On Performance Tuning of Serverless IoT Applications

JAIME DANTAS

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING

YORK UNIVERSITY
TORONTO, ONTARIO, CANADA

APRIL 2022

© **JAIME DANTAS, 2022**

Abstract

Cloud computing has become a predominant IT operation platform in the past decade. Small and large companies have been migrating their workloads to the cloud, and serverless architectures, such as container and Function as a Service (FaaS), are among the popular choices for cluster software deployments. Within this context, autoscaling, the ability to dynamically adapt the cluster capacity based on the current demand is pivotal for maintaining Quality of Service (QoS) and optimizing the cost in the presence of workload fluctuations. The first contribution of this thesis is a novel autoscaling solution that uses burstable instances along with regular instances to handle the queueing arising in traffic and flash crowds. In a second contribution, we evaluate different types of deployments for FaaS, and present three recommendations that developers can consider when deploying their workloads on the public cloud. Finally, we present a resource-aware dynamic load balancer component for edge computing platforms using one of the most fast-growing IoT services in the industry. The contributions are tested and validated on public clouds.

Preface

The research presented in this thesis is the original work of Jaime Dantas, and it has been conducted in collaboration between the Performant and Available Computing Systems (PACS) Lab led by Dr. Hamzeh Khazaei, and the NSERC CREATE Program in Dependable Internet of Things Applications (DITA) Lab led by Dr. Marin Litoiu. This thesis is organized in paper format following the guidelines for paper-based theses. Parts of this report have been accepted to the peer-reviewed publication listed below.

- Dantas, J., Khazaei, H., & Litoiu, M. (2021). BIAS Autoscaler: Leveraging Burstable Instances for Cost-Effective Autoscaling on Cloud Systems. In Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021 (pp. 9-16). DOI: https://doi.org/10.1145/3493651.3493667

Parts of this report are still undergoing review by the following venues:

- Dantas, J., Khazaei, H., & Litoiu, M. (2022). Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda. In 2022 IEEE International Conference on Cloud Computing (CLOUD). (submitted)

- Dantas, J., Khazaei, H., & Litoiu, M. (2022). GreenLAC: Resource-Aware Dynamic Load Balancer for Serverless Edge Computing Platforms with AWS Greengrass. In 2022 IEEE International Conference on Edge Computing (EDGE). (submitted)

Acknowledgments

Firstly, I would like to express my deepest gratitude to my supervisors, Dr. Hamzeh Khazaei and Dr. Marin Litoiu, for providing support and guidance throughout my graduate studies. They have always been present and have provided valuable insights and feedback throughout my research. I could not have accomplished my objectives without their support.

Thanks to all my friends and colleagues in the Performant and Available Computing Systems (PACS) Lab who helped me throughout my studies. I would particularly like to thank Dr. Nima Mahmoudi and Changyuan Lin for their assistance and guidance with my research.

My sincere thanks also go to Dr. Manos Papagelis for his inspirational lectures and guidance.

Finally, I would like to thank my parents for their support and encouragement throughout my life.

Table of Contents

List of Tables

List of Figures

Abbreviations & Acronyms

API Application Programming Interface.

AWS Amazon Web Services.

CPU Central Processing Unit.

EC2 Amazon Elastic Compute Cloud.

ECR Amazon Elastic Container Registry.

ECS Amazon Elastic Container Service.

GCP Google Cloud Platform.

HTTP Hypertext Transfer Protocol.

IoT Internet of Things.

JPEG Joint Photographic Experts Group.

MQTT Message Queue Telemetry Transport.

OS Operating System.

Chapter 1

Introduction and Background

1.1 Cloud Computing

Over the past decade, cloud computing has been consolidated as the main architectural model for building applications and services. This rapid growth in adoption is a direct consequence of improvements regarding cost, scale, performance and security that cloud systems often offer. This is because cloud computing providers eliminate the initial capital investment companies would usually spend when building their own on-premise data centers. In addition, the ability to dynamically adapt the service capacity based on the current demand is pivotal for many businesses that may experience exponential growth on their service's demand. Autoscaling cloud resources can not only reduce the overall cost for their customers, but also preserve the quality of their services. Furthermore, public cloud providers often offer better performance and security compared to on-premise servers since they are regularly upgraded to the latest generation of fast and efficient computing hardware.

Cloud computing models can be classified into three categories: public, private, and hybrid. Large corporations usually have services on all three layers of cloud computing. Small companies, on the other hand, often rely on public cloud providers for deploying their services. However, hybrid clouds are the most popular model design, accounting for 42% of the deployments in 2020 [1].

Most cloud computing services fall into four broad categories: IaaS, PaaS, FaaS, and SaaS [2]. The first concept of cloud computing was introduced with the Infrastructure as a Service (IaaS) model. On this model, companies rent infrastructure, servers, virtual machines (VMs), and storage from a public cloud provider. Then, the Platform as a Service (PaaS) model was introduced right after the popularity of containers and microservices. In this new paradigm, developers were able to deploy web or mobile apps without worrying about managing the underlying infrastructure of servers or storage. Some services that were once part of the PaaS model evolved to create a new category of cloud computing named serverless computing. Function as a Service (FaaS) and serverless services are often referred to as technologies that enable programmers to focus only on building app functionality with almost no infrastructure managing required. Some of the most popular cloud services of each category are shown in figure 1.1.

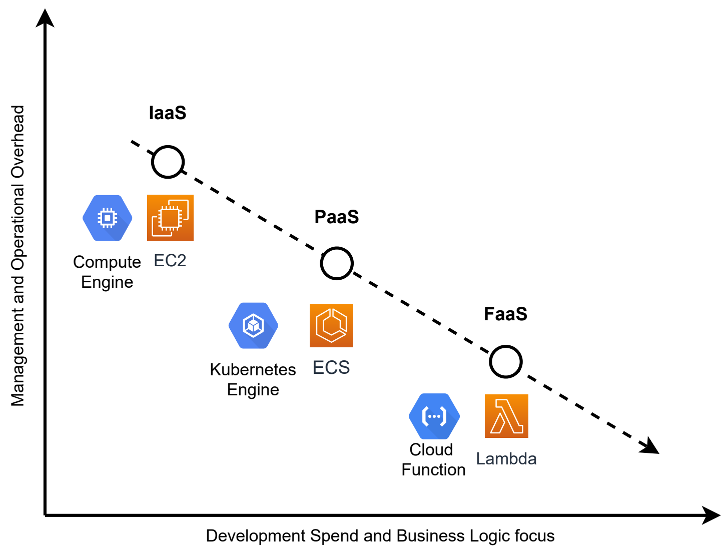
Management and Operational Overhead
IaaS
Compute Engine
EC2
PaaS
Kubernetes Engine
ECS
FaaS
Cloud Function
Lambda
Development Spend and Business Logic focus

Figure 1.1: Cloud computing.

