code execution when the function is actually called. The FaaS service requires the tasks of the developer as input, logically works, returns the yield and then shuts down. Only the resources used during the real processing of those tasks are charged to the designer. Serverless is a microservice architecture's next development. Cloud providers are essentially taking what were best practices with containers and Docker, while enforcing them as part of the serverless model. The four factors from the tradition application has been migrated to new serverless platform which are processes, concurrency, disposability and logs as event streams. (Boyd et al., 2018)

There are many perks of using the serverless architecture. The developer doesn’t need to be concerned about all the server-related issues like runtime patching and OS configurations. They don’t have to worry about deploying, managing hosts and server management. In reaction to the load, the facilities scale up or down or depending on the designated unit job capability. In reaction to the load, the facilities scale up or down or depending on the designated unit job capability. The programmer no longer needs to use parameters such as memory ability or CPU cores to scale. Instead, they indicate the complete quantity of job to be done.

This has huge consequences on reducing overall cost production since the user is not liable to pay for idle compute capacity of infrastructure. Operating expenses are also decreased as technicians no longer have to care about server-related problems (such as server provisioning and server safety), fine tuning host settings, creating machine images, computer pictures, and application deployments on particular server. Since the bunch of routine infrastructure task are offloaded, the developers can concentrate their engineering resources on monitoring app performance from a business value perspective.

Most of today's web-enabled computer-controlled system has backend servers performing multiple types of computing and client-side front ends offering customers with an interface for their browser, mobile or desktop computer to perform. The server accepts HTTP requests from the front end and processes requests. Before being saved to a database, data could pass through various processing layers. Finally, the backend produces a response — it might be in the form of a JSON or a completely rendered markup — that is returned to the client. Of course, most applications will become more complicated once components like load balancing, operations, clustering, caching, messaging and information redundancy are taken into consideration. Most of this software needs data center or cloud servers that need to be operated, retained, patched, and backed up. Server provisioning, handling, and patching is a time-consuming job that often involves individuals with dedicated processes. It is difficult to set up and run a non-trivial environment efficiently. Infrastructure and technology are needed elements of any IT system, but they are also often a diversion from what the key focus should be — solving the business issue. (Sbarski 2017: 4)

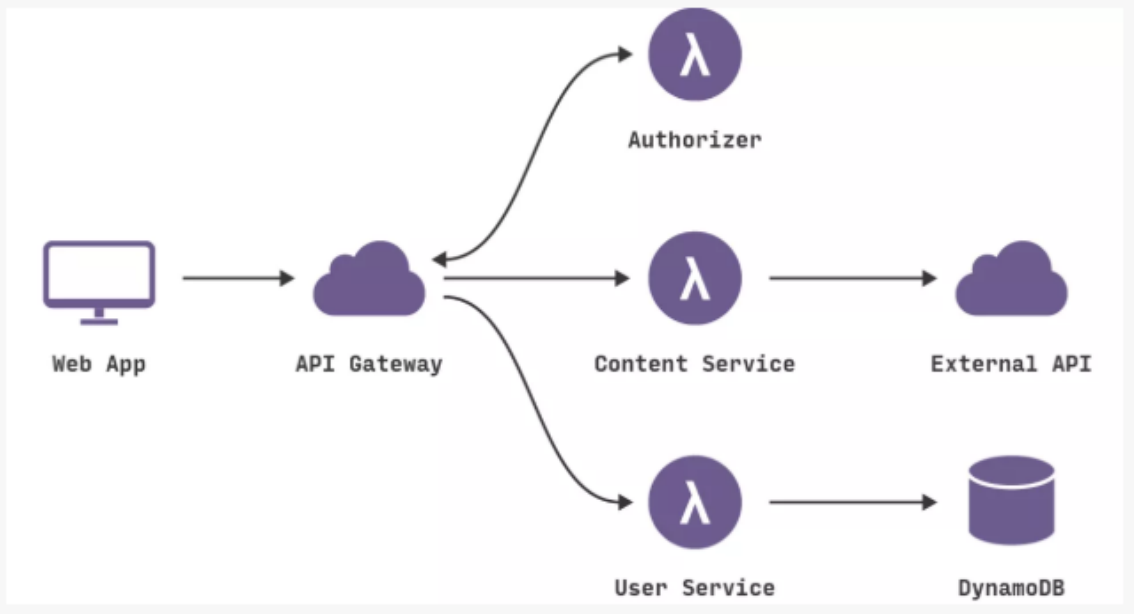

Figure 3. Typical model of serverless architecture.

Serverless architectures can assist with the layering issue and where too many components need to be updated. Developers have space to avoid or minimize layering by splitting the structure into tasks and enabling the front end to interact straight safely with utilities and even the database. All of this can be achieved in a structured manner to avoid spaghetti applications and dependency nightmares by precisely identifying service limits, enabling for autonomous Lambda tasks, and scheduling how features and services will communicate. (Sbarski 2017: 7)

The engineer doesn't have to care about infrastructure configuration and ability in a serverless architecture which implies the user can concentrate completely on product planning application layout, and system development. Serverless is progressively versatile, allowing to quickly deploy extremely scalable apps and readily generate easy prototypes to test application's features. This adds agility to the method of growth and enables to rapidly roll out new features in application. It is simpler to attain scalability and elevated accessibility, and allocation is often fairer.

It is possible to build serverless architectures to serve any objective. To take benefit of this design, systems can be constructed serverless from scratch or current monolithic applications can be reengineered gradually. Event driven systems are the most versatile

and strong serverless models. Building event-driven, push-based systems often reduces costs and complexity and possibly smoother the general user experience. While event-driven and push-based architecture are a decent strategy, in all conditions they may not be suitable or achievable. It may be case that sometimes, the service has to introduce a Lambda feature that will scrutinize the origin of the event or run on a schedule. (Sbarski 2017: 7)

There are many methodologies for SDLC and server-based architecture that are also valid for serverless architectures such as eliminating single error points, pre-deployment test modifications and encrypting delicate data. Because of how distinct the operating system is, attaining best methods for serverless architectures can be a difficult challenge. The AWS Well-Architected Framework involves policies that will assist to compare the workload with best practices and provide recommendations to produce consistent and effective systems. The five pillars of serverless practices are performance efficiency, cost optimization, security, reliability and operational excellence. (d1.awsstatic.com 2017: 10)

2.2.3 Lambda Functions

Lambda functions can be generally defined as the codes that are uploaded in AWS Lambda. Configuration information such as name description, access point, and resource specifications are related to each function. They should be stateless, and it should assume that there is no affinity to the underlying compute infrastructure. The processes and requirements involving those functions should be independent and limited. They should not extend beyond the lifetime of request.

A function is a short code section centered on performing a single job. Functions can be rapidly published and readily modified and substituted afterwards. Like most FaaS systems, AWS Lambda operates tasks as a stateless service, meaning no condition between each invocation function is maintained between the tasks. They don't store information either. If they need to store or access information, they can incorporate via application programming interfaces (APIs) with Database as a service offering or serverless storage facilities. Events occurring in the storage system trigger the deployed features which is automated, and event driven. The service will shut down the function when the

function is not in use, so there won't be any cost charged for a server that is in idle state. (Boyd et al., 2018: 12)

All custom code is produced and performed in serverless design as separate, autonomous, and often granular functions running in a stateless storage service like AWS Lambda. Developers can write functions to perform nearly every normal job, like reading and writing to an information source, calling other functions and carrying out calculations. In more difficult instances, designers can create more elaborate pipelines and organize various function invocations. Some situations may still require a server to do something. However, these instances could be very different, and as a developer, if feasible, they should prevent operating and communicating with a server. (Sbarski 2017: 9)

It is essential to note that it should be fast to perform custom code executing in Lambda. Functions that terminate sooner are less expensive as the pricing of Lambda is determined by the number of requests, the length of execution and the quantity of memory allocated. In addition, constructing an efficient front end (instead of a complicated back end) that can communicate directly to third-party services can lead to a stronger customer experience. Less hops between online resources and less latency will lead to a stronger perception of the application's efficiency and usability. It is not necessary to route everything. The front end can interact with a search vendor, database, or other helpful API instantaneously.

