Louisiana State University

LSU Scholarly Repository

8-8-2022

Evaluating Serverless Computing

Charitra Maharjan

Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://repository.lsu.edu/gradschool_theses

Part of the Computer and Systems Architecture Commons, and the Other Electrical and Computer Engineering Commons

Recommended Citation


This Thesis is brought to you for free and open access by the Graduate School at LSU Scholarly Repository. It has been accepted for inclusion in LSU Master's Theses by an authorized graduate school editor of LSU Scholarly Repository. For more information, please contact gradetd@lsu.edu.
EVALUATING SERVERLESS COMPUTING

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Division of Computer Science and Engineering

by
Charitra Maharjan
B.E., Advanced College of Engineering and Management, 2013
M.E., Louisiana State University, 2017
December 2022
Acknowledgments

Dr. Hao Wang, Asst. Prof. of the Division of Computer Science and Engineering has been an inspirational and wonderful person who introduced me to the world of Serverless platforms. I sincerely thank him for the opportunity to work with him. I would also like to appreciate his moral support and immense tutelage towards completing my master’s program.

My sincere thanks go to Dr. Juana Moreno, a Patricia Hewlett Bodin Distinguish Professor, for her consent to be the committee member. In addition, I feel to acknowledge my indebtedness and a deep sense of gratitude for her opportunity to work as a Graduate Assistant (GA) on the BRBytes project, a research practitioner partnership between East Baton Parish School System (EBRPSS) and Louisiana State University (LSU), under Fernando Alegre.

I cannot express enough thanks to Dr. Bijay B. Karki, Prof. and Chair of Computer Science, for his consent to be a committee member and his valuable suggestions to improve this document.

Finally, I am grateful to my parents, Mr. Gyan Bahadur Maharjan and Mrs. Ratna Maya Maharjan, and my family members for their love and blessing. I sincerely thank my dear wife, Mrs. Anjana Maharjan, for all her moral support and inspiration in this journey.
# Table of Contents

Acknowledgments .......................... iii

List of Figures ........................... vi

Nomenclature, Symbols, Acronyms ........ x

Abstract .................................. xi

Chapter 1. Introduction ..................... 1
  1.1. Types of Cloud Computing services ..... 1
  1.2. Serverless ................................ 3
  1.3. Motivation and Thesis Goal .......... 7
  1.4. Thesis Layout .......................... 8

Chapter 2. Background ..................... 9
  2.1. Related Work .......................... 10
  2.2. Data Collection ....................... 12

Chapter 3. Amazon Web Services .......... 14
  3.1. Advantages of AWS ................... 14
  3.2. AWS Services .......................... 15
  3.3. Cold and warm Start Call of Lambda Functions . . 20
  3.4. AWS Lambda Invocation ............... 22
  3.5. Conclusion ............................ 24

Chapter 4. Methodology .................... 26
  4.1. Creating an Environment ............ 26
  4.2. Regression Testing ................... 27
  4.3. Benchmark Invocation ............... 28
  4.4. Running Experiments ................. 30
  4.5. Conclusion ............................ 31

Chapter 5. Result and Analysis ........... 32
  5.1. Perf-Cost .............................. 32
  5.2. Latency ................................ 47
  5.3. Overhead .............................. 49
  5.4. Conclusion ............................ 52

Chapter 6. Discussion and Future Work ... 53

Appendix A. Getting access and secret keys from AWS ... 57
List of Figures

1.1. Types of Serverless Cloud Computing Services ................................. 2
1.2. Cloud Computing Structure .............................................................. 3
1.3. A Simple Serverless Architecture ...................................................... 5
3.1. AWS Lambda Infrastructure ............................................................... 17
3.2. VM and function instance organization in AWS Lambda. A rectangle represent a function instance, A or B indicates different tenants ................................. 17
3.3. Cold and warm Start Call ................................................................. 21
3.4. Synchronous Invocation .................................................................... 22
3.5. Asynchronous Invocation .................................................................. 23
3.6. Event Source Mapping with Polling Invocation ................................. 24
4.1. A simple Experimental Topology ......................................................... 26
4.2. Regression test for a single benchmark ............................................. 27
4.3. A Snapshot of a config/example.json File .......................................... 28
4.4. Changing Architecture to ARM64 .................................................... 28
4.5. Available Options for Benchmark Command ................................. 29
4.6. Options Available for an experiment ............................................... 30
4.7. Parameters of Per-Cost experiment ................................................. 31
5.1. Virtual Environment ......................................................................... 32
5.2. Configuration File example.json ....................................................... 33
5.3. Effect on Execution Time at various memory sizes for x86 and ARM64 for Dynamic-HTML
   (a) x86-vs-ARM64 at cold start for Dynamic HTML .............................. 34
   (b) x86-vs-ARM64 at warm start for Dynamic-HTML ............................ 34
5.4. Effect on Client Time at various memory sizes for x86 and ARM64 for Dynamic-HTML ................................................. 35
   (a) x86-vs-ARM64 at cold start for Dynamic HTML ............................ 35
   (b) x86-vs-ARM64 at warm start for Dynamic-HTML ............................ 35

5.5 Effect on Provider Time at various memory sizes for x86 and ARM64 for Dynamic-HTML ................................................. 36
   (a) x86-vs-ARM64 at cold start for Dynamic HTML ............................ 36
   (b) x86-vs-ARM64 at warm start for Dynamic-HTML ............................ 36

5.6. Effect on Execution Time at 128 MB for various multiple workloads ........... 38
   (a) x86-vs-ARM64 at cold start ..................................................... 38
   (b) x86-vs-ARM64 at warm start ................................................... 38

5.7. Effect on Client Time at 128 MB for various multiple workloads ................ 38
   (a) x86-vs-ARM64 at cold start ..................................................... 38
   (b) x86-vs-ARM64 at warm start ................................................... 38

5.8. Effect on Provider Time at 128 MB for various multiple workloads ............. 39
   (a) x86-vs-ARM64 at cold start ..................................................... 39
   (b) x86-vs-ARM64 at warm start ................................................... 39

5.9. Effect on Execution Time at cold and warm start call for various multiple workloads 40
   (a) cold-vs-warm at x86 for multiple workloads ................................. 40
   (b) cold-vs-warm at ARM64 for multiple workloads ............................. 40

5.10. Effect on Client Time at cold and warm start call for various multiple workloads 41
      (a) cold-vs-warm at x86 for multiple workloads ................................. 41
      (b) cold-vs-warm at ARM64 for multiple workloads ............................. 41

5.11. Effect on Provider Time at cold and warm start call for various multiple workloads 42
       (a) cold-vs-warm at x86 for multiple workloads ................................. 42
       (b) cold-vs-warm at ARM64 for multiple workloads ............................. 42

5.12. Effect on Memory Usage at cold and warm start call for multiple workloads . 43
      (a) x86-vs-ARM64 at cold start ..................................................... 43
      (b) x86-vs-ARM64 at warm start ................................................... 43

5.13. Effect on Memory Usage at x86 and ARM64 architecture for multiple workloads 44
       (a) cold-vs-warm at x86 arch. ...................................................... 44
       (b) cold-vs-warm at ARM64 arch. .................................................. 44

5.14. Effect of Input Size on Execution Time ........................................... 45

5.15. Effect of Input Size on Client Time .............................................. 46
5.16. Effect of Input Size on Provider Time ........................................ 46
5.17. Effect of Input Size on Memory Usage .................................... 47
5.18. Effect on Connection Time at various memory sizes for x86 and ARM64
       Dynamic-HTML .................................................. 48
       (a) x86-vs-ARM64 at cold start for Dynamic-HTML ............... 48
       (b) x86-vs-ARM64 at warm start for Dynamic-HTML ........... 48
5.19. Effect on Connection Time at 128MB for various workloads .......... 49
       (a) x86-vs-ARM64 at cold start .................................. 49
       (b) x86-vs-ARM64 at warm start .................................. 49
5.20. Effect on Connection Time at cold and warm start call for various workloads . 50
       (a) cold-vs-warm at cold x86 architecture .................... 50
       (b) cold-vs-warm at ARM64 architecture ...................... 50
5.21. Execution time Cold startup overhead for uploader .................. 50
5.22. Provider time Cold startup overhead for uploader .................. 51
5.23. Execution time Cold startup overhead for multiple workloads ...... 52
A.1. AWS Login Page .................................................... 57
A.2. Steps For Getting Keys .............................................. 58
B.1. Setting Environment Variables on a Ubuntu Host ..................... 59
B.2. An environment setup ................................................ 60
C.1. Effect on Execution Time at various memory sizes for x86 and ARM64 for uploader 62
       (a) x86-vs-ARM64 at cold start for uploader .................. 62
       (b) x86-vs-ARM64 at warm start for uploader ................ 62
C.2. Effect on Client Time at various memory sizes for x86 and ARM64 for uploader 62
       (a) x86-vs-ARM64 at cold start for uploader .................. 62
       (b) x86-vs-ARM64 at warm start for uploader ................ 62
C.3. Effect on Provider Time at various memory sizes for x86 and ARM64 for uploader 63
       (a) x86-vs-ARM64 at cold start for uploader .................. 63
       (b) x86-vs-ARM64 at warm start for uploader ................ 63
C.4. Effect on Execution Time at various memory sizes for x86 and ARM64 for compression
   (a) x86-vs-ARM64 at cold start for compression
   (b) x86-vs-ARM64 at warm start for compression