1.2 Serverless Computing

Serverless computing, also known as FaaS, has gained a lot of popularity in the last five years. It allows developers to build applications faster by transferring the responsibility of managing the cloud infrastructure to the cloud provider. This means that the cloud provider is in charge of automatically allocating resources and managing the infrastructure required to run the code. By taking this approach, companies can focus on the core business logic while the cloud provider manages the scalability and security of their services. Cloud Functions are functions offered by Google Cloud where customers can deploy and run their code with zero server management [3]. Azure also provides their FaaS service named Azure Functions[4], and AWS has the AWS Lambda functions [5].

1.3 Edge Computing

Edge computing is a distributed computing paradigm that brings computational resources closer to the sources of data. This recently introduced model enables low latency applications to access computing and storage services deployed on-premises. Due to privacy and security reasons, companies may opt to process their data locally, and this growing necessity has driven tech companies to launch services dedicated to edge computing. One of the main concerns nowadays regarding distributed data across multiple nodes connected through the Internet is the risk of data leaks or cyber-attacks. Additionally, the increase of IoT devices at the edge of the cloud is producing a massive amount of data that needs to be processed and analyzed in real-time in most cases.

A recent survey conduction in 2022 has indicated that 83% of organizations have improved their efficiency by introducing IoT technology [6]. Although most of the major cloud providers offer some services at the edge, only AWS and Microsoft Azure allow their customers to execute serverless functions on the edge. With the AWS Greengrass and Azure IoT Edge, IoT applications and sensors can send their data to be processed locally at the edge using serverless functions. However, scaling serverless applications at the edge remains an open issue to this day. This is because both AWS Greengrass and Azure IoT Edge are not able to scale serverless services in the core dynamically. Figure 1.2 shows some of the edge and core services offered by AWS, Google Cloud and Microsoft Azure.

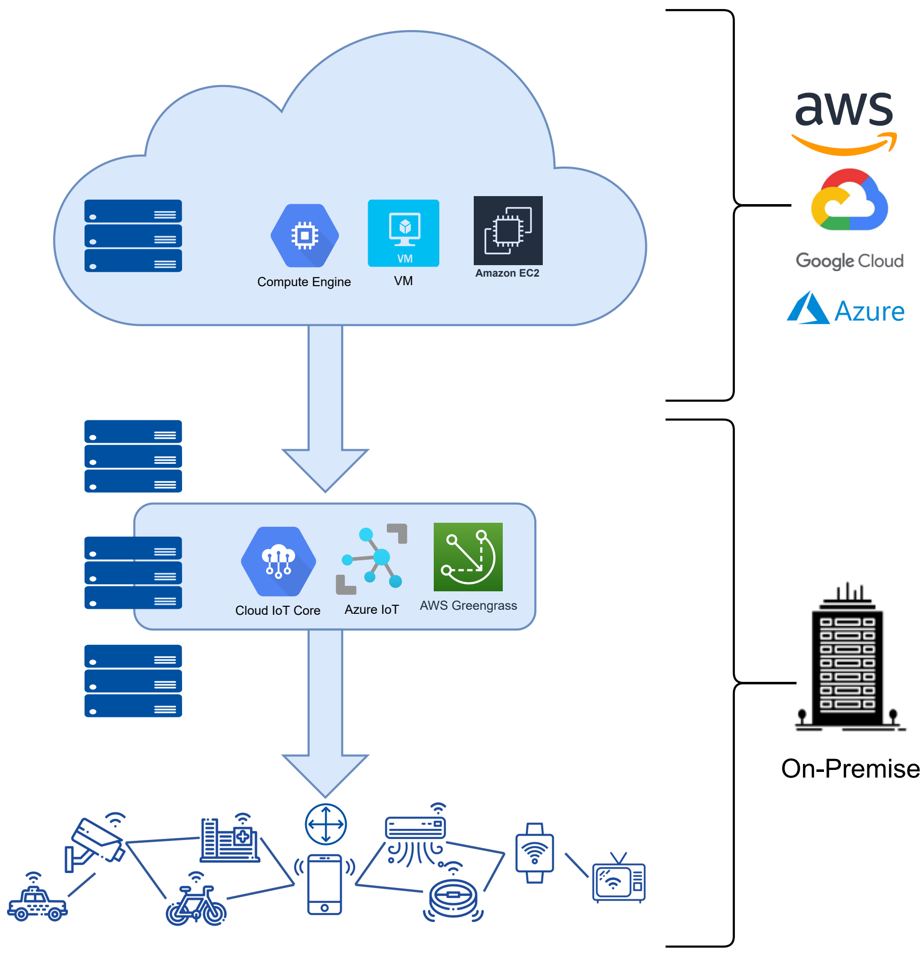

Figure 1.2: Edge computing.

1.4 Motivation and Objectives

Performance and cost optimization are pivotal for distributed systems. Serverless computing and edge computing are important architectures for the deployment of complex systems in the cloud. Hence, creating efficient scaling strategies for microservices, mitigating performance issues on serverless platforms and designing modern architectures for edge computing are of utmost importance for today's modern applications. Therefore, we address these challenges by proposing tools and an in-depth analysis of distributed systems. Specifically, in this thesis, we aim to:

- Create the first open-source autoscaling solution for Google Cloud Compute Engines that uses burstable instances and provide extensive documentation to allow developers to extend this work to other cloud services, especially Google Kubernetes Engine.

- Provide the first extensive analysis of AWS Lambda that takes into account the *ZIP* and *container-based deployment*, the language runtime, the memory and the package size of the function.

- Build the first AWS Greengrass component that performs dynamic load balancing of serverless applications across the core and edge nodes.