Lambda transports invocation events to the function which is then processed to return a response. AWS Lambda enables the processing of events in the language selected to run functions in a serverless environmentIn an isolated execution context, each instance of the function runs and processes one event at a time. Lambda will automatically scale up the function's number of instances to manage large event volumes. (docs.aws.amazon.com 2019: 5)

There are two methods of generally invoking a Lambda functions. The first model is called Push Model in which Lambda function is invoked every time a particular event occurs within another AWS service. Another is Pull Model in which Lambda polls a data source and invokes the function with any new data coming to the source, batching new records in a single invocation function. Lambda function can be processed both

synchronously or asynchronously. The method of function invocation is defined by each event. It is also the responsibility of the event source to create its own event parameter. (d1.awsstatic.com 2017: 10)

The Lambda runtime system is modeled on an Amazon Linux AMI, so the components that is intended to operate inside Lambda should be compiled and tested within a compatible environment. AWS offers a collection of features called AWS SAM Local to allow local testing of Lambda features to assist conducting this sort of testing before operating in Lambda When invoking the Lambda feature in one of the permitted languages, an event component is one of the parameters supplied to the handler function. Depending on which event source made it, the event varies in composition and content. The event parameter's contents include all the information and metadata that the Lambda function requires to guide its logic. (d1.awsstatic.com 2017: 5)

2.3 Use Cases

AWS Lambda can be used to execute code in response to event triggers like manipulation, changes in system state, or actions created by client. AWS facilities like S3, DynamoDB, Kinesis, SNS, and CloudWatch can immediately trigger Lambda, or AWS Step Functions can initiate it into workflows. This enables a range of serverless information handling technologies to be built in real time. For instance, Amazon S3 bucket can be utilized to trigger AWS Lambda to process data simultaneously after an upload. With this ability Lambda can perform various web services related tasks like indexing files, processing logs, resizing thumbnail images, transcoding videos, validating content, and aggregate and filter data in real-time.

The range for serverless construction is extensive and one of its benefits is that it can be used for both tiny and large functions in a same manner. Serverless techniques and architectures can be used to construct complete systems generate separate parts or perform complex functions. The serverless systems are designed to power web and mobile applications for large number of users at the same time. It can be disintegrated into smaller component systems to solve specific issues. Moreover, Lambda can be integrated with third-party services and APIs to cut down on the amount of work required for production and implementation of web services. (Sbarski 2017: 16)

Another prominent use cases of AWS Lambda is to host S3 static websites. While hosting the web frontend on S3, the acceleration of content delivery should be done with Cloud front caching. In this case, the web frontend can send requests to Lambda functions via API Gateway HTTPS endpoints. Lambda can manage the implementation logic and persist with data to a completely managed database system (relational RDS, or non-relational DynamoDB). To isolate your Lambda features and databases from other networks, developers can host them within a VPC. This setup is highly cost effective because for Lambda, API Gateway and S3, only the traffic volume is charged. The other fixed costs include cost of running the database service only.

Figure 4. Flow of events on image resizing service.

Lambda function can be easily incorporated to check log files from CloudTrail or CloudWatch. Lambda can search for particular events or log entries and submit notifications via SNS in the logs as they happen. Custom notification hooks can be easily implemented by calling their API endpoint within Lambda. Amazon Kinesis Streams can be integrated with lambda to track events such as logs, system events, transactions, or user clicks. Lambda functions can respond in a stream to new records and can rapidly process, save, or delete data. When a particular amount (batch size) of data is accessible for processing, a Lambda feature can be designed to function so that it does not have to perform for every single record added to the stream. Kinesis streams and Lambda functions are a great match for apps that produce a bunch of data to analyze, aggregate and store. The amount of functions created to process messages from a stream when it comes to Kinesis is the same as the number of fragments or shards. In addition, if a batch is not processed by a Lambda function, it will be retried. This can last up to 24

hours if processing fails each time. This combination can be powerful for real-time processing and analytics. (Sbarski 2017: 18)

Other important use of Lambda can be generation of automated backups on everyday tasks. The Lambda functions can be written to perform specific tasks like scheduling and maintain AWS accounts and services. It can be coded to give responses and information about checking idle resources while running web applications within AWS IaaS platform. The lambda function can also work closely with Amazon S3 bucket, which is storage for all kinds of components. Since they both run on top of same layer of resources, the time and costs are reduced dramatically.

A common use for serverless technologies is data processing, conversion, manipulation, and transcoding. The simple uses of AWS Lambda functions can be processing of CSV, JSON, and XML files, collation and aggregation of data, image resizing, and format conversion. Lambda and AWS facilities are well adapted to build pipelines for data processing functions guided by events which sets permissions for files and generates records for metadata. When AWS Lambda is carefully integrated with other services it can give excellent outcomes related to serverless technologies. (Sbarski 2017: 19)

Effective governance of the AWS infrastructure was one of the main and most commonly used use cases for Lambda, primarily around EC2 instances, as this is where most of the expenses are incurred needlessly. Before the introduction of Lambda features, many organizations had to depend on third-party automation instruments and facilities to perform easy and straightforward duties on their instances which were costly and complex to maintain. The most difficulties were related to management of automation. This would have created unwanted overhead to administer the services. But with Lambda, these problems are effectively solved because Lambda enabled developers to breakdown bulky system into simpler functions. These functions are capable to solve complex tasks without the use of third-party tools to handle the system. (Wadia & Gupta 2017: 229)

Lambda features can operate on a timetable to effectively perform repetitive tasks such as data backups, imports and exports, reminders and alerts. Developers can use Lambda functions on a schedule to periodically ping their websites and review their status. The AWS Lambda service also have collection of blueprints for common issues,

which can be used for simpler tasks. Also, it helps to automate several types of repetitive tasks such as file backup and file validation which needed to be done manually before the introduction of Lambda services. (Sbarski 2017: 19)

2.4 Pricing

In a competitive market of technology, cost of production and maintenance of services is the major aspect of development. AWS released Lambda with the goal of cutting down such unnecessary overheads while maintaining best performance. The pricing model of Lambda is base on requests made to server. It counts requests each time the function is invoked by an event. The charge for each such invocation is $0.0000002 per request for the total number of requests across all the functions. This service also offers free tier facility which includes 1M free requests per month and 400,000 GB-seconds of compute time per month. The memory size you chosen to run the Lambda functions determines how long they can run in the free tier. Often the price of operating serverless architecture can be much lower than operating traditional infrastructure. The costs may vary according to the need of resources but still is cheaper than running traditional servers and API gateways or third-party tools.

Lambda offers a single control point to switch up and down the function's quantity usage of compute resources that is the quantity of RAM assigned to the function. The quantity of assigned RAM also affects the function's volume of CPU time and network bandwidth. Choosing the least memory may save the price but that may not be ideal case every time. Lower memory can add latency to the application and deciding to choose several functions to run in low memory resources may be more expensive in many cases. Because Lambda is billed in 100-ms increments, this strategy might not only add latency to your application, it might even be more expensive overall if the added latency outweighs the resource cost savings. The Lambda function should be tested separately each time in available resource levels to determine what the optimal level of price/performance is for the application. The performance of the function should improve logarithmically as resource levels are increased. The logic that is executing will define the lower bound for function execution time. There will also be a resource threshold where there is no significant output gain any extra RAM / CPU / bandwidth applicable to the function. (d1.awsstatic.com 2017: 27)

Length of processing is calculated from the instant that the Lambda function starts running until it starts or ends, rounded up to the closest 100ms. The cost relies on how much memory the function is allocated to. In the AWS Lambda resource model, there is options to choose the amount of memory required for the service instance and are allocated proportional CPU power and other resources. A rise in memory size causes an equal rise in the function's processing power in the CPU. This increase in resources usage also increases the additional costs.

Figure 5. Choosing the optimal Lambda function memory size.

Pricing, however, rises linearly as Lambda's resource concentrations rise. To select the ideal configuration for the function, the tests should discover where the logarithmic function switches. The chart above demonstrates how the optimal distribution of memory to an instance function can enable both greater price and reduced latency. Here, the extra overhead of calculating 512 MB per 100 ms over the lower storage alternatives is outweighed by the quantity of latency decreased in the feature by allocating more resources. After 512 MB, the performance gains are diminished for this function’s logic, so the additional cost per 100 ms now drives the total cost higher. From this calculation, it can be concluded that 512MB is ideal choice for minimizing total cost. (d1.awsstatic.com 2017: 28)

The efficiencies acquired from getting Amazon look after the platform and scale functions when it goes to Lambda come at the cost of being prepared to customize the operating

system or fine tune the inherent instance. The programmer can alter the quantity of RAM assigned to a feature and the timeouts and different third-party services will have varying levels of customization and flexibility. Serverless architecture enables developers to concentrate on service design instead of underlying infrastructure maintenance. It is simpler to attain scalability and elevated accessibility, and cost is often fairer because only used resources are fixed cost. (Sbarski 2017: 19)

The following figure displays the example pricing model for common use case.