C.5. Effect on Client Time at various memory sizes for x86 and ARM64 for compression
   (a) x86-vs-ARM64 at cold start for compression
   (b) x86-vs-ARM64 at warm start for compression

C.6. Effect on Provider Time at various memory sizes for x86 and ARM64 for compression
   (a) x86-vs-ARM64 at cold start for compression
   (b) x86-vs-ARM64 at warm start for compression

C.7. Effect on Connection Time at various memory sizes for x86 and ARM64 for compression
   (a) x86-vs-ARM64 at cold start for compression
   (b) x86-vs-ARM64 at warm start for compression

C.8. Effect on Connection Time at various memory sizes for x86 and ARM64 for uploader
   (a) x86-vs-ARM64 at cold start for uploader
   (b) x86-vs-ARM64 at warm start for uploader
Nomenclature, Symbols, Acronyms

AWS  Amazon Web Services
FaaS  Function as a Service
IaaS  Infrastructure as a Service
FaaS  Function as a Service
SaaS  Software as a Service
BaaS  Backend as a Service
CPU  Central Processing Unit
SNS  Simple Notification Service
SQS  Simple Queue Service
ELB  Elastic Load Balance
S3  Simple Storage Service
API  Application Program Interface
HTTP  Hyper Text Transfer Protocol
IAM  Identity Access Management
CLI  Command Line Interface
CRM  Customer Relation Management
Abstract

Function as a Service (FaaS) is gaining admiration because of its way of deploying the computations to serverless backends in the different clouds. It transfers the complexity of provisioning and allocating the necessary resources for an application to the cloud providers. The cloud providers also give an illusion of always availability of resources to the users. Among the cloud providers, AWS serverless platform offers a new paradigm for developing cloud applications without worrying about the underlying hardware infrastructure. It manages not only the resource provisioning and scaling of an application but also provides an opportunity to reimagine the cloud infrastructure as more secure, reliable, and cost-effective.

Due to the lack of standardized benchmarks, serverless functions must rely on ad-hoc solutions to build cost-efficient and scalable applications. However, with the development of the SeBS framework, we can test, evaluate and do performance analysis of different cloud providers. Various researches have been conducted to differentiate the serverless platforms among the cloud providers. However, there is no research conducted so far within the AWS Lambda service in ARM64 architecture and between its different CPU architectures (x86 and ARM64).

Thus in this thesis, we have analyzed the perf-cost, latency, and cold startup overhead for both x86 and ARM64 architecture. We have conducted a meticulous evaluation of the perf-cost analysis in different sections. Our results show that increasing the code size and complexity directly affects the perf-cost metrics in both x86 and ARM64 architecture. However, at each invocation, either cold or warm startup, ARM64 is performing better than x86. Furthermore, our work showed the behavior of cold and warm startups at each architecture for any specific workload.
Taking the viewpoint of a serverless user, we also conduct experiments to show the
effect of complexity on memory usage at both x86 and ARM64 architecture. We found
that each architecture consumes nearly the same amount of memory for any particular
workload regardless of invocation methods -cold and warm. In addition, we observed that
cold invocation and ARM architecture would be efficient configurations for any specific
workload regarding memory usage.

Our analysis also shows that the input size directly impacts perf-cost metrics. Re-
garding the latency, ARM64 needs less time than ARM64 irrespective of invocation methods.
However, if we look closer, a warm startup’s latency is less than a cold one. Therefore, the
most efficient configuration for any specific workload would be a warm invocation and ARM
architecture.

Similarly, in the case of cold startup overhead, our results illustrate that for any spe-
cific workload, ARM64 has lower execution and provider time overhead than x86. However,
these overheads decrease with the increment of complexity due to high memory consumption
at higher complex workloads. Therefore, we can say that our work and results provide a fair
and transparent baseline for the comparative evaluation of each AWS architecture. Over-
all, this thesis has provided us with a great learning opportunity in serverless computing
assessment.
Chapter 1. Introduction

Cloud computing is a network of computers or servers worldwide connected via the internet. It consists of two parts: the front end and the back end. The front end is where users access the internet via computers, applications, or any client tools to access the cloud computing system. However, the backend is the section where all the hardware infrastructures necessary for cloud computing resides [18]. It is easy to do maintenance of infrastructure at the backend because the hardware parts are separated from the development of an application. This chapter explains the basic concept of serverless, serverless computing, types of cloud computing services and provides a straightforward thesis layout.

1.1. Types of Cloud Computing services

Before cloud computing, we need to know about the cloud and its types. Cloud deployment consists of three models: public, private, and hybrid [7].

Private Cloud:
This type of model has an infrastructure that belongs to a single business, and infrastructures are hosted either in-house or externally. Even though it is expensive, the private cloud is one of the best options for organizations whose main priority is security, computing power, and customizability.

Public Cloud:
All the organizations share this model’s infrastructure. It has vast space and is easily scalable where users will pay models based on a per-pay-use. It is generally provided by third-party providers like Amazon Web Services, Salesforce, Microsoft Azure, Google Cloud, etc.

Hybrid Cloud:
It is a combination of both public and private clouds, and it is highly secure and economical. There may be some conflict during the communication between private and public cloud as it combines two models to create an efficient solution.

Similarly, cloud computing services mainly consist of three groups: Platform-as-a-Service (PaaS), Infrastructure-as-a-Service (IaaS), and Software-as-a-Service (SaaS) [1].

![Diagram of Types of Cloud Computing Services]

Figure 1.1. Types of Serverless Cloud Computing Services

**IaaS:**

It provides on-demand cloud computing infrastructures to the organization based on pay-as-you-go via the internet. It includes virtual machines, servers, storage, networks, and OS facilitated by cloud providers. However, there is a data security issue associated with its multi-tenant architecture.

**PaaS:**

In this model, developers can lease the needed cloud computing infrastructure for a complete life-cycle of an application: development phase, testing phase, deployment phase, and maintenance phase. It is mainly created to provide an accessible environment so that developers can build mobile and web applications quickly without concern about software infrastruc-
ture. The main drawback of PaaS is that its support, reliability, and speed highly depend on the vendor.

**SaaS:**

It is the simplest model of all. Different providers give us access to its infrastructure over the cloud via API or web browser. It is not mandatory to install anything on the host computer to access the SaaS products. Some examples are Gmail, outlook, customer relation management (CRM) software of Salesforce, Jira, Trello, etc. The main disadvantage of SaaS is that we must have network connectivity to use SaaS, and we have lost control while using SaaS solutions [2].

We can also view cloud computing in a pyramid structure.

![Cloud Computing Structure](image)

**Figure 1.2. Cloud Computing Structure**

1.2. **Serverless**

There is a misconception that serverless means that there is no server. However, it simply means developers who deal with business logic do not have to care about the server
behind the scene. In other words, developers do not worry about creating, maintaining, and deploying a server. That is why it is called serverless [6].

**Serverless Evolution:**

The emerging popularity of containers and on-demand cloud computing infrastructure offered by cloud providers worked as a catalyst to grow the serverless architecture concept and serverless computing hand-in-hand. If we try to trace serverless evolution, we can find its three phases.

The phase “Serverless 1.0” came with many limitations, making it unsuitable for general computing. It has only HTTP and few other resources. The execution time is also limited (5-10 minutes), there is no orchestration, and the local development experience is minimal.

The “Serverless 1.5” era emerged with the advent of Kubernetes, where containers can be auto-scale in many serverless frameworks. It is characterized by Kubernetes-based auto-scaling, Microservices, and functions, easiness of debugging, and can be tested locally. Most importantly, it is portable.

The “Serverless 2.0” era is the most current and emerging with a summation of state and integration. Many cloud providers have found a solution to make serverless more suitable for general-purposes workloads. It is blended with enterprise PaaS. Moreover, it is characterized by basic state handling, enterprise integration patterns, etc [12].

**Serverless Architecture:**

Serverless contradicts other cloud computing models in a concept where cloud computing providers are responsible for scaling apps and managing the infrastructure. However, serverless apps deployed in the containers are automatically scaled up or can be scaled down according to their needs. One of the most straightforward serverless architecture patterns
in AWS is shown below diagram Figure 1.3. Here, the API gateway asynchronously invokes Lambda functions, and data are retrieved or saved from other AWS services.

Under a standard IaaS, users pay public cloud providers for server components that are always on to run the applications. Moreover, it is always the user’s responsibility to scale up server capacity during the high demands and scale down when such power is no longer necessary. The main disadvantage is that infrastructure always needs to be active even though an application is not running.

However, in serverless architecture, applications are invoked only when needed. With the help of a trigger event, a cloud provider runs an application code by allocating the resources dynamically. The significant benefit is that users will stop paying when the code has finished its execution. Moreover, it also allows developers to focus on application development without worrying about menial and routine tasks associated with server provisioning and its scaling. In addition, cloud providers also provide everyday tasks such as security patches, managing OS and file systems, capacity management, load balancing, and monitoring [12].