1.5 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 presents the design and experimental evaluation of BIAS Autoscaler, which is an autoscaling for cloud systems that leverages burstable instances. In Chapter 3, we propose guidelines to application developers for reducing the cold start delay of serverless functions. In Chapter 4, we present the architecture and performance evaluation of our dynamic load balancer for serverless edge computing platforms, GreenLAC. Finally, Chapter 5 lists our contributions and elaborates on the possible future directions for this research.

Chapter 2

BIAS Autoscaler: Leveraging Burstable Instances for Cost-Effective Autoscaling on Cloud Systems

Burstable instances have recently been introduced by cloud providers as a cost-efficient alternative to customers that do not require powerful machines for running their workloads. Unlike conventional instances, the CPU capacity of burstable instances is rate limited, but they can be boosted to their full capacity for small periods when needed. Currently, the majority of cloud providers offer this option as a cheaper solution for their clients. However, little research has been done on the practical usage of these CPU-limited instances. In this chapter, we present a novel autoscaling solution that uses burstable instances along with regular instances to handle the queueing arising in traffic and flash crowds. We design BIAS Autoscaler, a state-of-the-art framework that leverages burstable and regular instances for cost-efficient autoscaling and evaluate it on the Google Cloud Platform. We apply our framework to a real-world microservice workload, and conduct extensive experimental evaluations using Google Compute Engines. Experimental results show that BIAS Autoscaler can reduce the overall cost up to 25% and increase resource efficiency by 42% while maintaining the same service quality observed when using conventional instances only.

2.1 Introduction

Elasticity is one of the most important concepts of cloud computing. The ability to dynamically adapt the cluster capacity based on the current demand is pivotal for maintaining Quality of Service (QoS) and optimizing the cost. Autoscaling the resources can not only reduce the overall cost for the customer, but also preserve the Service-level Agreements (SLAs) and Service-level Objectives (SLOs) of the services. This is because most of the workloads face unpredicted variations in traffic during their usage. Sometimes, these spikes in usage can cause interference in the QoS metrics, leading to a negative impact on both the cloud providers and the customers.

Many autoscaling solutions try to mitigate these problems by provisioning additional computational resources to handle unpredicted spikes on their workloads. This is commonly known as overprovisioning the number of resources above the minimum required to handle sudden variation in traffic. The downside of this approach, however, is because this extra capacity is often not entirely used during normal demand, which in turn, leads to waste of resources and consequently, increasing the cost. To avoid wasting computational resources, cloud providers such as Amazon Web Service (AWS), Google Cloud Platform (GCP) and Microsoft Azure introduced burstable instances. We believe that this type of instance is the key to designing cost-efficient solutions on the public cloud, especially for small clusters.

Burstable instances are virtual machines whose CPU capacity is limited to a predefined threshold. Even though they are meant to operate under their operational CPU threshold, the CPU capacity can be boosted to the full standard capacity for small periods. Each cloud provider implements its own proprietary system to manage these instances, and they are usually based on tokens for CPU credits. AWS, for example, controls the frequency by which their burstable instances can operate above the CPU threshold by using a token-based system. Each minute the instance operates below its CPU threshold, the customer receives a token that can then be spent to boost these instances to CPU values above its threshold for one minute. Similarly, Microsoft Azure implements a token-like control system for their burstable virtual machines.

When we compare the cost of burstable instances with regular ones, we can see huge differences in the overall savings one could have. For instance, the Google Compute Engine *N1 shared-core g1-small* instance (with 1 vCPU at 50% sustained rate and 1.7 GB of memory) cost 52% less than the *N1 standard 1* instance (with 1 vCPU with 3.75 GB of memory). This difference can be as high as 10 times depending on the type of instance and the cloud provider.

We introduce the Burstable Instance Autoscaler (BIAS), an application autoscaler that combines different instance types for scaling virtual machines in the public cloud. BIAS Autoscaler was evaluated on GCP, and it uses the already existing infrastructure and services of GCP to add and remove resources as needed as well as distribute the traffic among the different types of instances. We use the Square-Root Staffing Rule to calculate the number of required servers on the fly, and evaluate our framework on a microservice workload. Our work presents two main contributions:

- We use a well-known technique to create a new solution for reducing the cost and increasing the efficiency of autoscaling systems by leveraging burstable instances in combination with conventional ones. We implement and evaluate our solution in our prototype BIAS Autoscaler, which is open-source on GitHub[1], and validate our technique under two distinct scenarios: transient queueing arising in traffic, and flash crowds.

- We demonstrate how our framework can be extended to other cloud providers, and how it can be used to manage other serverless services based on containers such as Kubernetes. We also explain how to implement customized scaling policies on BIAS Autoscaler.

[1]https://github.com/BIAS-Cloud/BIAS-Autoscaler

2.2 Background

In this section, we present the main features of burstable instances and GCP, and discuss how we validate our proposed framework, BIAS Autoscaler.

2.2.1 Burstable Instances

Burstable instances are machine types that limit their CPU utilization at a fixed rate. Most cloud providers such as AWS [7], GCP [8] and Microsoft Azure [9] offer this type of instance as an affordable option for workloads that usually do not require high computational power. They usually offer these instances as a solution to maximize the utilization of their idle resources. On GCP, for example, these instances are implemented as shared-core virtual machines that use context-switching for sharing a physical core between other vCPUs [8].

Studies have shown that for most cloud providers, underutilization of resources is a present issue for most of their services. For Microsoft Azure, approximately 60% of all virtual machines utilize less than 20% of its full capacity [10]. Similarly, the CPU utilization of Google Compute Engine instances is as low as 35% for most of the time [11]. To address this issue, we propose BIAS Autoscaler to maximize resource efficiency using burstable instances.

AWS offers several types of EC2 burstable instances under the families *T2, T3, T3a, and T4g* [7]. On this cloud provider, for on-demand EC2 instances, the cost of burstable instances varies from 90% to up to 10 times less than regular instances [12]. This figure is similar for GCP as well, reaching up to 8 times in savings for some burstable instances [13]. Unlike AWS, though, GCP offers only two families of burstable instances: *E2 shared-core* and *N1 shared-core* [8].