Example 1

If you allocated 512MB of memory to your function, executed it 3 million times in one month, and it ran for 1 second each time, your charges would be calculated as follows:

Monthly compute charges

The monthly compute price is $0.00001667 per GB-s and the free tier provides 400,000 GB-s.

Total compute (seconds) = 3M * (1s) = 3,000,000 seconds

Total compute (GB-s) = 3,000,000 * 512MB/1024 = 1,500,000 GB-s

Total compute – Free tier compute = Monthly billable compute GB- s

1,500,000 GB-s – 400,000 free tier GB-s = 1,100,000 GB-s

Monthly compute charges = 1,100,000 * $0.00001667 = $18.34

Monthly request charges

The monthly request price is $0.20 per 1 million requests and the free tier provides 1M requests per month.

Total requests – Free tier requests = Monthly billable requests

3M requests – 1M free tier requests = 2M Monthly billable requests

Monthly request charges = 2M * $0.2/M = $0.40

Total monthly charges

Total charges = Compute charges + Request charges = $18.34 + $0.40 = $18.74 per month

Figure 6. Example pricing case.

3 Language Support

Programming languages are major tool to give instructions to a computer. There are as many programming languages as existing human-based languages which are used to communicate with computer. The part of the language a machine can comprehend is referred to as a "binary." These languages enable machines to process big and complicated sections of data rapidly and effectively. For instance, if an individual receives a list of randomized figures varying from one to ten thousand and is requested to position them in ascending order, it is likely to take a considerable quantity of moment and include some mistakes. These types of hectic problems can be solved by neatly writing code in programming language with different logic and patterns.

AWS supports several languages for writing Lambda functions. But the common pattern or logic of writing the function remains mostly identical. The core pattern of such logic is that the Lambda function code should be in stateless style and can't have affinity and dependence to underlying compute infrastructure. Such code must limit lifetime of its request while performing tasks like accessing local file system, child processes and other similar artifacts. Instead Amazon S3, Amazon DynamoDB, or another cloud storage service should be used as a method of storing persistent state of data. The advantage of keeping functions as stateless enables scaling the incoming rate of events and requests when AWS Lambda launches as many copies of a function as needed. The specific instance of computation is preserved and reused again in identical scenarios without duplicating the process itself. (docs.aws.amazon.com 2019: 31-32)

Choosing a language runtime performance is obviously based on the amount of convenience and abilities with each of the assisted runtime. However, if the performance of application is taken into consideration, the specific performance of each language running in lambda should be noted. For example, the compiled language such as Java and .NET have the highest initial setup cost for invocation of containers for first time but have better performance after initial invocations. In comparison to the compiled languages, interpreted languages (Node.js and Python) have very quick initial invocation times, but cannot achieve as high a performance as the compiled languages mentioned before. Therefore, while implementing AWS serverless Lambda, if the application is latency-sensitive and cost reduction is priority, the interpreted languages are best options. In cases

where these criteria are not important, the languages that the developer is comfortable with can be chosen. (d1.awsstatic.com 2017: 28-29)

Each language available for Lambda have their own specific pros and cons. Some perform better than other but they may lack some features that others have. In many instances, the logic implementation using the available language is more important than the language itself. The requirement of project and comfortability for the developers should also be taken into consideration while choosing the language for AWS Lambda. The feature currently supports six native languages and any other languages via runtime APIs. In this study, the focus is more into natively supported languages rather than runtime APIs because they are basically third-party tools.

3.1 Native Languages

The AWS Lambda supports some of the popular languages natively. The support for such languages has been priority of AWS Lambda since its development and the number is still growing. The most important technical reason is likely the cold start performance one language over another. The cold starts are mostly to relevant to user-facing which adds latency. The Lambda performance shouldn't be dependent in such cases and it is designed to minimize latencies. Apart from cold starts, other factors such as library support and company restrictions can also influence the language choice. The support for native languages adds comfortability and the developers tends to choose the languages they are most comfortable with.

All the apps can be split into one or several straightforward functional chunks and uploaded for execution to AWS Lambda. Lambda then provides the resources required to operate the task together with other management operations, such as self-scaling, accessibility of features, etc. A developer then has to perform tasks like writing the code, packaging it for deployment, and finally monitoring its execution and fine-tuning. Using native languages makes this job easy and straightforward. Each native language has their own programming model or a programming pattern. Currently, AWS officially supports Node.js, Java, Python, and C# as the programming languages for writing Lambda functions, with each language following a generic programming pattern. (Wadia & Gupta 2017: 34-35)

Choosing natively supported languages for serverless functions reduces cold starts execution period which is very important having a snappy user experience. When the function is run for the first time, the request asks for the code to be executed. This particular request is memory heavy task and after the initial run the other function execution is short and simple. Since the server time is main basis for the cost and performance, choosing native language is ideal compared to runtime APIs as they increase the execution period. in some set period where a request asks for your code to be executed. This is the main advantages of using the native languages where the AWS handles the abstraction of managing servers, containers & scaling. The introduction to each natively supported language and their most prominent features with supported runtime version is discussed below.

3.1.1 Java

Java is still important language as it was decades earlier, one of the finest programming dialects ever to be used on the open source environment. Java's position as one of the most common programming languages has been maintained since its inception in the mid-90s, for excellent purposes. The JVM paradigm used in Java ensures nearly any application on every system runs published with this language. Other top languages of programming often don’t contain Java's capacity to scale even the extensive applications. (Hasan 2019)

With one of the biggest developer communities, Java will certainly continue to be one of the leading forms of programming in the coming years. The developers with Java skills are high demand on industries as this language is mostly used in developing a revolutionary open source application. These are main reasons behind popularity of Java in tech industry and community. (Hasan 2019)

Name	Identifier	JDK	Operating System
Java 8	`java8`	java-1.8.0-openjdk	Amazon Linux

Figure 7. Java runtimes.

The above figure is the current runtime specifications for Java language in AWS Lambda. As Java is a compiled language, Lambda provides the Amazon Linux build of openjdk 1.8 by default. The Java programmer can also use standard tools like Maven or Gradle to compile the Lambda function. The build process should mimic the same build process that the developer would use to compile any Java code that depends on the AWS SDK. For the Lambda function, the Java compiler tool execute on the source files and include the AWS SDK 1.9 or later with transitive dependencies on the classpath.

3.1.2 Python

Python is the most used open sourced programming language currently. It has large base of community-based contributors and is on continuous development process. One of the most used programming languages of our time, Python continues to hold its ground among open source contributors and isn't going away soon. Python can be used as a full language in many distinct kinds of applications. This includes APIs, crawlers, scrapers, backend schemes, etc. The developers can even develop complex desktop applications using this popular dialect of computer languages.

Due to a large number of top-class plugins and third-party libraries interaction, Python is extensively implemented in machine learning and data analysis. Python has several popular libraries like SciPy and Panda. This access to useful third-party libraries makes Python one of the most common languages of today's programming. While not dimmed appropriate for managing apps requiring low-level system manipulation, this language can be used for nearly every form of open source project. (Hasan 2019)

Name	Identifier	AWS SDK for Python	Operating System
Python 3.6	`python3.6`	boto3-1.7.74 botocore-1.10.74	Amazon Linux
Python 3.7	`python3.7`	boto3-1.9.42 botocore-1.12.42	Amazon Linux
Python 2.7	`python2.7`	N/A	Amazon Linux

Figure 8. Python runtimes.

The above figure is the current runtime specifications for Python language in AWS Lambda. The AWS provides SDKs for different version of Python used. If the provided SDK is not used, the user has to create the deployment packages manually. The additional libraries, dependencies and tools needs to be uploaded to Lambda alongside function codes.