1.2.1. Serverless Computing

Serverless computing consists of Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS). BaaS provides access to developers to a variety of third-party apps and
services. For example, authentication service, extra encryption, high-fidelity data usage, cloud-accessible databases, etc. With BaaS, APIs are highly used to call serverless functions. However, when developers talk about serverless, they refer to FaaS Model, where they write custom server-side logic even though cloud providers fully manage it. FaaS is a way to carry out serverless computing where developers focus only on writing code and conveying the business value. In addition, it is an event-driven execution model that executes functions in stateless containers.

1.2.2. Pros and Cons of Serverless Computing:

Serverless computing has its advantages and disadvantages, and some are discussed below.

Pros:

- It increases the productivity of a developer and minimizes operational costs. Developers can better utilize their time to focus on their apps by offloading routine tasks such as managing servers and provisioning.

- It helps in enabling the adoption of DevOps by drastically reducing the importance of developers to define the infrastructure they require for operation.

- With the help of BaaS offerings from third parties, application development can be entirely streamlined.

- As we are paying only for the time resources are used, the operational costs are highly reduced.

Cons:
• We cannot control our server-side logic as we are not running it.

• It does not have a persistent state as an event triggers it.

• There is a flexibility and customizability issue as cloud providers can have constraints on their components. Similarly, in the BaaS environment, developers can view their services, but the code is out of their control.

1.3. Motivation and Thesis Goal

Although there are many other cloud platforms like Azure, Google Cloud, Alibaba, and many other open source serverless platforms like OpenWhisk, AWS of an Amazon is the most popular one. According to Katy, AWS alone occupies 32% of the global market, followed by Azure at 19%, Google Cloud at 7%, and the rest occupied by others [17]. There are hundreds of services in AWS that are suited for various applications and the needs of users and organizations. In addition, AWS has recently launched a new feature, “ARM64” - a 64-bit processor architecture in its Lambda service. Lambda is a serverless feature of AWS. Earlier, there existed only x86, a 64-bit processor. AWS claims that ARM64, a Gravitation2 processor, has better price and performance than x86 [11]. Different researches have been conducted to differentiate the serverless platforms provided by other cloud providers using various metrics. However, there is no research conducted so far within the AWS Lambda service in ARM64 architecture and between its different CPU architectures (x86 and ARM64). Therefore, we are highly motivated by this opportunity. This thesis aims to conduct this research to find out results in ARM64 architecture and identify the efficiency between x86 and ARM64 architectures. We are using various tested
and BSD-3 clause licensed workloads in AWS. To invoke such benchmarks synchronously in AWS, we will use HTTP API Gateway.

1.4. Thesis Layout

The thesis layout is as follows: Chapter 2 reviews the background and related work that has been done earlier and will explain how data has been collected for this research. Chapter 3 discusses the concept of Amazon Web Services (AWS), Lambda functions, and invoke mechanisms. Chapter 4 starts with building a Lambda function in AWS and how data has been collected for various workloads using different metrics. Chapter 5 will explain the results so obtained and will do some analysis. Finally, Chapter 6 deals with discussion, conclusion, and future work.

Moreover, appendices are also included to support this thesis work. Appendix A discusses how to get the AWS secret keys essential to invoke the benchmarks in AWS Lambda. Similarly, Appendix B will provide some additional information about the requirements that need to be installed on the host (user computer - Linux) and how to set up some environment variables. Finally, in Appendix C, we have discussed how to invoke workloads, how the experiment run and its results have been processed, and provide additional plots supporting our results.
Chapter 2. Background

An application has one or more functions in serverless computing platforms. Such functions are usually standalone, small, and stateless components programmed to handle particular tasks. A function can be defined as a code written in some specific programming or scripting language. Moreover, it executes within a particular instance called function instance. A function instance may be a container or a sandbox with a limited amount of memory and CPU time provided by public cloud providers [15].

One or more function instances will be launched when it is invoked by a function request to execute the function. Such function instance(s) will become idle once function instance(s) processed the request, exited, or reached its maximum execution time limit. Function instance(s) are reusable to handle another subsequent request to circumvent delays in launching new instances. However, the function instance(s) can also be immediately terminated if unused or idle. Moreover, a non-persistent local disk, which stores the data temporarily, linked with a function instance, will also be erased once such instance(s) is terminated. The main benefit of serverless computing is users are charged only when function instances consume resources [3].

Serverless computing providers like AWS, Azure, Google Cloud, etc., maintain and manage the execution of application environments, support backend computers/hardware or servers of a function, and dynamically allocate the resources to ensure their availability and scalability during failover and high demands respectively.

In traditional IaaS platforms like virtual machines (VMs), hardware always needs to be turned on for a function, although it is not used. However, a function instance or server is launched only when a particular Lambda function is invoked in modern serverless computing
platforms. After servicing or handling the request, function instances will immediately be put to sleep.

In such scenarios, users are charged only on a per-invocation basis, and users are not bound to pay for idle and unused resources. It is the beauty of modern serverless computing. Serverless computing originated from handling low duty-cycle workloads[20]. For example, they are processing a request in response to rapid changes to cloud storage files.

2.1. Related Work

Function as a Service (FaaS) is gaining popularity exponentially due to its ability to deploy the computation to serverless backends in the public cloud. This paradigm moved the intricacy of provisioning and allocating the resources to public cloud providers. Moham-mad et al. [15] have characterized the FaaS workload of Azure Functions at the production level and proposed the resource management policy that reduces the number of cold start functions while spending low resources. They have gathered the data on all the invocation functions across Azure’s infrastructure. Mainly, they have collected data in four sets: first - trigger per function, second - invocation counts: per function, and third - execution time per function: average, min, max, and count of samples for every 30-second interval. Finally, the memory usage per application is sampled every 5 seconds. By doing result analysis, they showed that even with lower resource cost, their policy can achieve the same or reduced number of cold start. This allowed us to experiment with AWS Lambda functions between x86 and ARM64 architectures.

Wang et al. claimed their SIREN framework could achieve higher parallelism and elasticity using a swarm of stateless functions on the cloud. They introduced the Deep Reinforcement
Learning method to control the memory and number of stateless functions. Moreover, they compared the cost and performance of training the ML models on AWS EC2 clusters versus training on AWS Lambda using the SIREN framework [19]. AWS Lambda, by default, will choose only x86 architecture. However, AWS recently launched new architecture, ARM64, which could be the new area of research regarding performance and cost.

Apart from private cloud providers, some research has also been conducted on open source serverless computing like Openwhisk. Yu et al. introduced a new Resource Manager for serverless platforms, Freyr. Freyr dynamically harvests idle resources from over-provisioned to under-provisioned functions to increase resource efficiency. Using a deep reinforcement learning algorithm, it monitors the resource utilization of each function and safely harvests the idle resource [21].

Similarly, many serverless application developers have conducted multiple experiments and researches to measure CPU usage in AWS Lambda, function instance lifetime, cold start latency, etc. Liang et al. showed one of the most extensive experiments by launching 50,000 function instances across three leading cloud providers to measure their resource management efficiency and performance. They have characterized the performance in resource efficiency, cold start latency, and scalability. They have duplicated the experiments under different settings during their data collection by adjusting workloads and function configurations to find critical factors affecting measurement results. They have used a measurement function - a single function consisting of all the subroutines and necessary design, to perform two tasks—one to collect runtime information and invocation timing. Second, to measure the throughput using specific subroutines. Although they have provided insights into performance and resource utilization across three modern serverless platforms, they have not
shown the results between two architectures supplied by the AWS Lambda [20].

However, Anthony et al. [3] have experimented on a MicroFaaS prototype against traditional serverless platforms to compare the results between x86 and ARM-based single-board computers. They have developed the prototype based on the hypothesis that serverless functions better opt for low-overhead and smaller execution environments than conventional infrastructures. Those conventional infrastructures are highly virtualized and multiplex environments. In a traditional serverless cluster, they have replaced many large multi-core rack servers with many single-core nodes. Each node has only bare minimum software - just enough to run OS like Linux containers. They have developed a single-tenant, run-to-completion model where no other function can run until the assigned function has finished its execution. They have conducted a complete evaluation and thorough cost analysis using this approach between two architectures. They found that energy efficiency increases by 5.6x, and total cost decreases by 34.2%. However, this model’s optimization in the real market has been discussed. Therefore, we want to take this advantage. Due to the cheap, popularity, and high demand of AWS, we want to conduct experiments on AWS Lambda to show the results between x86 and ARM64 architectures that will bring more users or organizations to this platform.

2.2. Data Collection

Serverless functions are the new middleware for building or developing cost-efficient and scalable applications. But due to the scarcity of standardized benchmark suites, we incur to use micro benchmarks and ad-hoc solutions in serverless research. Marcin et al. developed a serverless benchmark suite -SeBS to address this issue or challenge that covers
a broad spectrum of applications and cloud resources. SeBs (Serverless Benchmark Suite) is one of the diverse suites that help in performance analysis of public cloud providers like AWS, Azure, Google Cloud, and open source platforms like OpenWhisk.