Table 2.1 shows a comparison of the savings that customers could have if they choose burstable instances for their services in the three major cloud providers. For this study, only instances with 1 and 2 vCPUs with memory matches were considered.

Table 2.1: Comparison of regular and burstable instances in terms of cost saving.

Cloud	**Burstable**	**Regular**	**vCPUs**	**Thld. (%)**	**Savings (%)**
Azure	B1MS	A1 v2	1	20	43
Azure	B2S	A2 v2	2	40	45
AWS	t2.small	a1.medium	1	20	10
AWS	t4g.medium	a1.large	2	20	34
GCP	g1-small	n1-standard-1	1	50	52
GCP	e2-medium	e2-standard-2	2	50	50

Even though AWS advertises savings up 10 times, when we compare the cost of burstable and regular instances with 2 vCPUs and 4 GB of memory, the savings drops to only 10%. GCP, on the other hand, offers a highest cost-benefit ratio for this configuration, with 50% in savings.

2.2.2 Google Cloud Platform

Google Cloud Platform (GCP) is the only public cloud provider among the major players that offer an application and network load balancer highly tunable, making it possible to distribute traffic based on the CPU utilization of instance groups [14]. We exploit this unique feature to control the traffic distribution among the different instance types. Our framework, BIAS Autoscaler, is the first of its kind to rely entirely on the existing cloud load balancer to control the traffic. The existing solutions that are similar to our framework BurScale [15] and CEDULE [16], which are both applied on AWS, require a customized load balancer implementation to work. This customized load balancer often seen on autoscaler implementations can not only add additional complexity to the cluster infrastructure, but also increase the probability of interference in the QoS metrics of the service.

Even though BIAS Autoscaler is applied and tested on Google Compute Engines [17], we explain how it can also be used to manage container-based services such as Google Kubernetes Engine [18] with small changes on its controller module. In addition, we demonstrate how it can be adapted to other public cloud providers such as AWS by using a customized load balancer.

2.2.3 Workload

In order to evaluate the performance of our framework, we created the Load Microservice and made it open-source on GitHub[2]. This microservice simulates a web-server application with a RESTFul API with adjustable CPU load and processing time. It is written in Kotlin, and it uses the Micronaut Framework[3], which is a full-stack framework with fast startup time. This microservice has an HTTP GET endpoint that retrieves the data from the resource ID sent by the client in the JavaScript Object Notation (JSON) format. We set this API to simulate a heavy CPU load for a few milliseconds on each call. The full documentation of the Load Microservice is available on GitPages[4].

2.2.4 Benchmark Test

We use the Locust[5] benchmark tool for our performance tests. Locust is an open-source, scriptable and scalable performance testing tool that allows customized use test cases written in Python.

2.3 BIAS Autoscaler Design

In this section, we present the architecture and design of BIAS Autoscaler, and show how it uses a combination of regular and burstable instances to reduce cost on the public cloud. We evaluate BIAS Autoscaler on GCP, and the source code is openly accessible on GitHub[1].

[2]https://github.com/BIAS-Cloud/Load-Microservice
[3]https://micronaut.io
[4]https://bias-cloud.github.io/Load-Microservice
[5]https://locust.io

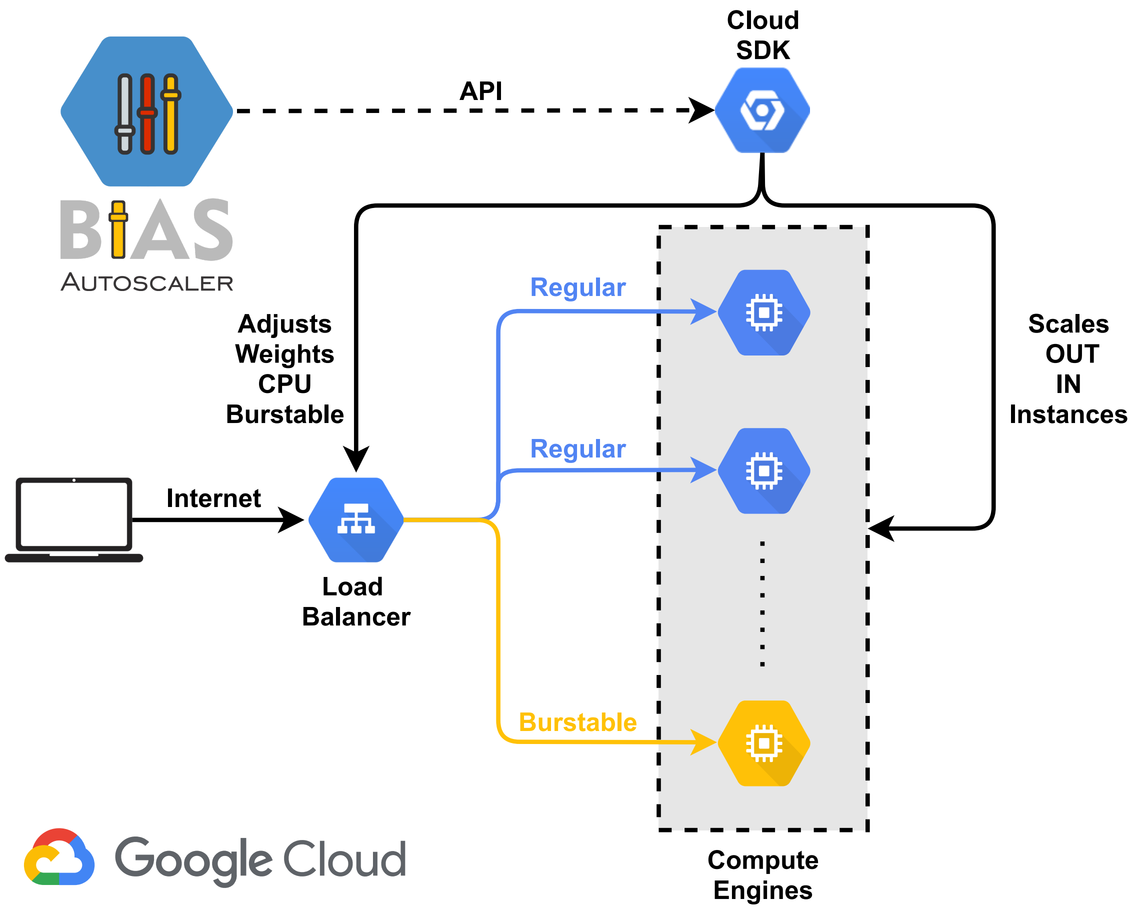

Figure 2.1: Cluster architecture managed by BIAS Autoscaler on GCP.