3.1.3 Node.js

Node.js (Node) is a platform for open source implementation to execute server-side JavaScript applications. Node is mostly used for real-time web services such as chat, news feeds and web push notifications because of its persistence while connecting server and browser. It is intended to run on a dedicated HTTP server and to use only a single thread for processing. Node.js applications are event-based and run asynchronously. Node-built applications does not follow the traditional model of receiving, processing, sending, waiting and receiving again cycle. Instead, Node processes incoming requests in a continuous stack of events and receives relatively small queries without waiting for responses. (Rouse 2017)

Node.js is an extremely customizable server engine that became famous as a means of creating real-time web APLs that can operate across platforms for JavaScript applications. Due to its continuous integration into the latest open source projects, JavaScript has been experiencing a massive growth and usages.

This is a change away from mainstream models running bigger, more complicated processes and running multiple threads simultaneously, with each thread waiting before moving on for its suitable response. According to its creator Ryan Dahl, the major advantages of Node is that it does not block input/output (I/O). Some developers are extremely critical of Node.js and point out that the implementation will block if a single thread needs a substantial amount of CPU cycles and the blocking can crash the application. But it is also claimed that the CPU processing time is less of a concern as of the Node can be based on high amount of small processes. (Rouse 2017)

Name	Identifier	Node.js Version	AWS SDK for JavaScript	Operating System
Node.js 10	`nodejs10.x`	10.15	2.437.0	Amazon Linux 2
Node.js 8.10	`nodejs8.10`	8.10	2.290.0	Amazon Linux

Figure 9. Node.js runtimes.

The above figure is the current runtime specifications for Node.js in AWS Lambda. Most of the deployment commands are handled through lambda console and codes can be uploaded directly or from the storage buckets.

3.1.4 Ruby

Ruby is a flexible language developed in the mid-90s but only became popular decades later. It is like Python in many aspects as it is also an interpreted, dynamic, and object-oriented language. Most of the webservices were powered by Ruby after the development of powerful web frameworks such as Ruby on Rails and Sinatra. But development of JavaScript frameworks has been replacing the Ruby usages greatly in recent times. (Hasan 2019)

Many well-known current web applications like GitHub, Airbnb, ASKfm, Goodreads, and Fiverr use ruby in one way or another. This language is also employed by many popular open source projects like Homebrew, Discourse, Metasploit Framework, etc. It is mostly preferred by developers working in open-sourced project due to its dynamic and flexible programming. (Hasan 2019)

Name	Identifier	AWS SDK for Ruby	Operating System
Ruby 2.5	`ruby2.5`	3.0.1	Amazon Linux

Figure 10. Ruby runtimes.

The above figure is the current runtime specifications for Node.js in AWS Lambda. Layering of dependencies is commonly used while implementing Lambda functions in Ruby

language. The plugin excludes all files by default, so it is needed to whitelist (add to the package/includes section) any files or directories. This package then accomplishes the part where it copies all the gem files into the vendor directory in the working directory. The plugin then analyzes the Gem file to figure out which gems should be included or excluded if they are in the development/test groups.

3.1.5 Go

Go is the recently developed programming language which is rapidly growing its market share. It is promising language used to tackle some of the hardest computational problems with a relatively subtle approach. It was developed by Robert Griesemer, Rob Pike, and Ken Thompson and currently maintained by Google. Go incorporates all of C's advantages such as being a static typed, compiled language and adding advanced characteristics such as garbage collection, structural typing, and concurrent CSP-style. (Hasan 2019)

While still being highly powerful language, it is straightforward to grasp and there is no complex programming. Kubernetes, Docker, Hugo, and Ethereum are some of the latest open source project which has embraced Go for its convenience it offers without lacking any fundamental features. From its current performance it is anticipated that Go will power most future systems. (Hasan 2019)

Name	Identifier	Operating System
Go 1.x	`go1.x`	Amazon Linux

Figure 11. Go runtimes.

The above figure is the current runtime specifications for Go language in AWS Lambda. Additionally, AWS lambda provides few implementations that is useful while using this language. Such additional features can be accessed from the repository of AWS.

3.1.6 C#

C# is the programming language developed by Microsoft. It was developed in such a way that it can be used for many different projects. It was intended to be used only from Microsoft's .NET framework. Later it was popular and widely used because it could also solve other demanding computational. The language is uncomplicated and modern, with the ability to let developers take an object-oriented approach which is not available in C. C# codes compile like C++ codes and have syntax like Java, so it can be call the blend between C++ and Java. (Hasan 2019)

The ability of this language to craft any application as wanted is the main reason for its popularity. It can be used to build robust application based on any requirements possible in tech industry. The uses cases of this language ranges from tarting from complex web APIs to full-fledged desktop applications. It is also a powerful and compiled language which is in high demand in market. (Hasan 2019)

Name	Identifier	Languages	Operating System
.NET Core 2.1	`dotnetcore2.1`	C# PowerShell Core 6.0	Amazon Linux
.NET Core 1.0	`dotnetcore1.0`	C#	Amazon Linux

Figure 12. Go runtimes.

The above figure is the current runtime specifications for C# language in AWS Lambda. C# Lambda function can be created using the Visual Studio IDE by selecting "Publish to AWS Lambda" in the Solution Explorer. Alternatively, the programmer can directly run the "dotnet lambda publish" command from the dotnet CLI which has the [# Lambda CLI tools patch] installed. The C# source code zip file is created with all NuGet dependencies and DLL assemblies, and automatically uploaded to AWS Lambda using the runtime parameter “dotnetcore1.0”.

4 Project Work Report: Comparison among Native Languages

In the field of information technology, the discussion about comparison of programming languages is a common topic among developers, software engineers and other stakeholders. To match the pace of technological development and evolution, numerous programming languages are designed, specified, and implemented every year. The ability of certain programming language to fulfill the needs of changing technology will determine its popularity and usage. For the case of this thesis, the project work is related to comparison of native language supported by AWS Lambda.

4.1 Introduction

This project report is the part of current thesis topic. As it has been explained in previous parts about the theoretical background about AWS Lambda and serverless framework, the functionality is implemented in real practice and the findings of research is presented in following parts. The project has been chosen to reflect the core theme of thesis as much as possible. For this purpose, a study into performance of each natively supported languages running the similar task is measured using the AWS Lambda service. The key results with conclusive analysis are presented below. The main aim of this project was to get familiarized with AWS Lambda implementations and overview to its performance.

The project work also elaborates the perks of AWS Lambda which is the dynamically scaled and billed-per-execution compute service. In AWS lambda, the instances of Lambdas are added and removed dynamically for the perfect infrastructure operational service. While initiating this service, the first-time running of application service functions is very critical aspect of observation as this task consumes the most resources and time. This first-time run is also called ‘cold start’. While on cold start phase, the new instance handles its first request provoking the response time increment.

The first cold start happens when the very first request comes in after deployment to AWS Lambda console. After that request is processed, the instance stays alive for the time being to be reused for subsequent requests. The length of time that the request is alive is dependent on various factors. There is no predefined threshold after the instance gets recycled, the empiric data show some variance of the idle period.

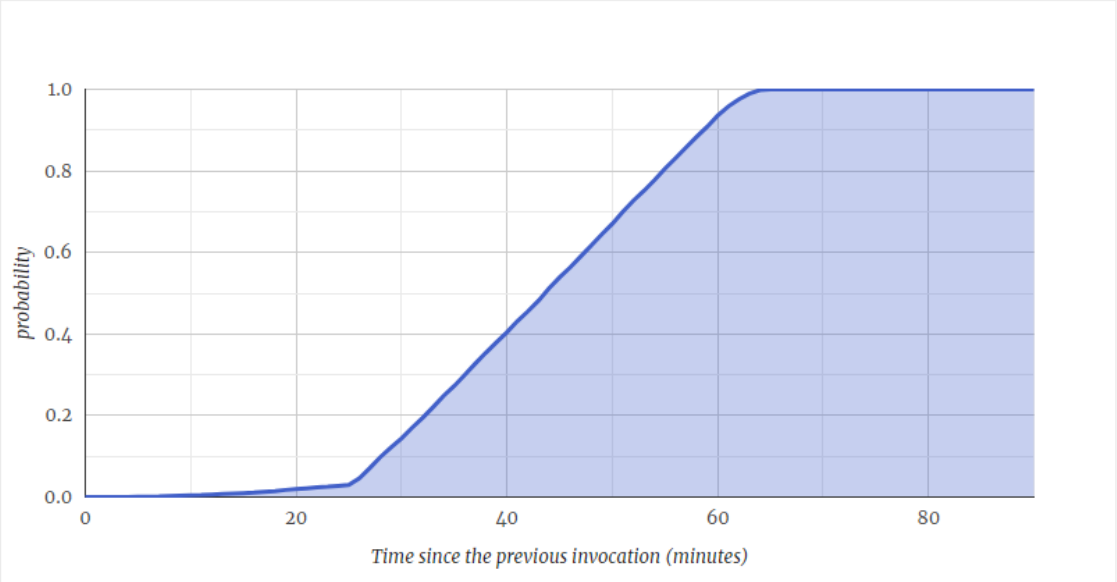

Figure 13. Probability of a cold start happening before minute X

The above-mentioned figure estimates the probability of cold by the interval between two subsequent requests. The lifetime of an instance doesn't seem deterministic, but it can be estimated to be between 25 and 65 minutes. An idle instance almost always stays alive for at least 25 minutes. Then, the probability of it being disposed slowly starts to grow and reaches 100% somewhere after 1 hour since the last request. The probability data in this figure can't be assumed to be precise as the processing of request by lambda can be variable but the overall trend should be representative.