Benchmarking an application is a process of measuring the quality and number of application features and then comparing it to other similar applications in the industry. SeBS provides a suite of benchmark applications and experiments tested, verified, and licensed BSD-3. Thus we are using the opportunity to collect our data for both x86 and ARM64 architecture of the AWS Lambda service. We have collected the data in 3 different categories: perf-cost, latency, and cold startup overhead. The details of data collection and experiments are discussed in Chapter 4.
Chapter 3. Amazon Web Services

Amazon Web Service (AWS) is a public cloud platform that provides cost-effective, reliable, and scalable cloud computing services to users or organizations compared to its competitors like Azure, Google Cloud, etc. It is adopted broadly and offers on-demand operations like database storage, content delivery, compute power, etc., to assist corporates in growing and scaling. There are numerous applications of AWS, and some of them are storage/Backup, web application, best online gaming experience, and mobile, web, and social applications [16]. This chapter mainly focuses on AWS Lambda, its architecture, benefits, and invocation mechanisms.

3.1. Advantages of AWS

The potential of AWS lies in its ability to reach the marketplaces with low investment. Some of the advantages of AWS are discussed below:

Easy to Use

AWS provides a user-friendly interface known as a Management Console when we finish the sign-up process. Hundreds of services can be launched instantly, and it removes the necessity of an on-site server where we can deploy an application or a complete IT ecosystem promptly.

Security

There is a misunderstanding that data stored in public clouds are vulnerable. On the contrary, AWS not only provides security tools at a cheaper rate than its competitors, but it is one of the most extensive, reliable, and secure cloud platform.

Global Availability

Currently, AWS has 84 availability zones across 26 geographic regions worldwide. It has
recently announced to expand 24 more availability zones and eight more AWS regions. Each AWS region has one or more data centers called an availability zone.

**Scalability and Flexibility**

AWS provides limitless scalability and flexibility on demand, and it qualifies organizations to plan a roadmap for their infrastructure on a subscription basis, even without commitment. In addition, AWS allows us to pay only for those resources being used.

3.2. AWS Services

AWS offers hundreds of services for cloud applications. Some of the primary or highly used services that the AWS ecosystem provides are compute service, storage, database, monitoring tools, security tools, developer tools, etc. We will discuss only those services in this chapter that were used during our research.

**AWS CloudWatch**

CloudWatch is a monitoring tool that monitors customers’ resources and applications running on the AWS platform. It provides a single interface to gather and access all operational data in the form of logs. It helps to gain system-wide visibility into application performance, resource utilization, and operational health. We used this service to debug the issue that occurred during the invocation of Lambda functions and run the experiments.

**AWS S3**

Amazon Simple Storage Service (S3) is an open cloud-based storage service that provides data availability, industry-leading scalability, performance, and security. It has many utilities like backing up online data, storage, invoking the Lambda functions, etc., and retrieving the data any time from anywhere. We have used this service to store necessary objects while
invoking the Lambda functions and running the experiments. In addition, we use AWS Lambda extensively, which we discuss in detail in a separate section.

### 3.2.1. AWS Lambda

AWS Lambda is an event-driven compute service responsible for executing the application code of users. It executes the program without worrying about the managing servers at the backend. It is also called serverless because we do not need to maintain our servers to run functions or applications. Moreover, AWS Lambda can perform many computing tasks like processing data streams, serving web pages, integrating with other AWS services, etc.

**How does the AWS Lambda work?**

To run each Lambda function, it needs its container. Lambda will package function into a new container whenever it is created and then executes that container on backend servers managed by AWS. We can also visualize such containers at the backend from the Figure 3.2. Before starting the function’s execution, each function’s container is provided with the necessary CPU and RAM. Once the function is executed, allocated RAM at the beginning is multiplied by the functions’ time spent to execute. Then a user or customer is charged based on the run time of a function to execute and its allocated memory. Figure 3.1 shows the above explanation.

AWS manages the complete infrastructure layer of AWS Lambda. Users or customers do not get enough visibility of how the system works behind the scenes. However, AWS will take care of updating the underlying machines, network congestion, and so on - we do not need to worry about it. Moreover, we do not need to spend time on operational tasks as AWS Lambda fully manages the service. Developers can spend more time on application
code as they are not supposed to focus on maintaining the infrastructure. But, this will lose our control of the flexibility of operating our infrastructure. One of the unique architectural properties of AWS Lambda that differentiate it from other services is that we can...
concurrently execute many instances of the same function or different functions from the same AWS account. Moreover, we get charged only for computing function - even though concurrency can vary during daytime or day of the week, such variation does not affect the Lambda [13].

**Why is Lambda a necessary part of the serverless architecture?**

Lambda is one of the primary candidates to run an application code while building a serverless application. Generally, a serverless stack consists of: a compute service, a database service; and an HTTP API gateway service. Lambda plays a significant and primary role as a compute service. It combines with API gateway and many other AWS services like DynamoDB, RDS, etc., forming the essence of a serverless solution. Lambda is a good solution for many serverless developers as it supports many most popular languages and runtimes.

**Supported architectures, languages, and runtimes**

Earlier, Lambda used to support only x86 architecture. Recently, AWS has announced ARM64 (AWS Galvatron2 processor) architecture for AWS Lambda. AWS claims that ARM64 delivers faster execution and gives more immediate results. It also asserts that ARM64 functions running on ARM64 processors provide 34% better price performance than x86 processors [9]. When writing this thesis, AWS Lambda supports many popular languages and runtimes. They are GO, Node.js, Powershell, Java, Python, C#, and Ruby Code [14].

<table>
<thead>
<tr>
<th>Name</th>
<th>AWS SDK for Python</th>
<th>Operating System</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python3.9</td>
<td>boto3-1.20.32 botocore-1.23.32</td>
<td>Amazon Linux 2</td>
<td>X86_64, ARM64</td>
</tr>
<tr>
<td>Python3.8</td>
<td>boto3-1.20.32 botocore-1.23.32</td>
<td>Amazon Linux 2</td>
<td>X86_64, ARM64</td>
</tr>
<tr>
<td>Python3.7</td>
<td>boto3-1.20.32 botocore-1.23.32</td>
<td>Amazon Linux</td>
<td>X86_64</td>
</tr>
<tr>
<td>Python3.6</td>
<td>boto3-1.20.32 botocore-1.23.32</td>
<td>Amazon Linux</td>
<td>X86_64</td>
</tr>
</tbody>
</table>
Similarly, we are using the python language and its runtimes in this research. The supported runtimes and their architecture is shown in Table 3.1.

**Advantages of Using AWS Lambda**

AWS Lambda has some distinctive benefits over maintaining its servers. Some of them are:

*Pay Per Use:*

In AWS Lambda, we are supposed to pay only for the function’s runtime. This type of billing is generally cost-effective for those whose workloads will scale up significantly during the daytime.

*Fully Managed Infrastructure:*

AWS manages all the underlying infrastructure that is necessary for function. This results in significantly saving operational cost tasks such as managing network layer, upgrading OS, etc.

*Automatic Scaling:*

Lambda creates an instance of a function when the request is received. The underlying infrastructure for that function will automatically scale up when the load increases. We do not need to worry about it as AWS manages all necessary stuff behind the scene, and we only pay for each function’s runtime.

*Integration with Other AWS Services:*

Lambda also integrates with services like API gateway, DynamoDB, S3, etc., enabling us to build a complete application functionality.

**Limitation of AWS Lambda**

Besides many advantages, Lambda has a few limitations as well [12].

*Cold Start Time:*
Whenever a function is started in response to an event, there is a small latency between the event and the actual function run. The latency can reach 5 - 10 seconds if a function is not being used for the last 15 minutes. This makes latency-critical applications hard to rely on Lambda.

Function Limits:

There are a few limits in the Lambda function, which are listed below:

(a) A Lambda function will time out after running for 15 minutes, and such a time limit is unchangeable. So, if a Lambda function needs more than 15 minutes to execute, Lambda is not a solution.

(b) The range of RAM available for any function is 128MB to 3008MB with a step of 64MB.

(c) The unzipped Lambda package should not exceed 250MB, and the zipped version should not be large than 50MB.

(d) By default, each AWS account supports 1000 concurrent Lambda function execution. Lambda functions triggered above this currency will have to wait until other running functions finish their execution.

(e) The maximum 10MB of API gateway payload is allowed to trigger the Lambda function.

3.3. Cold and warm Start Call of Lambda Functions

AWS Lambda brings many benefits to the deployment of web applications, such as availability, auto-scaling, fine granularity on cost, and the absence of server management. However, inconsistent startup performance makes Lambda users frustrate. AWS packages the Lambda functions inside the container and runs at the back-end server code. But, to execute its tasks, AWS must spin up the server, and this preparation consumes some time
before the code runs. This time is known as a cold start.

A cold start occurs whenever a fresh container is required to run a Lambda function. This new container has to warm up before the event handler code run. Generally, this process can take from 400 milliseconds to a few seconds. However, AWS does not immediately terminate the container even after it has finished the event task. Such containers can be reused again to rerun the function, and users do not have to pay to keep the containers warm [4]. Therefore, the warm start has a container whose layers, runtime, and initialization is already complete so that Lambda functions can rerun. That means the event handler code can run
immediately, which benefits us from paying only when the event handler runs. Moreover, multiple warm containers are needed to run multiple concurrent invocations. The Figure 3.3 also depicts the difference between cold and warm start calls [8].