2.3.1 The Architecture of BIAS Autoscaler

BIAS Autoscaler is a ready-to-use autoscaler with little to no configuration required for cloud systems. Even though it was primarily evaluated and developed on GCP, it can be extended to other public cloud providers such as AWS and Azure. To the best of our knowledge, BIAS Autoscaler is the first open-source autoscaler fully tested and validated on GCP that leverages burstable instances for scaling Google Compute Engine instances. BIAS Autoscaler is also the first autoscaler to use the Google Load Balancer to dynamically change the traffic distribution among the instances. It uses the existing GCP services to manage and monitor the cluster, and it was developed using the Java programming language in combination with the Micronaut Framework.

A full step-by-step guide and documentation is provided on GitPages[6]. We used the Google Cloud Java API and SDK for scaling and controlling the cluster, and the Google Cloud Stackdriver Monitoring Client for monitoring the necessary metrics. BIAS Autoscaler can be deployed either on a Google Compute Engine instance or run as a container on Google Kubernetes Engine. It is a reactive autoscaler that uses the Google Load Balancer to adjust the CPU utilization of the burstable instances based on traffic distribution. Figure 2.1 shows how BIAS Autoscaler is used to scale out/in the resources on GCP. Its architecture is divided into three distinct modules: monitor, scaling and controller. The internal architecture of BIAS Autoscaler is shown in figure 2.2.

CPU Utilization of Burstable Instances

BIAS Autoscaler works by maxing out the CPU utilization of the burstable instance only when the cluster is scaling out new resources. It sets the CPU threshold of the burstable instances to its default value (T) as soon as the new resources are added and are ready to be used in the cluster. This CPU threshold is refereed as weights (w_b) on BIAS Autoscaler. By doing this, the full capacity of the burstable instances is used when the cluster requires additional computational power to process the current demand. On this strategy, BIAS Autoscaler will boost the burstable instances only when necessary.

Monitoring: The monitoring component is in charge of acquiring metrics from the load balancer and the instances. Since we validated BIAS Autoscaler on GCP, the Google Load Balancer and the Google Compute Engines were used. It fetches metrics from these cloud services through the Google Cloud Monitoring service.

Scaling: This component is where the scaling algorithm is implemented. It reads the metrics provided by the monitor component, and calculates the number of burstable and regular instances of the current demand. This information is then fed

[6]https://bias-cloud.github.io/BIAS-Autoscaler

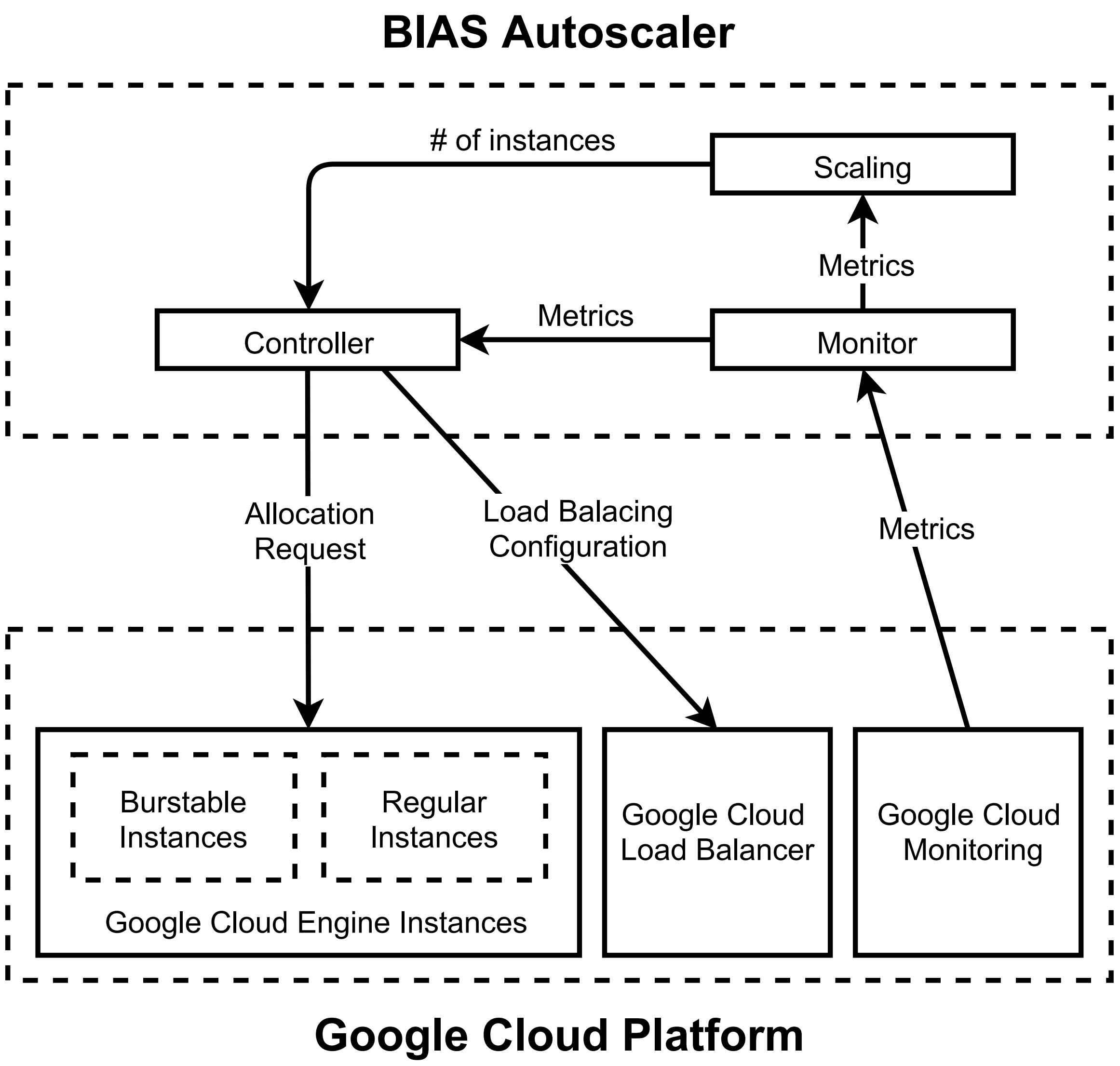

Figure 2.2: BIAS Autoscaler architecture.

to the controller module so it can perform the scaling of the cluster. Currently, BIAS Autoscaler supports only the Square-Root Staffing Rule (SR Rule) scaling policy, but any policy can be applied.