4.2 Objectives

The objective of this project report is to perform practical analysis of performance results of different available native languages provided by AWS Lambda. In previous sections of this thesis, it is mentioned about present context of serverless framework architectures and its benefits. Languages are the important tools for execution and construction of service infrastructure. This report in general is trying to reflect those advantages and making the comparative study of available tools.

The project was conducted to get the grasp of knowledge about implementation of AWS Lambda with serverless architecture. For this project a simple test case is used, which is not as complex as the normal tasks and functions that is used generally in tech industry. Although the real-life problems are more complicated compared to the case that is being tested, it is enough to give overall representation of the service itself. The performance result may vary rapidly from the result of this test case, but overall trend remains the same.

The goal of this project work and report is to present the insights of performance variations and statistical deviation among the native languages of AWS lambda. The following part of this report explains the test case used for this instance and its implementation methods. After that, there are presentation of statistical data analytics for each of the languages where the performance difference can be seen. With this data, the observations are made from which overall conclusions are drawn.

4.3 Implementation Methods and Test Case

For this project, a simple test case was implemented on AWS lambda service. Since this is a comparative test among various native languages available in AWS Lambda, a mathematical problem was chosen and written in all available language with their respective syntax and pattern. It was coded in such a way that the result of the function is same from all the language implementation. These codes were uploaded to Lambda service of AWS and the time duration of each execution were measured recursively multiple times. As the Lambda performance is varying continuously, a large number of data outcomes were collected and observed using the theory of empirical probability.

The empirical probability of an event is the proportion between number of outcomes in which a specific event occurs in total number of trials. It is mostly used for estimation purposes and the result represents the trend rather than actual regression. It is also known generally as cumulative probability. The general empirical probability measure for measurable subsets is given by following mathematical proof,

Let $X_1, X_2, \ldots$ is a sequence of identically distributed variables which are also random and independent in a space S with probability of distribution P, then the empirical measure is given by

$$P_n(A) = \frac{1}{n}\sum_{i=1}^{n} I_A(X_i) = \frac{1}{n}\sum_{i=1}^{n} \delta_{X_i}(A)$$

Where, I_A is the indicator function and δ_X is the Dirac measure.

For writing the function codes in different language for same results, the task of printing first 30 Fibonacci series is chosen. The Fibonacci numbers is a unique sequence of natural numbers where each number is sum of two preceding numbers. This interesting ratio was first introduced by Italian Mathematician Leonardo Fibonacci at around 200 BC. This numbers are also strongly related to golden ratio. They are used widely in field of science and mathematics. Applications of Fibonacci numbers include computer algorithms such as the Fibonacci search technique and the Fibonacci heap data structure, and graphs called Fibonacci cubes used for interconnecting parallel and distributed systems.

The general formula for Fibonacci series is given by following equations,

If $F_0 = 0$ and $F_1 = 1$, then,

$$F_n = F_{n-1} + F_{n-2},$$

For n > 1, where n is a natural number.

For the test case of our project the above state algorithm is used. Codes in every native language supported by AWS lambda is written to give the list of first 30 Fibonacci numbers. The data is collected after these codes were uploaded and executed multiple times. The actual codes in each specific language is presented at Appendix 1 at the end section of this thesis.

4.4 Comparative Analytics

The performance statistics is largely dependent on the choice of programming language used for creation of AWS Lambda function. The details of each language have been discussed on previous sections of this thesis. JavaScript, Python, Go, Java, and Ruby are all comparable: most of the time they complete within 500 milliseconds and almost always within 800 milliseconds. C# is a distinct underdog with cold starts spanning between 0.8 and 5 seconds. The following chart shows the typical range of cold starts in AWS Lambda, broken down per language. The darker ranges are the most common 67% of durations, and lighter ranges include 95%.

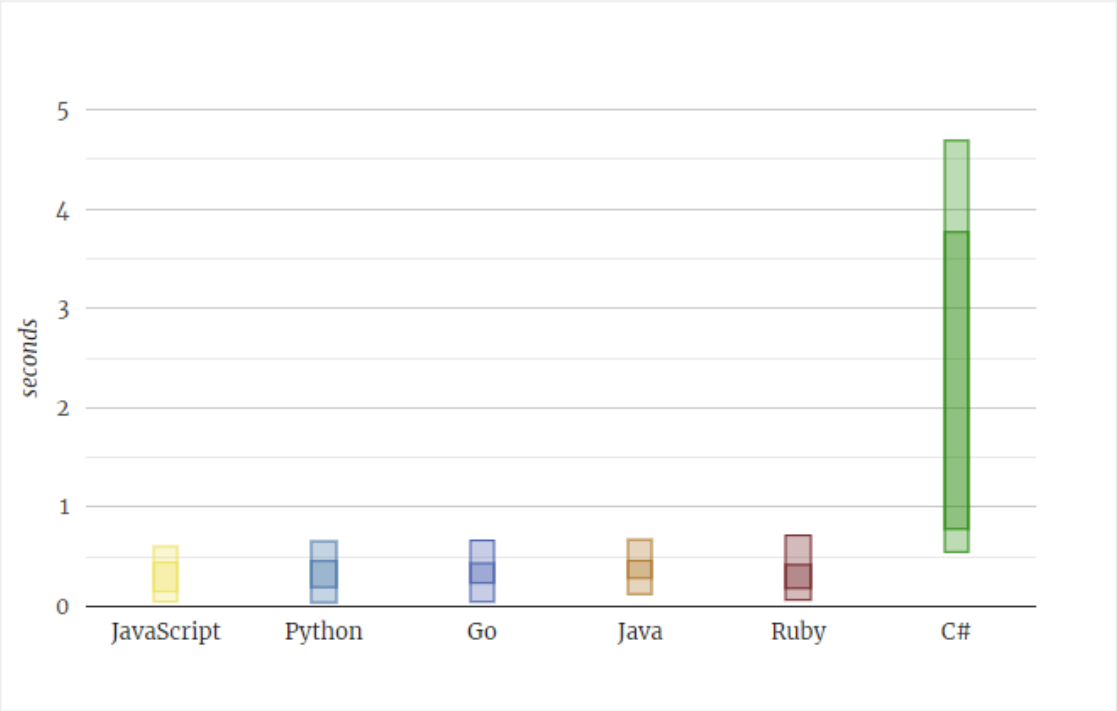

Figure 14. Typical cold start durations per language

This figure represents the cumulative representation data for different languages under same test cases. It can be observed that duration for execution of AWS serverless Lambda are similar except for the case of C# which takes more time for resolving the requests than other native languages. In the following parts, language specific performance analytics is presented.

Apart from the language choice, the package size of each function chunks with dependencies also play important role on execution time of requests in Aws Lambda. Adding dependencies and thus increasing the deployed package size will further increase the cold start durations. In most of the cases, the functions with many dependencies which in turn makes the size of zip file more are slower to have cold starts. The relation between deployment size and time duration for execution are somewhat in logarithmic ratio.

Moreover, AWS Lambda also provides the options for choosing the instance size. The instance size can be defined as the memory size that gets allocated to single instance to function. The cost of service is directly proportional to chosen instance size. Besides, the CPU resources are allocated proportionally to the memory. So, in theory, larger instances could start faster. However, there seems to be no significant speed-up of the cold start as the instance size grows.

The charts below give the distribution of cold start durations per supported programming language. All charts except the last one has the same horizontal scale (0.0-1.0 sec) to make them easily comparable.