3.4. AWS Lambda Invocation

Triggering a Lambda function is called an invocation of the Lambda function. Sometimes, we invoke via HTTP; some are triggered based on different events. It is beneficial to fathom different ways of invoking the Lambda functions. There are three types of invocation systems that exist in AWS Lambda today [10].

Synchronous Invocation

It is a simple push model. Whenever we invoke Lambda via API call, the Lambda function executes immediately and waits for a response. Response data are retrieved from DynamoDB and some other AWS services. A standard scheme of synchronous innovation is API Gateway + Lambda integration which is shown in Figure 3.4.

The invocation type flag should be RequestResponse, which instructs AWS to execute the Lambda function and wait for the function’s response to complete. Services that invoke synchronously are Amazon HTTP API Gateway, Amazon CloudFront, Amazon Kinesis Data Firehose, Elastic Load Balancer (ELB), Amazon Cognito, Alexa, and Lex.
Asynchronous Invocation

In an asynchronous invocation, Lambda functions are invoked by events and do not wait for a response. During the asynchronous invocation, Lambda sends the event to its internal queue and returns a successful response without any additional information. After that, a different process reads the event from the queue to execute the Lambda function. A general scenario for asynchronous invocation is S3/SNS + Lambda + DynamoDB, shown in Figure 3.5. In the Figure 3.5, whenever a new object is written to the S3 bucket, it will call the Lambda function asynchronously. A parameter event must be set in the invocation type to call the Lambda function asynchronously. The AWS services that support asynchronous invocations are S3, SNS, SES, CloudWatch Logs, CloudWatch Events, CloudFormation, and EventBridge.

Event Source Mapping with Polling Invocation

This pool-based invocation model does not invoke the function directly but instead integrates with AWS streams and Queue based services. Here, Lambda polls from AWS Kinesis...
streams or SQS to retrieve their records and then invoke the function. Data queues or streams will be read in batches while executing the function; it receives multiple items. The batch size can be configured according to service type and will split up if its size is too large. A general scheme of this mode is SQS + Lambda, which is depicted in the Figure. Here

![Event Source Mapping with Polling Invocation](image)

Lambda function pools events from SQS and receive queue records for invocation. SQS will get queue records from other services like SNS, EventBridge, etc. The service that uses this model is Kinesis, DynamoDB, and Simple Queue Service (SQS).

3.5. Conclusion

This chapter briefly discusses the Amazon Web Service (AWS). The reason for choosing AWS is its ease, security, global availability, scalability, and flexibility. In addition, AWS provides hundreds of services according to the client’s necessity. This chapter also focuses on the AWS Lambda function, which is the core of our thesis experiment. It answers why Lambda is a necessary part of the serverless architecture and what are the advantages of using AWS Lambda. The beauty of AWS Lambda is that it supports many popular languages and recently added the new AWS Galvatron2 processor (ARM64), which allowed us to evaluate the serverless computing between x86 and ARM64 architecture. Furthermore,
we have discussed the Lambda function’s cold and warm start call and different types of
AWS Lambda innovation methods.
Chapter 4. Methodology

The Serverless Benchmark Suite (SeBS) is an accurate and consistent methodology that compares serverless providers in terms of reliability and performance to various workloads. It is a complete framework for building, deploying, and invoking the Lambda functions in different cloud platforms and open-source platforms like OpenWhisk [5]. Therefore, we have used this opportunity to evaluate SeBS on two AWS Lambda architectures, x86 and ARM64. This chapter answers: How to set up an environment for our research work, how to do regression tests, how to invoke benchmarks, how to run different experiments, and finally, how to process the results.

4.1. Creating an Environment

We have run all our experiments in the AWS region us-east-1. We used S3 for persistent storage, HTTP endpoints as function triggers, and deployed Python 3.7 for x86 and 3.9 for ARM64 architecture, respectively. A simple model or topology we used in this research is shown below. The steps to get AWS access and secret keys necessary to access the AWS Lambda service are shown in Appendix A. Similarly, The requirements that need to be installed on the server/host and setting up environment variables are also explained in

Figure 4.1. A simple Experimental Topology
detail in Appendix B.2. Finally, a user must have a Putty tool installed to access the server remotely.

4.2. Regression Testing

The SeBS offers three basic commands: benchmark, experiment, and local. We can also pass –verbose flag in each command to expand the verbosity of the output. Moreover, all the commands must run inside the python virtual environment. A regression test offers to execute all benchmarks on an AWS Lambda. An illustration of running a regression with an input size test on AWS Lambda is shown below.

```
./sebs.py benchmark regression test –config config/example.json –deployment aws –verbose
```

Similarly, we can also execute a regression test on a single benchmark. An example is shown below.

```
./sebs.py benchmark regression test –config config/example.json –deployment aws –benchmark-name 120.uploader –verbose.
```

![Figure 4.2. Regression test for a single benchmark](image)

By default, all the Lambda functions are created and invoked via HTTP on x86 architecture. Two things must be considered before running the Lambda functions on ARM64 architecture. First, we must run the functions on python3.9 because ARM64 is supported only from python3.8. The runtime environment of a Lambda function can be changed by editing the config/example.json file, as shown in Figure 4.3. After the Lambda function is executed with runtime python3.9, the ARM64 architecture can be chosen from the AWS
management console after clicking on edit run time settings, as shown in the Figure 4.4.

Figure 4.4. Changing Architecture to ARM64

4.3. Benchmark Invocation

SeBS offers a command to build, deploy, and execute various serverless benchmarks in the AWS cloud. An example is shown below: it invokes 120uploader on AWS via HTTP
trigger.

`./sebs.py benchmark invoke 120.uploader test --config config/example.json --deployment aws --verbose`

we can also view the complete list of available options in the Figure 4.5 for the benchmark by running the command: `./sebs.py benchmark invoke --help`.

As explained in the regression testing, a similar approach can be followed to invoke the benchmarks on ARM64 architecture. Moreover, we can invoke benchmarks with various input sizes like `tiny`, `small` and `large`. 

Figure 4.5. Available Options for Benchmark Command
4.4. Running Experiments

SeBS also offers a command to execute the benchmarks in the AWS cloud. It provides four different types of experiments: perf-cost, network-ping-pong, invocation-overhead, and eviction mode. An example of running the perf-cost experiment is shown below:

```bash
./sebs.py experiment invoke perf-cost --config config/example.json --deployment aws --verbose
```

There is also a complete list option available to run the experiments. A snapshot is depicted in the Figure 4.6.

Moreover, we can also play with different parameters in the configuration file (config/example.json) while running the experiment. Such parameters are experiment type, a benchmark name, input-size, concurrent invocation, memory sizes, and repetitions. A snapshot of the configuration file is shown in the Figure 4.7.

We can also run and obtain the results for different architectures for each experiment by
4.5. Conclusion

This chapter explains how to create a simple experimental environment on a host or server. We have also described in detail for setting an AWS environment variables, installing requirements, and activating an environment necessary to communicate with the AWS cloud from our host/server. In addition, we also showed how to do regression testing, benchmark invocation, and run various experiments along with figures in x86 and ARM64 architecture.
Chapter 5. Result and Analysis

In this chapter, we have discussed the results obtained from running experiments for multiple workloads in detail. A separate lab setup is created for each architecture using two AWS accounts to remove confusion about the obtained results while making the plots. We have analyzed the results in terms of perf-cost, latency and overhead.

5.1. Perf-Cost

Cloud metrics are the set of metrics available in the cloud, which is limited due to the black-box nature of the FaaS system. So, for perf-cost analysis, we have focused on metrics like execution time, client time, provider time, and memory usage. Before running the experiments, connect to the server/host where the SeBS framework is installed via putty. Then, we need to activate our python virtual environment as below:

```
. python-venv/bin/activate
```

![Virtual Environment](image)

Figure 5.1. Virtual Environment

In the beginning, we have run the experiment in x86 architecture for the workload Dynamic-HTML using the below command:

```
./sebs.py experiment invoke perf-cost –config config/example.json –deployment aws –verbose
```

However, to obtain the desired perf-cost metrics, we have passed the inputs via configuration file `example.json`. The configuration file helps us get the inputs’ results in one shot. The
file (Figure 5.2) shows the experimenting nature (cold and warm), workload or benchmark name, memory sizes (128 to 3008MB), the number of repetitions and concurrent invocations that can be passed during the experiment.

![Figure 5.2. Configuration File example.json](image)

The obtained results (JSON file) are further processed to get the desired metrics and their value in CSV format using the below command:

```
./sebs.py experiment process perf-cost -config example.json -deployment aws -verbose
```

Similarly, to get results in ARM64 architecture for the same workload -Dynamic-HTML, we first changed python runtime to 3.9 in the configuration file `example.json`. Then, we used the management console of AWS to change the architecture to ARM64 from x86(default).

We have repeated the process above to get desired cloud matrices values in ARM64 for Dynamic-HTML.