Controller: This module is the core of BIAS Autoscaler. The number of calculated burstable (k_{b_c}) and regular (k_{r_c}) instances is provided to this component, and it outputs the necessary changes to the cluster. Once again, since we validated BIAS Autoscaler on GCP, it uses the Google Cloud Java API to control the load balancer traffic distribution among the instance groups, and scales out/in the Google Compute

Engine instances. The controller module is also responsible for updating the weights of the CPU threshold (w_b) of the burstable instances. Whenever it scales out the regular or burstable instances, it sets w_b to 100% to burst the burstable instances to their maximum capacity while the new resources are being provisioned in the cluster. This helps to reduce the CPU load of the regular instances while the new resources are added to the cluster. As soon as the calculated number of instances (k_{b_c}, k_{r_c}) are identical to the current number of instances (k_b, k_r), BIAS Autoscaler sets w_b to its original threshold value, T.

Although BIAS Autoscaler was primarily designed and validated on GCP, it can be extended to AWS and Azure as well. In order to control EC2 instances on AWS, though, a customized load balancer is required since the AWS Elastic Load Balancer does not support dynamic adjustments in the traffic distribution among different instance groups. The same approach should be applied when using BIAS Autoscaler to control Azure Virtual Machines on Microsoft Azure. A generic interface is provided so users can implement a class to communicate with their customized load balanced using RESTful/gRCP APIs. Additionally, BIAS Autoscaler can be extended to manage services based on containers on GCP and other cloud providers as well. For GCP, a generic interface is provided to implement procedures to control the Google Kubernetes Engine using the Google Cloud SDK.

2.3.2 Scaling Policy

Both predictive and reactive scaling algorithms can be applied on BIAS Autoscaler. However, we chose a reactive approach to scale our resources. Our reactive strategy assumes that the future demand resembles the current state. We use the well-accepted Square-Root Staffing Rule (SR Rule) as our scaling strategy for BIAS Autoscaler. Many works [15], [19], [20] have been developed around autoscaling cloud resources based on the SR Rule in recent years. Since the Google Cloud Load Balancer allows the distribution of the traffic based on the CPU utilization of instance groups, we leverage this feature to control the utilization level of the burstable instances. Based on the previous development done on [21], we consider our system as an M/M/k queueing system.

Theorem 1 *(Square-Root Staffing Rule [22]) Given an M/M/k queueing system with arrival rate λ and service rate μ, and $R = \frac{\lambda}{\mu}$, where R is large, let k^*_α denote the least number of servers needed to ensure that the probability of queueing $P_Q^{M/M/k} < \alpha$. Then $k^*_\alpha \approx R + c\sqrt{R}$ where c is the solution for the equation $\frac{c\Phi(c)}{\phi(c)} = \frac{1-\alpha}{\alpha}$ where $\Phi(.)$ denotes the c.d.f. of the standard Normal distribution and $\phi(.)$ denotes its p.d.f.*

The parameter c is related to the probability of queueing, α, which determines the mean response time of our service, $E[T]$. Equation 2.1 shows how we can calculate $E[T]$ using P_Q where $\rho = \frac{\lambda}{k\mu}$ is the system utilization [22]. We assumed the probability of queueing (P_Q) of our service as 10% for all benchmark tests we performed, but this property can be easily changed on BIAS Autoscaler configuration file.

$$E[T] = \frac{1}{\lambda} \cdot P_Q \cdot \frac{\rho}{1-\rho} + \frac{1}{\mu} \tag{2.1}$$

Therefore, the SR Rule used to determine the number of servers k required to handle an arrival rate λ is $k_c = R + c\sqrt{R}$. Note that the value of R is known, and it varies depending on the cluster configuration and workload used. We use the approach proposed on [15] to determine the number of and regular ($k_{r\_c} = R$) and burstable ($k_{b\_c} = c\sqrt{R}$) instances.

2.4 Evaluation

We conducted two different experiments to evaluate the performance of BIAS Autoscaler for scaling virtual machines using Compute Engines on GCP. In order to evaluate the performance of our framework, we created the Load Microservice and made it open-source on GitHub[7]. This microservice simulates a web-server application with a RESTFul API with adjustable CPU load and processing time. The full

[7]https://github.com/BIAS-Cloud/Load-Microservice

documentation of the Load Microservice is available on GitPages[8]. We use the Locust[9] benchmark tool for our performance tests. Locust is an open-source, scriptable and scalable performance testing tool that allows customized use test cases written in Python. We analyzed the performance of our autoscaler during a transient queueing in traffic and a flash crowd scenario. For the former case, we evaluated BIAS Autoscaler in three distinct configurations, and then compared our performance to the rule-based GCP autoscaler. Finally, we tested BIAS Autoscaler for handling flash crowds and analyzed the QoS metrics and the SLOs violations for each test. All the experiments are openly accessible on GitHub[10].

2.4.1 Transient Queueing

For this experiment, we simulated a fixed increasing rate in traffic for a long period, and we analyzed the QoS metrics and SLOs violations when running BIAS Autoscaler with both burstable and regular instances compared to regular instances only and burstable instances only. In addition to evaluating QoS metrics and SLOs violations, we compare the computational power and efficiency of the burstable instances with their equivalent regular ones.