Java:

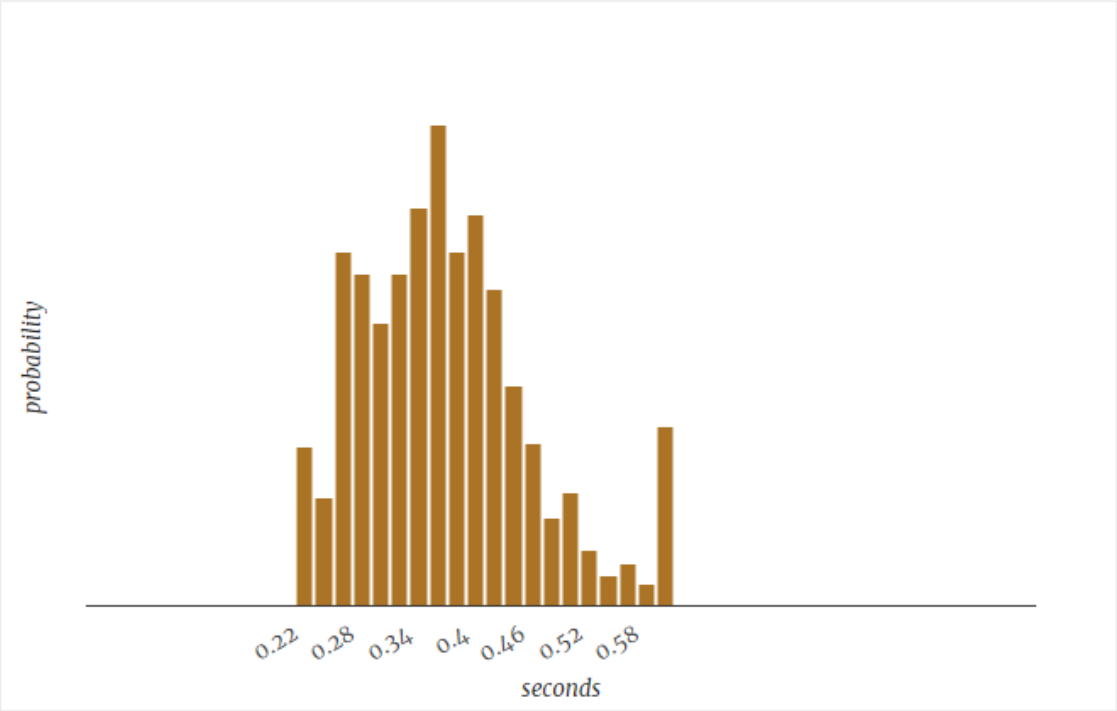

Figure 15. Cold start durations of AWS Lambda in Java

The above figure is a distributive graph for duration required for execution of test case multiple times for Java. It can be observed that in most of the cases the time of response lies between 0.28 to 0.46 seconds.

Python:

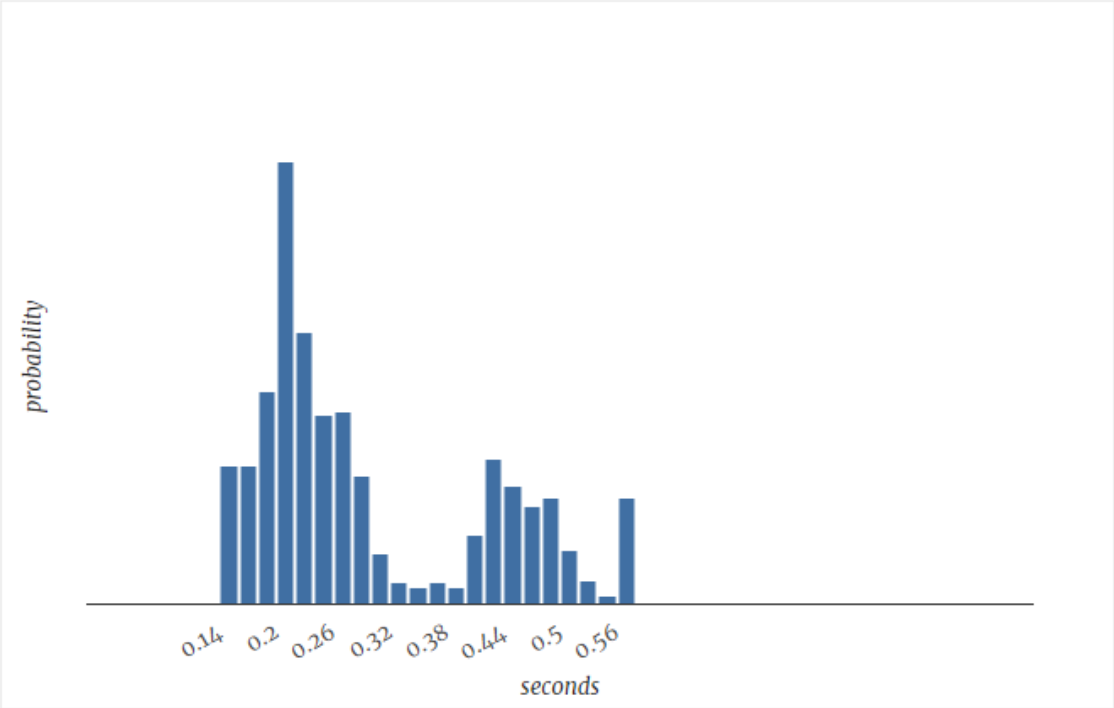

Figure 16. Cold start durations of AWS Lambda in Python

The above figure is a distributive graph for duration required for execution of test case multiple times for Python. It can be observed that in most of the cases the time of response lies between 0.20 to 0.32 seconds and some around 0.5 seconds.

Node.js/JavaScript:

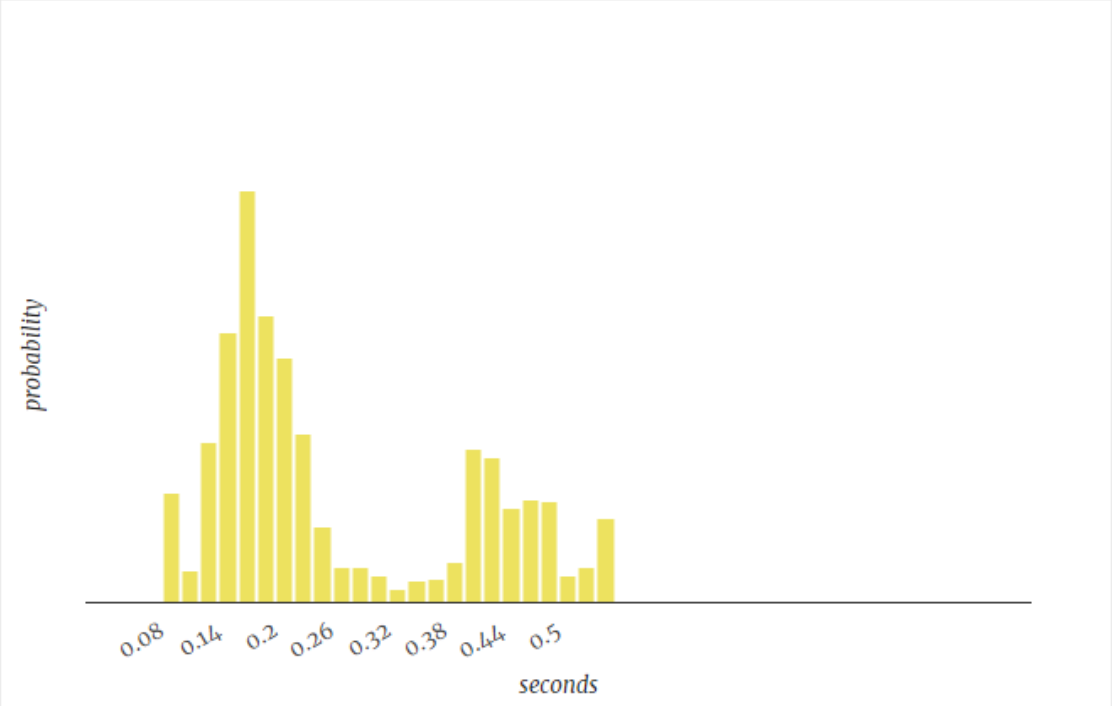

Figure 17. Cold start durations of AWS Lambda in Node.js

The above figure is a distributive graph for duration required for execution of test case multiple times for Node.js. It can be observed that in most of the cases the time of response lies between 0.14 to 0.26 seconds.