In addition, we have repeated the above steps in both x86 and ARM64 architecture for other workloads like uploader and compression by changing the benchmark name in the configuration file `example.json`. To analyze the perf-cost in detail, we divided the topic (perf-cost) into multiple sub-topics explained below.
5.1.1. Behavior of x86 and ARM64 architecture at different memory sizes at cold and warm start call for Dynamic-HTML

Dynamic-HTML is a simple web application where dynamic features are offloaded to a serverless backend. It simply generates the Dynamic HTML from a predefined template. Moreover, execution time is the time the backend server takes to execute in the AWS cloud, including the work performed by the function. For the cold start, we need to provide the option cold to the parameter “experiments” in the file example.json. Similarly, use the option warm for the warm start, as shown in the Figure 5.2.

We have obtained the results for 50 to 100 concurrent invocations of Lambda function at static memory allocation ranging from 128MB to 3008MB. To understand the behavior of x86 and ARM64 architecture for Dynamic-HTML, we have plotted the graph (Figure 5.3) between the execution time and memory allocation at both cold and warm starts.

![Graph](image)

(a) x86-vs-ARM64 at cold start for Dynamic HTML (b) x86-vs-ARM64 at warm start for Dynamic-HTML

Figure 5.3. Effect on Execution Time at various memory sizes for x86 and ARM64 for Dynamic-HTML

At the cold start (a), it is clear that the Lambda function of Dynamic-HTML is taking more time to execute in x86 architecture than ARM64. However, at the warm start (b), x86
performs better than ARM64. Usually, AWS charges the bill for the total computational cost - duration plus declared or consumed memory. In other words, the longer the execution time, the more the billing amount will be. Therefore, x86 will be charged more at a cold start, whereas ARM64 will be charged more at a warm start.

Moreover, we have also analyzed the client time. Client time is the time that refers to the scheduling and deployment of a function at the client side. For this also, we ran 50 to 100 concurrent invocations and obtained results for static memory allocation ranging from 128MB to 3008MB. The Figure 5.4 illustrates the plot between client time and memory allocation in both x86 and ARM64 architectures.

![Figure 5.4](image-url)

(a) x86-vs-ARM64 at cold start for Dynamic HTML (b) x86-vs-ARM64 at warm start for Dynamic-HTML

The Figure 5.4 shows that the ARM processor performs better during the scheduling and deployment of a Lambda function than x86 for both cold and warm start at each memory allocation. So, regarding the client time, we can choose ARM64 architecture.

Finally, we observed the time required for cloud providers to add language overhead
and serverless sandbox. This time is called \textit{provider time}, and we average provider time after running 50 to 100 concurrent invocations for both cold and warm start at each memory allocation.

![Graph showing provider time versus memory size for x86 and ARM64](image)

(a) x86-vs-ARM64 at cold start for Dynamic HTML (b) x86-vs-ARM64 at warm start for Dynamic HTML

Figure 5.5. Effect on Provider Time at various memory sizes for x86 and ARM64 for Dynamic-HTML

The provider time versus memory size plot (Figure 5.5) showed that, at cold start, x86 architecture consumes more time to provide a serverless sandbox and add language overhead before executing a Lambda function than ARM64. And these results also hold for warm starts execution. Therefore, the billing amount will be less in ARM64 than in x86 regarding the provider time.

We observed similar results at execution time, client time, and provider time at both cold and warm start for other workloads - uploader and compression. Their plots are shown in Appendix C.1 and C.2, respectively.
5.1.2. Behavior of x86 and ARM64 architecture at cold and warm start call for different workload

Apart from the single and simplest workload - *Dynamic-HTML*, we have also observed the perf-cost metrics on multiple complex workloads: uploader and compression. *Uploader or storage uploader* is another web application that uploads a file from a URL to cloud storage, and its requirement and size are comparatively higher than Dynamic-HTML. Similarly, *compression* is a utility function that compresses a set of files and returns an archive to the users, similar to the online document text editors and office suites. These functions are used as backend processing tools when web servers or application frontend find complex problems, and its size and complexity is larger than the previous two.

Moreover, these workloads represent the code size and complexity, and we know that the size and complexity of dependencies directly impact the cold and warm start execution. That means the larger and more complex code package increases the warmup time of language runtime and deployment time from cloud storage.

During this experiment, we invoked each workload’s 50 to 100 concurrent functions at 128MB memory allocation. The Figure 5.6 depicts the effect on the *execution time* by the complexity of code.

From the Figure 5.6, at the cold start (a), we can see that increasing the code size and complexity increases the execution time. In addition, x86 architecture consumes more time to execute the function than ARM64 at each workload. This result also holds in warm start calls (b) where the ARM64 processor can operate in less time, decreasing the billing amount.

Similarly, at 128MB memory allocation, we processed results of *client time* of 50 to
Figure 5.7. Effect on Client Time at 128 MB for various multiple workloads

The Figure 5.7 tells us that at cold start, the client time is also increasing with the complexity and the size of the code, and the billing amount also decreases in ARM64 as it needs less time than x86. In addition, ARM64 did perform a similar performance at a warm start in
comparison to x86.

Finally, we also did analyze the provider time for different workloads. Here also, we run 50 to 100 concurrent functions with 50 repetitions each. After processing the result in CSV format, we took an average of it. We plotted the Figure 5.8 between provider time and complexity of code at static memory allocation of 128MB.

![Figure 5.8](image)

(a) x86-vs-ARM64 at cold start  
(b) x86-vs-ARM64 at warm start

Figure 5.8. Effect on Provider Time at 128 MB for various multiple workloads

The Figure 5.8 illustrates that regarding the provider time, the code size and complexity directly affect the provider time, and ARM64 performs better than x86 at both cold and warm starts. So, ARM64 is better than x86 regarding cost and performance about the provider time. Similar results can be seen for execution time, client time, and provider time at other static memory allocations.

5.1.3. Behavior of cold and warm start at x86 and ARM64 architecture for different workloads

So far, we have analyzed x86 versus ARM64 architecture for an individual workload and among the workloads. However, we have not seen cold and warm start behavior in
each architecture (x86 and ARM64). We know a cold start occurs when a new container is required to run a Lambda function which has two components. The first is the time AWS takes to set up an execution environment; another is code initialization duration -managed by the developer. Since the container does not terminate immediately after a function’s execution, such a container can be reused again, directly impacting execution time, client time, and provider time.

At first, we analyzed the effect on execution time by the complexity of code size at cold and warm start call for x86 and ARM64 architecture separately. For this, we have run the experiment with 50 to 100 concurrent invocations of each workload’s Lambda function repeating 50 times each at a static memory allocation of 128MB. The Figure 5.9 depicts the behavior of cold and warm start in each architecture.

![Graphs showing cold vs. warm start for x86 and ARM64 architectures](image)

(a) cold-vs-warm at x86 for multiple workloads  
(b) cold-vs-warm at ARM64 for multiple workloads

Figure 5.9. Effect on Execution Time at cold and warm start call for various multiple workloads

From the Figure, it is clear that the execution time increases with increases in the code size and complexity at both architectures. In addition, the cold start at each architecture takes
longer to execute the function than the warm start for each workload.

Moreover, we have analyzed the client time at x86 and ARM64 architecture with the same configuration, i.e., 50 to 100 concurrent invocations with 50 repetitions of each function at the static memory of 128MB. The Figure 5.10 shows a plot of client time versus workloads at cold and warm start calls.

The Figure 5.10 tells us that code size and complexity directly impact client time at both x86 and ARM64 architecture. In addition, a warm start call performs better than a cold one because the hardware remains on standby at a warm start.

Finally, we observed the provider time at both x86 and ARM64 architecture. We used the same configuration setting as in execution time and client time. The result so obtained from the experiment, we processed the results, took the average, and plotted the diagram - provider time versus workloads as shown in the Figure 5.11.

Like other perf-cost metrics, the code size and complexity directly affect the provider time at both x86 and arm64 architecture. Furthermore, the Lambda functions in cold start envi-

Figure 5.10. Effect on Client Time at cold and warm start call for various multiple workloads
Figure 5.11. Effect on Provider Time at cold and warm start call for various multiple workloads

environments took longer than in the warm environment. Therefore, the most significant configuration for any specific workload is ARM64 architecture with a warm start call because it will increase the performance while decreasing the bills.

5.1.4. Effect of Complexity of Code on Memory Usage for Different Architecture

Memory is also one of the critical parameters or metrics that can impact performance and cost. The peak memory usage is crucial in determining application configuration, billing policies, and execution settings. It helps the developer and enables cloud providers to limit active or suspended containers.

Moreover, the amount of memory also decides the amount of virtual CPU available to a Lambda function. That means the amount of virtual CPU will increase proportionally with more memory, increasing the overall computation power available. Therefore, changing the memory setting will directly impact the performance of the Lambda function.

At first, we ran the experiment at x86 architecture with 50 to 100 concurrent invocations of
each Lambda function repeating 50 times in cold and warm environments at static memory allocation of 2048 MB. We repeated the same experiment on ARM64 architecture and plotted the memory used versus the complexity of workloads shown in the Figure 5.12.

Figure 5.12. Effect on Memory Usage at cold and warm start call for multiple workloads

The memory consumption of any specific workload at any memory allocation is almost the same. Moreover, the memory usage for any particular workload at x86 and ARM64 architecture is nearly the same, and this result also stands for the warm start.