Experimental Setup: We created a cluster on GCP with burstable and regular on-demand instances. We used *N1 shared-core g1-small* instances as our burstable instances, and *N1 standard 1* as our regular ones. Both these instance types have 1 identical vCPU (Intel(R) Xeon(R) CPU @ 2.30GHz). The main difference between them is that the burstable instances are CPU limited to 50% utilization, but they can boost themselves up to 100% of 1 vCPU for small periods. For memory, however, our instances differ a bit. Whereas our regular instances have 3.75 GB RAM, our burstable ones have only 1.7 GB RAM. The CPU utilization target of the burstable instances was set to 40%. We first perform three benchmark tests: one with BIAS

[8]https://bias-cloud.github.io/Load-Microservice
[9]https://locust.io
[10]https://github.com/BIAS-Cloud/Experiments

Autoscaler scaling regular instances along with burstable ones, one scaling regular instances only where $k_r = R + c\sqrt{R}$, and another one scaling burstable instances only where $k_b = R + c\sqrt{R}$. We then compare the results of these three tests with another performance test using the GCP autoscaler with regular instances only set to scale out each time the CPU utilization reaches 50%.

SLOs: The SLO for the average response time was set to 150 ms with 95% of the requests below 300 ms, and no error is allowed.

Service rate μ: We run a benchmark test to determine the service rate experimentally. This test consisted of running a regular instance for 60 minutes under a fixed arrival rate. The service rate for our evaluation tests was set to $\mu = 17$ requests/s for each regular instance. The probability of queueing P_Q for all tests we performed is 10%.

Load generation: For simulating the user traces, we created a test scenario on Locust where the arrival rate λ increases linearly from 10 to 75 request/s in a window of 108 minutes.

Results: We reduced the cost by 25% when replacing some conventional instances with burstable ones. To achieve this, we compared the cost of running BIAS Autoscaler with regular and burstable instances (figure 2.3) against running it with regular instances only (figure 2.4). The scaling algorithm used in both tests was the SR Rule, and we considered $c\sqrt{R}$ as the number of burstable instances. This scaling strategy differs from a pure reactive scaling algorithm since more than one instance can be added at once under the SR Rule as can be seen on figure 2.5(c) and figure 2.6(c). Another surprising finding was the outstanding performance reached when running BIAS Autoscaler with burstable instance only. This is because this configuration resulted in savings of 56% compared with regular instances only with almost no impact in the SLOs. However, this high saving in cost may be questionable since all burstable instances ran above their CPU threshold of 50% during the entire test as can be seen in figure 2.6(d). Therefore, since the burstable instances on GCP are

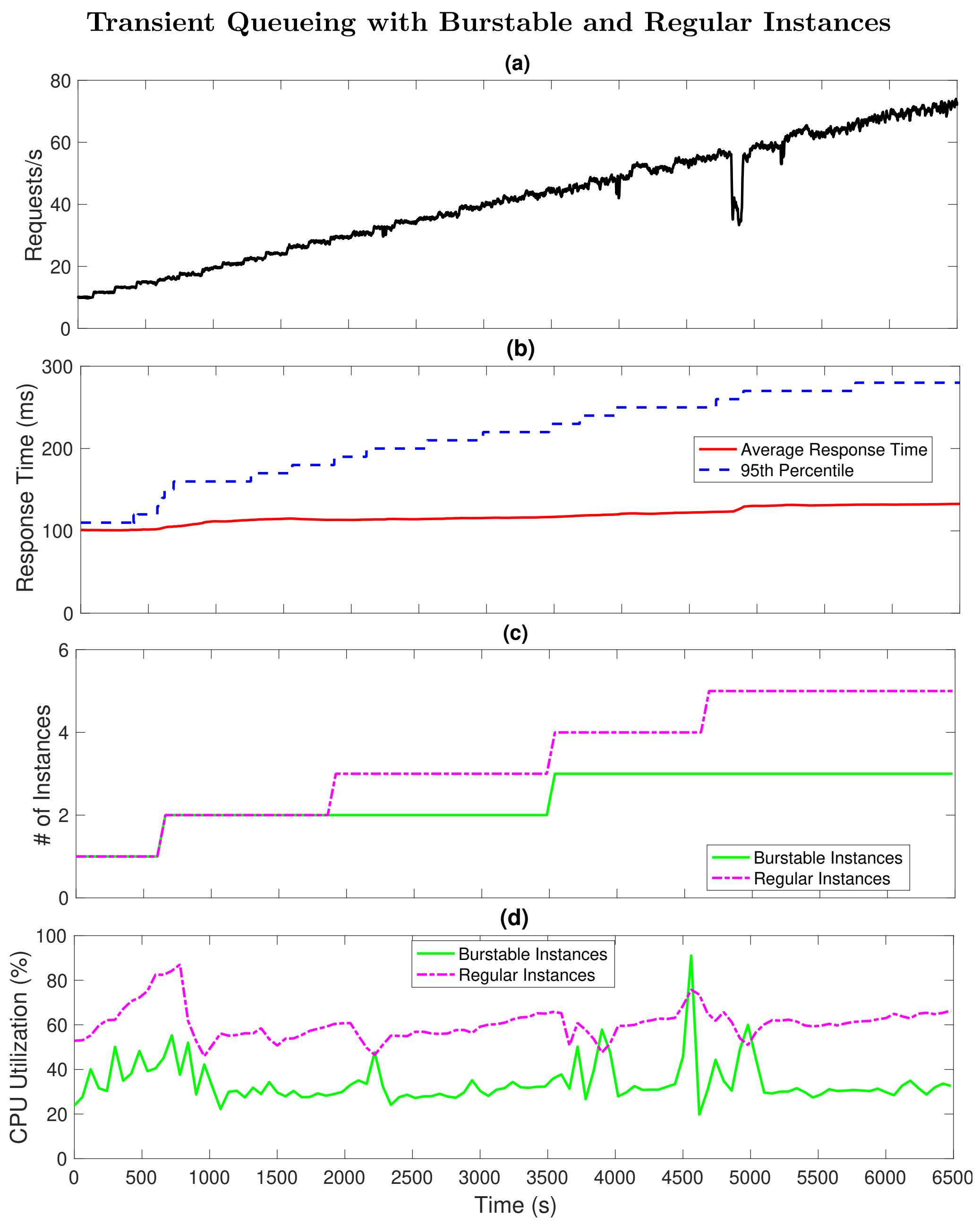

Figure 2.3: Results for the transient queueing experiment with burstable and regular instances.

highly workload dependable, they may not sustain long periods running at full CPU capacity.