Go:

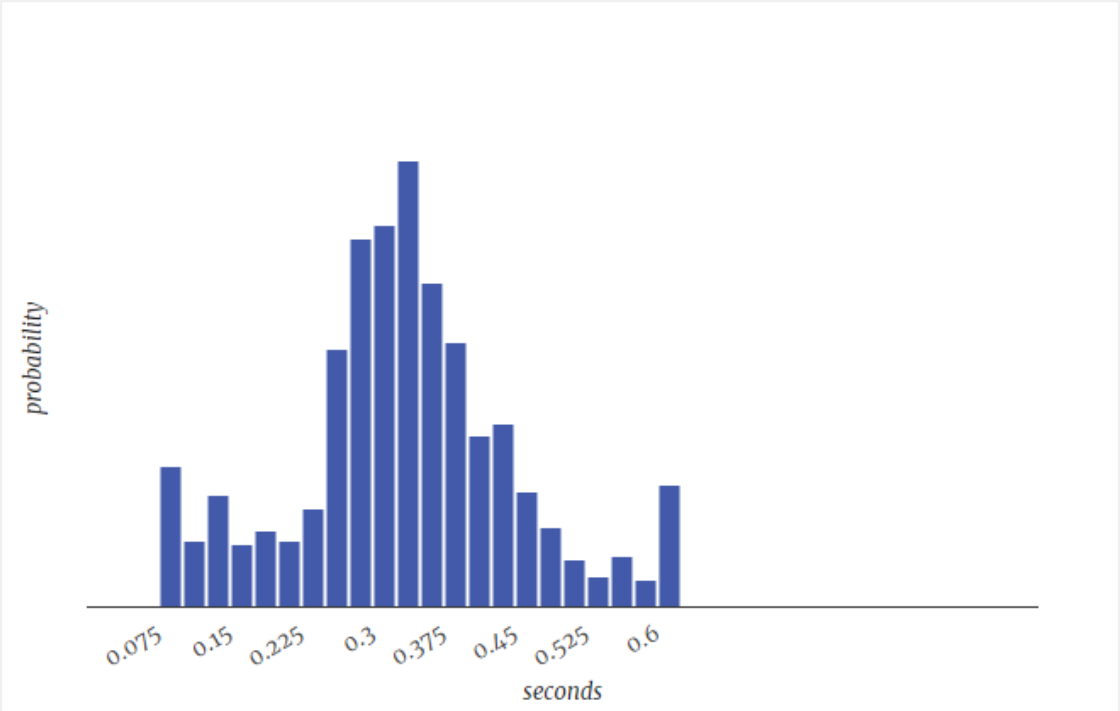

Figure 18. Cold start durations of AWS Lambda in Go

The above figure is a distributive graph for duration required for execution of test case multiple times for Go language. It can be observed that in most of the cases the time of response lies between 0.23 to 0.45 seconds.

Ruby:

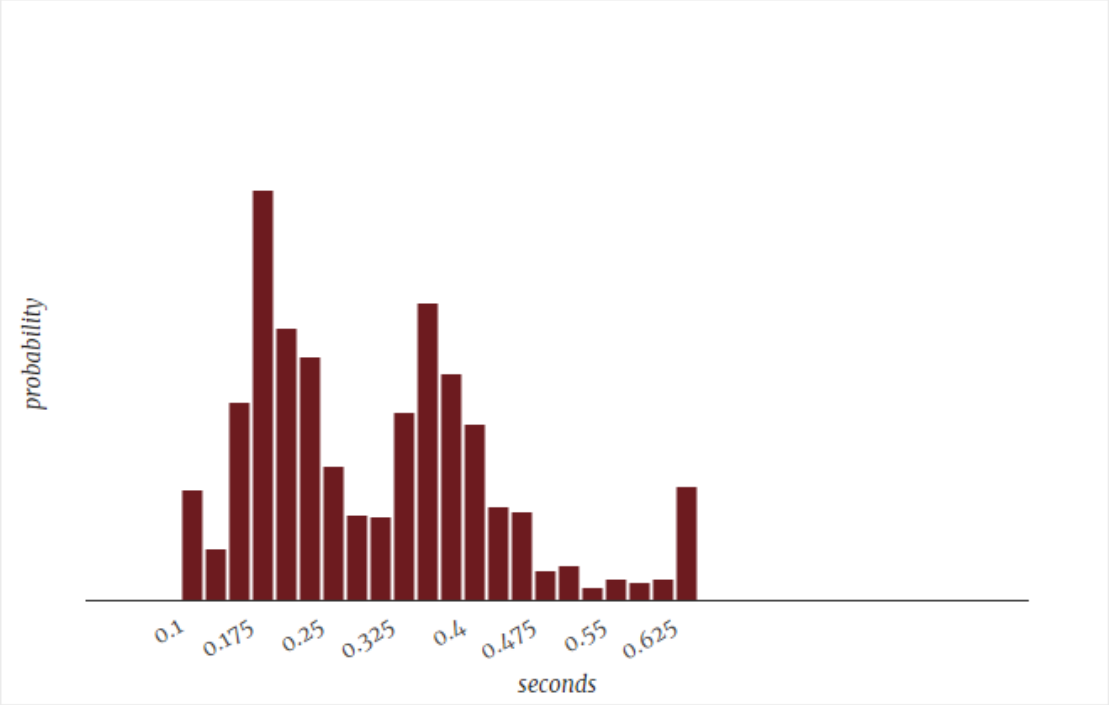

Figure 19. Cold start durations of AWS Lambda in Ruby

The above figure is a distributive graph for duration required for execution of test case multiple times for Ruby. It can be observed that in most of the cases the time of response lies between 0.175 to 0.45 seconds.

C#:

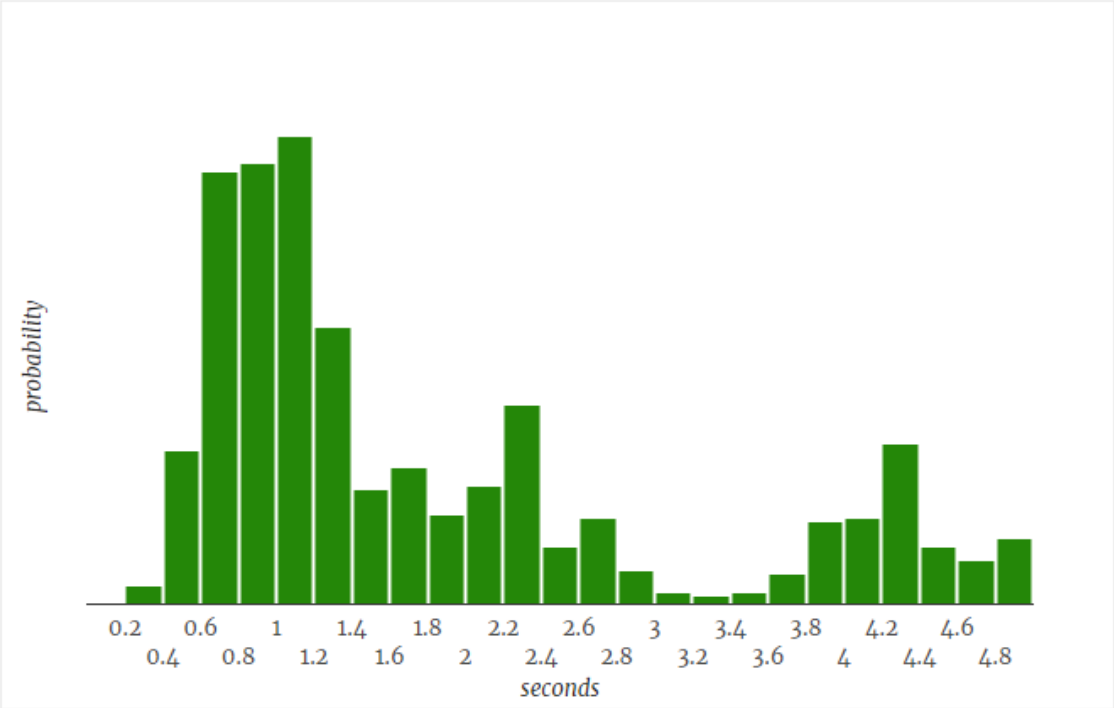

Figure 20. Cold start durations of AWS Lambda in C#

The above figure is a distributive graph for duration required for execution of test case multiple times for C#. It can be observed that in most of the cases the time of response lies between 0.60 to 2.22 seconds. Comparatively, C# has the most latency in cold starting than other languages.