However, comparing the memory consumption among the workloads is quite different, as shown in the Figure 5.12. At both cold and warm start, we can see that increasing the size and complexity of workload increases the probability of memory used at both x86 and ARM64 architecture.

Furthermore, we plotted the results between memory usage and the complexity of workloads at each architecture to understand the nature of cold and warm starts during peak memory usage. During this experiment also, we obtained and plotted the results (Figure 5.13) at memory allocation of 2048 MB, where we have invoked 50 to 100 concurrent
invocations repeating each Lambda function 50 times.

Figure 5.13. Effect on Memory Usage at x86 and ARM64 architecture for multiple workloads

Increasing the code size and workload complexity increases the memory usage for that specific workload at both x86 and ARM64 architecture. In addition, a cold start consumes less memory than a warm start for any particular workload. However, there is a significant difference in memory consumption when we move from web application to utility application, which holds for both architectures.

5.1.5. Effect of Input Size

The input size is also one of the factors which affect perf-cost metrics. To observe its effect, we invoked 100 concurrent Lambda functions of workload Dynamic-HTML. There are three different input sizes (tiny, small, and large) for which we run the experiments and obtain the results. In the tiny, we have around ten inputs; in small, we have 1000 inputs; in large, we have 100000 inputs. We have processed the result at a cold start call at a static memory allocation of 512MB. The exact process is repeated for the same workload at ARM64 architecture.
Effect on Execution Time:

The plot (Figure 5.14) execution time versus input size showed that increasing the input size increases the execution time by a small number when we moved from test to small. However, the execution time significantly increases when we increase the input size from small to large. Even though this result is actual for both x86 and ARM64 architecture, the ARM64 is slightly performing better.

![Figure 5.14. Effect of Input Size on Execution Time](image-url)

Effect on Client Time:

In addition to Execution Time, we also observed the effect of input size on client time. The Figure 5.15 illustrates that the input size directly affects the client time, and we can see a similar impact on both x86 and ARM64 architecture. Here also, ARM64 performed better than x86 architecture.

Effect on Provider Time:

The provider time also increases in a similar pattern to execution time and client time due to an increase in input size (Figure), and the ARM64 performs better than the x86 architecture.
Effect on Memory Usage:

Finally, we observed the effect of input size on memory usage (Figure). We can see that increasing the input size increases the memory usage for both x86 and ARM64 architecture.
However, using ARM64 architecture during the cold start is advisable as it consumes less memory than x86 architecture. We can see a similar kind of effect on other workloads as well.

![Figure 5.17. Effect of Input Size on Memory Usage](image)

5.2. Latency

In general, latency means the time required to reach the destination from the source. During the latency analysis, we measured the metric connection time, which is the time between the benchmark driver and function instance. We observed the effect on connection time in different scenarios.

First, we analyzed the effect on connection time at each workload separately at varying static memory allocation for both cold and warm starts calls. During this experiment, we invoked 50 to 100 concurrent invocations of the Lambda function of the workload Dynamic-HTML at different static memory allocations ranging from 128MB to 3008MB for both x86 and ARM64 architecture separately. We separately obtained the results for cold and warm start
and plotted connection time versus memory sizes as shown in the Figure 5.18.

![Graph](image1)

(a) x86-vs-ARM64 at cold start for Dynamic-HTML (b) x86-vs-ARM64 at warm start for Dynamic-HTML

Figure 5.18. Effect on Connection Time at various memory sizes for x86 and ARM64 for Dynamic-HTML

From the Figure 5.18, at a cold start, x86 architecture comparatively takes more time to connect than ARM64 architecture at each memory allocation, and this result also holds for a warm environment. Therefore, ARM64 architecture, either at a cold or warm start, is a better option for executing a Lambda function. In addition, we have also run the same experiments with the same configuration settings separately for other workloads - uploader and compression and obtained similar results. Their plots are shown in Appendix C.3.

Moreover, we also observed the effect on execution time due to the code size and complexity to compare x86 architecture versus ARM64 in detail. We invoked each workload’s 50 to 100 concurrent Lambda functions at 128MB memory allocation. We obtained and processed the results and plotted the execution time versus the complexity of workloads at both cold and warm start separately(Figure 5.19).

The Figure 5.19 shows that increasing the code size and workload complexity increases
the connection time in a cold start environment for both architectures. However, unlike the x86, there is not much difference in connection time during the warm start call for ARM64 architecture. In addition, ARM64 needs less latency for any specific workload than x86 architecture, and these results stand in both cold and warm start calls.

Finally, we plotted the execution time versus workloads at each architecture to understand the behavior of cold and warm start calls on latency.

The Figure 5.20, at both x86 and ARM64 architecture separately, further proves that code size and complexity directly affect the connection time. In addition, a cold start generally takes longer than a warm start calls, irrespective of architecture.

5.3. Overhead

The overhead or cold startup overhead generally affects the performance. We estimated the cold startup overhead by summarizing the ratios of cold and warm execution time and provider time for each workload, i.e., $T_{cold}/T_{warm}$. The result obtained from our earlier experiments for the workload uploader, we computed cold startup overhead ($T_{cold}/T_{warm}$).
Figure 5.20. Effect on Connection Time at cold and warm start call for various workloads for x86 and ARM64 architecture separately. As shown in the Figure 5.21, we plotted overhead at various memory allocations for x86 and ARM64 architecture.

Figure 5.21. Execution time Cold startup overhead for uploader

The Figure 5.21 shows that for the uploader, the cold startup overhead of the execution time is almost constant at any particular memory allocation for both x86 and ARM64 arhitec-
ture. However, the overhead at ARM64 is lower than x86.

In addition, we observed the provider time’s cold startup overhead and plotted it against memory allocation, as shown in the Figure 5.22.

![Figure 5.22. Provider time Cold startup overhead for uploader](image)

The Figures 5.22 illustrate that the cold startup overhead of the provider time is also nearly constant for both x86 and ARM64 architecture at any memory allocation. In addition, ARM64 has a lower cold startup overhead than x86. We can see similar kinds of results for other workloads as well.

Furthermore, we have analyzed the cold startup overhead of execution time at multiple workloads whose code size and complexity are different (Figure 5.23).

The Figure 5.23 shows that ARM64 comparatively has better performance than x86 architecture. However, we can see that increasing the code size and complexity decreases the overhead of both architectures and the difference between them is also low. The reason behind this nature may be due to high memory invocation.
Figure 5.23. Execution time Cold startup overhead for multiple workloads

We know from the earlier analysis that increasing code size and complexity increases the high memory invocation or high memory usage. Therefore, high memory usage helps to reduce the cold startup overhead of execution time and provider time on Lambda function by providing more CPU allocation during the initialization and compilation.

5.4. Conclusion

This chapter explained the behavior of x86 and ARM64 architecture at different memory sizes at cold and warm start call for particular and multiple workloads during the perf-cost analysis. For the perf-cost study, we also observed the behavior of cold and warm startup at each architecture, the effect of code size and complexity, and the effect of input size on perf-cost metrics. In addition, we also studied the impact on latency at various memory allocations for each workload and finally showed the influence of cold startup overhead over performance. A summary of results obtained from our study and some future work are discussed in Chapter 6.
Chapter 6. Discussion and Future Work

This chapter discusses our study’s applicable contribution and possible future work. In brief, we have studied the following:

• Behavior of each AWS architecture at cold and warm start call of Lambda functions for particular and different workloads.

• Behavior of cold and warm startup at each AWS architecture.

• Effect of Complexity and size of code on memory usage at each AWS architecture.

• Effect of Input size on perf-cost metrics and memory usage.

• Behavior of complexity of code on latency at each AWS architecture.

• Effect of cold startup overhead on performance.

To study the perf-cost analysis in detail, we have divided it into multiple sections. From the first analysis study, we observed that the execution time, client time, and provider time are comparatively higher in x86 architecture at any static memory allocation ranging from 128MB to 3008MB during the cold start. And this is true for any specific workload at a cold start. However, we can observe the different results for some perf-cost metrics during the warm start call. For example, the execution time for the workload Dynamic-HTML and uploader is higher in ARM64 than x86 architecture.

Moreover, our second study showed the impact of workloads on perf-cost metrics at each AWS architecture. Here, the workloads are the representation of increasing code size and its complexity. We saw increasing the code size and complexity increase the execution time, client time, and provider time at each AWS architecture during cold and warm start invocations at a static memory allocation of 128MB. In addition, the ARM64 architecture performs better than x86 in both methods of Lambda function invocations, and this result
also holds for other memory allocations. However, we can observe that a warm start call takes comparatively less time than a cold one. Therefore, we did the third study to check each AWS architecture’s cold and warm start behavior.

During our third study of perf-cost, we observed and verified that increment of code size and complexity directly impact the execution time, client time, and provider time at both x86 and ARM64 architecture. Furthermore, we also concluded that Lambda functions’ warm invocations perform better than cold ones irrespective of the AWS architecture.

We extend our perf-cost analysis further to see the effect of size and complexity of workloads on memory usage. Here we observed the results at static memory allocation of 2048 MB. The plots illustrate that increasing the code size and complexity increases the memory usage at each AWS architecture during cold and warm start up. Although the ARM64 performs slightly better than x86, there is not much difference in memory consumption for any specific workload.

However, if we study each AWS architecture separately, we see that warm invocation consumes slightly high memory than cold startup for web applications - Dynamic-HTML and uploader. And we further observed that the memory consumption difference between warm and cold is highly increased when we move from web application to utility application workload - compression.

Our final analysis of perf-cost is about the effect of the input size or the number of inputs on perf-cost metrics. We analyzed our results for the workload - Dynamic-HTML at cold startup at a static memory allocation of 512MB. The plots explain that when we increase our input size, from test to small, there is only a slight increment in the execution time, client time, provider time, and memory usage for each AWS architecture. However, when
we increase the input size from small to large, every perf-cost metric significantly increases in both x86 and ARM64 architecture. More importantly, the increment in input size does not deteriorate the performance of ARM64 in comparison to x86 architecture.

Apart from perf-cost, we have also analyzed the latency at each workload. For any specific workload, either at cold startup or warm startup, the connection time of x86 architecture is higher than ARM64 at any static memory allocation. In addition, we also observed that increasing the code size and complexity increases the connection time for each workload at both x86 and ARM64 architecture. This result is valid for both cold and warm start invocations. Moreover, the warm invocation takes less connection time than the cold one, irrespective of the architecture.

Since overhead has some impact on performance, we finally analyzed the effect of cold startup overhead (T\text{cold}/T\text{warm}) on performance for specific and multiple workloads. We observed a negligible difference between x86 and ARM64 for client time overhead of the workload - uploader. However, there is a significant difference between x86 and ARM64 architecture for execution and provider time overhead. Furthermore, we also observed the plot between overhead and complexity of code, where overhead decreases with the increase of code size and complexity. And this is mainly due to high memory consumption at bigger complex workloads, which have more CPU during the initialization and compilation. In all the plots, we saw ARM64 has less cold startup overhead than x86 architecture.

This thesis has emphasized the importance of perf-cost metrics, latency, and overhead for better performance and lower cost of an application running in the AWS Cloud. Moreover, this thesis demonstrated the working of a diverse suite of FaaS benchmarks - the SeBS Framework in both x86 and ARM64 architecture. The flexibility provided by the SeBS
framework and the recently added new architecture -ARM64 by AWS, have motivated us to utilize and enhance the research work to unlock more factors that affect the performance of an application. This thesis explains the process that can be performed while evaluating serverless computing. The SeBS framework allows us to run different experiments for multiple workloads of varying sizes and complexity.

In conclusion, the evaluation of serverless computing helps the user or organization choose the most efficient configuration in the AWS cloud to run their specific workload. Although cloud engineers strive to reduce the issues related to cold and warm startups, there will always be welcoming new emerging challenges. A successful evaluation of serverless computing with a proper methodology can guarantee the performance and lower cost of an application running at any architecture in the different cloud environments.

Using this study as a base, we can enhance our research for other workloads whose complexity and size are heavier. In addition, it will also be wise to see the effect of invocation overhead on the performance of an application. Currently, AWS is only the cloud provider which supports the eviction model where the test function can be executed multiple times with varying size, memory, and runtime configuration to check how long the function instances can stay alive. So, in the future, we can use the result of this model to estimate the analytical model describing cold startups. Moreover, we can also see the effect of GPU architecture (currently not supported in AWS) on the performance of various workloads.
Appendix A. Getting access and secret keys from AWS

AWS offers a free tier for 12 months where it provides many services, including a sufficient amount of AWS Lambda’s computing time. Once an account is created, there are two ways to log in: Root user or IAM user. To work with AWS, we need access and secret keys with sufficient permission to invoke and manage Lambda functions and S3 resources. The figures A.1 and A.2 depict extracting such root user keys from the AWS account.

By default, a root user has all the services’ permission. So, no extra consent is needed to invoke the functions via AWS HTTP trigger. However, those secret keys must have
Figure A.2. Steps For Getting Keys

**AmazonAPIGatewayAdministrator** permission for IAM users to invoke the functions. In addition, a custom Lambda role with enough permission can also be provided to access AWS Lambda and S3. If not provided, one will be created automatically. In our experiments, we used root user access, secret keys, and a default Lambda role.
Appendix B. Requirements On Host (Linux System)

The downloaded access and secret keys or credentials must be passed as environment variables for the first run to access AWS services via a CLI tool like Putty. In addition, some requirements are needed to be installed on the host machine (Ubuntu) before running the experiments. The details of creating the environment variables on the host (Ubuntu System) and installing requirements are explained below.

B.1. Steps For Setting an AWS environment Variables

First, login to the system with a user that has `sudo` privilege. Then follow below steps;

1. Open the file `/etc/environment` using vi or vim editor. Eg. `sudo vim /etc/environment`

![Image of setting environment variables]

   Figure B.1. Setting Environment Variables on a Ubuntu Host

2. Append two variables, `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, with their corresponding value obtained from the AWS account.

3. Restart the system to bring the changes into effect. Eg. `sudo shutdown -r now`
B.2. Requirements Installation and Environment Activation

Before running the testing and experiments, we need Python version 3.7+ and update the system by running command `sudo apt-get update`. In addition, some requirements must be installed on the host machine. They are:

**Git:** `sudo apt-get install git`

**Python3-pip:** `sudo apt install python3-pip`

**virtual Environment:** `sudo apt-get install python3-venv`

**Docker:** `sudo snap install docker.io`

**Libcurl:** `sudo apt-get install libcurl4 libcurl4-openssl-dev -y`

**Libssl:** `sudo apt-get install libssl-dev`

Other essential things that need to be kept in mind are that the zip package needs to be already installed, and a `sudo` user must also be a member of the `docker` group. Also, ensure our Daemon is running and a user has enough permission to use it. To do that, run the command `sudo usermod -aG docker $USER`. Here replace the user variable accordingly. Once

![Figure B.2. An environment setup](image)

60
a user is added to the docker group, run the command *newgrp docker* to bring changes into effect or restart the system. Once rebooted, follow the below steps to create an environment and make the system ready for experiments.

1. Clone SeBS from the GitHub repository. *git clone https://github.com/spcl/serverless-benchmarks.git*

2. Go to the repository and install benchmarks to support the AWS platform. *./install.py –aws*

3. Activate a new virtual python environment. *python-venv/bin/activate.*
Appendix C. Additional Plots

C.1. Supporting Plots for Benchmark - uploader

Execution Time Plot:

(a) x86-vs-ARM64 at cold start for uploader

Figure C.1. Effect on Execution Time at various memory sizes for x86 and ARM64 for uploader

Client Time Plot:

(a) x86-vs-ARM64 at cold start for uploader

(b) x86-vs-ARM64 at warm start for uploader

Figure C.2. Effect on Client Time at various memory sizes for x86 and ARM64 for uploader
C.2. Supporting Plots for Benchmark - compression

Provider Time Plot:

(a) x86-vs-ARM64 at cold start for uploader
(b) x86-vs-ARM64 at warm start for uploader

Figure C.3. Effect on Provider Time at various memory sizes for x86 and ARM64 for uploader

Execution Time Plot:

(a) x86-vs-ARM64 at cold start for compression
(b) x86-vs-ARM64 at warm start for compression

Figure C.4. Effect on Execution Time at various memory sizes for x86 and ARM64 for compression
Figure C.5. Effect on Client Time at various memory sizes for x86 and ARM64 for compression

Figure C.6. Effect on Provider Time at various memory sizes for x86 and ARM64 for compression
C.3. Supporting Plots for Latency

Connection Time Plots for benchmark - compression

(a) x86-vs-ARM64 at cold start for compression  (b) x86-vs-ARM64 at warm start for compression

Figure C.7. Effect on Connection Time at various memory sizes for x86 and ARM64 for compression

Connection Time Plots for benchmark - uploader

(a) x86-vs-ARM64 at cold start for uploader  (b) x86-vs-ARM64 at warm start for uploader

Figure C.8. Effect on Connection Time at various memory sizes for x86 and ARM64 for uploader
Bibliography


Charitra Maharjan, a native of Lalitpur district, Nepal, was born on June 1st 1989 to Mr. Gyan Bahadur Maharjan and Mrs. Ratna Maya Maharjan. After finishing his schooling at Deepmala Secondary School in 2006, he graduated 12th from United Academy, Kumari-pati Lalitpur. He studied Electronics and Communication Engineering at Advanced College of Engineering and Management in Lalitpur from 2009 through 2013 toward obtaining his Bachelor of Engineering. After graduation, he finished his Master’s in Electrical Engineering at Louisiana State University, Baton Rouge, USA, in 2017. After getting three years of experience in the corporate world, he is pursuing his second Master’s program as a graduate student in Computer Science Department. During his time at LSU, he has been working as Teaching Assistant (TA) and a Graduate Assistant for BrBytes, a research practitioner partnership between East Baton Parish School System (EBRPSS) and Louisiana State University (LSU). Upon receiving his second Master’s degree this December 2022, he plans to work in the DevOpsSec field.