4.5 Observation

Based on comparative analysis in previous section, the following observations were made.

Static languages have more consistent performance?

AWS Lambda provides both static and compiled languages to choose from as language options. C# and Java are compiled languages and others are static languages. While testing the performance and max duration, the compiled languages have similar outcomes. They tend to have longer cold starts but performance wise they are faster. The reason behind slower cold start is due to large number of dependencies that compiled language use as compared to static languages.

Java packages are huge but has very consistent performance?

Java was one of the first native language to be used in AWS Lambda. Java is widely used for its performance and consistent results. But to make that possible, Java utilize a lot of dependent components which in turn increases the overall package size of Lambda function. The package size of Java can be enormous compared to other available native languages but that will affect the initial cold start only. After initial setup, the performance of java is excellent.

C# is slower?

C# is the only odd language in terms of invocation duration. This will massively affect the time duration required for initial execution. Although, the consecutive processes except the cold start are like other languages, the time taken for initial cold start drops down its cumulative performance results in AWS Lambda.

5 Conclusions

The major goal of this thesis was to get the details on the working method of AWS Lambda and get subtle comparisons among the implementations of this feature. Looking at the bigger context, the development of technology is accelerating than ever and things are evolving at an incredible pace. Whatever performance discrepancies are present today can change quickly as AWS improves all the platforms behind the scenes. In present context, the differences on implementation methods of this service is subtle and almost similar. The test provides some insights, but it's hardly representative of a real-world application. AWS Lambda itself is the best example of serverless architecture and is trying continuously to adapt to changing market needs.

The main intention of AWS to provide options for various languages is mostly to let developers to write Lambda functions in which they are comfortable with. Therefore, AWS is continuously adding new supported native language since the time of its introduction. AWS has also planned to make Lambda feature to incorporate more languages in coming future. This will help AWS to attract more users to implement AWS Lambda services and increase their business.

This thesis also has some research and findings about the factors that affects the performance of serverless architecture other than choice of language itself. The AWS Lambda is a tool that can be modeled to fit the specific nature of service applications. The pricing model study provides the insights upon cost of production while maintaining the best performance and fulfilling the needs of service. They should also be able to catch the trend of technical development and reusability. These points should be noted and tested carefully while designing the serverless architecture.

From the analytics and observation presented in above sections, it can be learned that native language options available at implementing AWS Lambda doesn't have significant correlation with performance. There are many other variables that affect the performance of the Lambda functions directly. The best advantage of choosing AWS Lambda is the realization of serverless architecture which is replacing the traditional model of IaaS. AWS Lambda is a revolutionary computing feature which is expected to play huge role in future technological systems and service.

References

Wittig, A. & M. (2015). *Amazon Web Services in Action.* [online] Available at: https://www.pdfdrive.com/amazon-web-services-in-action-d34795205.html [Accessed 14 May 2019].

Poccia, D. (2016). *AWS Lambda in Action.* [online] Available at: https://livebook.manning.com/#!/book/aws-lambda-in-action/chapter-1/20 [Accessed 14 May 2019].

Sbarski, P. (2017). *Serverless Architectures on AWS.* [online] Available at: https://openlibrary.org/books/OL26836345M/Serverless_Architectures_on_AWS_With_examples_using_AWS_Lambda [Accessed 19 May 2019].

Wadia Y. & Gupta M. (2017). *Mastering AWS Lambda.* [online] Available at: https://www.pdfdrive.com/mastering-aws-lambda-d158232201.html [Accessed 21 May 2019].

Boyd M., Gain, B. & Gienow M. (2018). *Guide to Serverless Technologies.* [online] Available at: https://thenewstack.io/guide-to-serverless-technologies-free-ebook-on-the-new-stack/ [Accessed 28 May 2019].

docs.aws.amazon.com, Amazon Web Services, Inc. (2019) *AWS Lambda - Developer Guide.* [online] Available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf [Accessed 4 June 2019].

d1.awsstatic.com, Amazon Web Services, Inc. (2017) *AWS Serverless Architectures with AWS Lambda. AWS Lambda - Developer Guide* [online] Available at: https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf [Accessed 19 May 2019].

Hasan, M. (2019). *Top 20 Most Popular Programming Languages to Learn for Your Open-source Project.* [online] Available at: https://www.ubuntupit.com/top-20-most-popular-programming-languages-to-learn-for-your-open-source-project/ [Accessed 24 May 2019].

Rouse, M. (2017). *DEFINITION Node.js* [online] Available at: https://whatis.techtarget.com/definition/Nodejs [Accessed 24 May 2019].

Sample Codes for Project Test Case

Snippet of code for generating Fibonacci series in Java

```
import java.util.List;
import java.util.stream.Stream;
import static java.util.stream.Collectors.toList;

public class Fibonacci {

    /**
     * Java 8 / Lambda approach to generate fibonacci series.
     * Fibonacci always start as classic (e.g. 0, 1, 1, 2, 3, 5)
     * @param series Number of how many fibonacci number should be generated
     * @return List holding resulting fibonacci number.
     */
    public static List<Integer> generate(int series) {
        return Stream.iterate(new int[]{0, 1}, s -> new int[]{s[1], s[0] + s[1]})
                .limit(series)
                .map(n -> n[0])
                .collect(toList());
    }

    public static void main(String[] args) {
        System.out.println(Fibonacci.generate(30));
    }
}
```

Snippet of code for generating Fibonacci series in Node.js

```
exports.handler = (event, context, callback) => {
  const fibo = fib()
  for (let i=0; i<30; i++){
    console.log(fibo())
  }
  const result = {
    "isBase64Encoded": false,
    "statusCode": 200,
    "headers": {},
    "body": "done"
  }

  callback(null, result);
};

function fib() {
  let x = 0
  let y = 1
  return function () {
    const temp = x;
    x = x + y;
    y = temp;
    return y
  }
}
```

Snippet of code for generating Fibonacci series in Go

```
package main

import (
        "fmt"
        "github.com/aws/aws-lambda-go/events"
        "github.com/aws/aws-lambda-go/lambda"
)

func Handler() (events.APIGatewayProxyResponse, error) {
        f := fibonacci()
        for i := 0; i < 30; i++ {
                fmt.Println(f())
        }

        return events.APIGatewayProxyResponse{
                Body:       "done",
                StatusCode: 200,
        }, nil
}

func main() {
        lambda.Start(Handler)
}

func fibonacci() func() int {
        x, y := 0, 1
        return func() int {
                x, y = y, x+y
                return x
        }
}
```

Snippet of code for generating Fibonacci series in Python

```
def handler_name(event, context):

def fibonacci(count):
    fib_list = [0, 1]

    any(map(lambda _: fib_list.append(sum(fib_list[-2:])),
                                          range(2, count)))

    return fib_list[:count]

return print(fibonacci(30))
```

Snippet of code for generating Fibonacci series in Ruby

```
module Fibonacci
  class Operation
    def self.process(event:,context:)
    def fibonacci(n)
      num1 = 0
      num2 = 1
      while num2 < n
        puts num2
        num1, num2 = num2, num1 + num2
      end
    end
end

  res = Operation.new
  res.fibonacci(30)
end
```

Snippet of code for generating Fibonacci series in C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be
//converted into a .NET class.
[assembly: LambdaSerializer(typeof(Amazon.Lambda.Serialization.Json.JsonSerializer))]

namespace AWSLambda3 {
    class Program
    {
        static int FibonacciSeries(int n)
        {
            int firstnumber = 0, secondnumber = 1, result = 0;

            if (n == 0) return 0; //To return the first Fibonacci number
            if (n == 1) return 1; //To return the second Fibonacci number

            for (int i = 2; i <= n; i++)
            {
                result = firstnumber + secondnumber;
                firstnumber = secondnumber;
                secondnumber = result;
            }
            return result;
        }
        static void Main(string[] args)
        {

            for (int i = 0; i < 30; i++)
            {
                Console.Write("{0} ", FibonacciSeries(i));
            }
            Console.ReadKey();
        }
    }
}
```