This 25% cost savings can be understood better when we analyze the resource utilization during the two test scenarios. While we maintained an average CPU usage of our resources of approximately 45% when running with regular instances only, this figure was roughly 64% when we combined burstable and regular ones (considering we rate the CPU of the burstable instances at 50%). As a result, we increased our resource efficiency by 42% when using a combination of these two instance types. This demonstrates how BIAS Autoscaler can be used to not only reduce the cost, but also to increase the overall resource efficiency. Although relying solely on burstable instances appears to be the best cost-effective option at first glance, the black-box managing system of GCP states that there is no guarantee it can sustain long periods running on maximum CPU capacity. Thus, we do not advocate that cluster administrators should replace all their conventional instances with burstable ones instead. However, this demonstrates that burstable instances can indeed be used for replacing some regular instances as long as their CPU load is correctly managed by the autoscaler to avoid long runs at their maximum CPU capacity, as demonstrated in figure 2.3(d).

Even though we observed a slightly better average response time (7% only) when using regular instances only, the SLOs were not impacted when using burstable instances. The 95th percentile performance was also approximately equivalent for the two tests. The reason for that is because the maximum 95th percentile response time reached when running burstable and regular instances combined was less than 25% higher than running BIAS Autoscaler with regular instances only. Despite this small difference, both tests met the required SLOs for the 95th percentile.

Table 2.2 compiles the results of all four tests performed. Note that when we compare side-to-side BIAS Autoscaler with the rule-based GCP autoscaler, we can see that BIAS Autoscaler reduces the cost by approximately 18% while maintaining roughly the same SLOs.

2.4.2 Flash Crowd

For this experiment, we simulated a flash crowd for a short period, and we analyzed the QoS metrics and SLOs violations when running BIAS Autoscaler with both burstable and regular instances compared to regular instances only using the same

Table 2.2: Results of performance tests for transient queueing.

Test Scenario	**Average Response Time (ms)**	**Maximum 95th Percentile (ms)**	**Cost (10^{-3} USD)**
Regular instances only	110	210	493
Rule-based GCP autoscaler	108	220	450
Burstable and regular instances	118	280	371
Burstable instances only	120	220	218

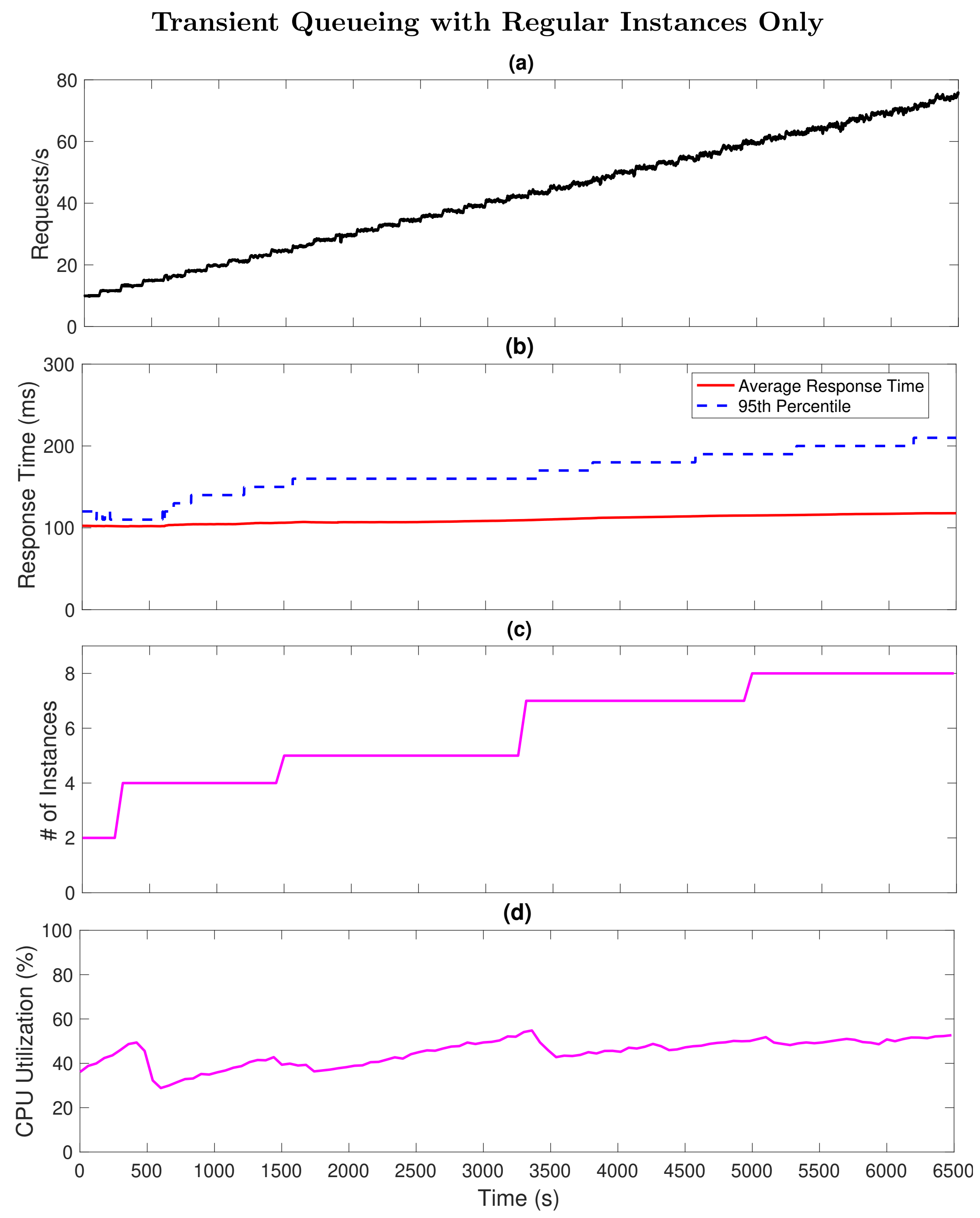

Figure 2.4: Results for the transient queueing experiment with regular instances only.

configuration as for the transient queueing experiment. We performed a benchmark test where BIAS Autoscaler runs with regular and burstable instances combined. The key difference of this experiment is the workload used. The SLO for the average response time was set to 300 ms with 95% of the requests below 1000 ms, and no error is allowed.

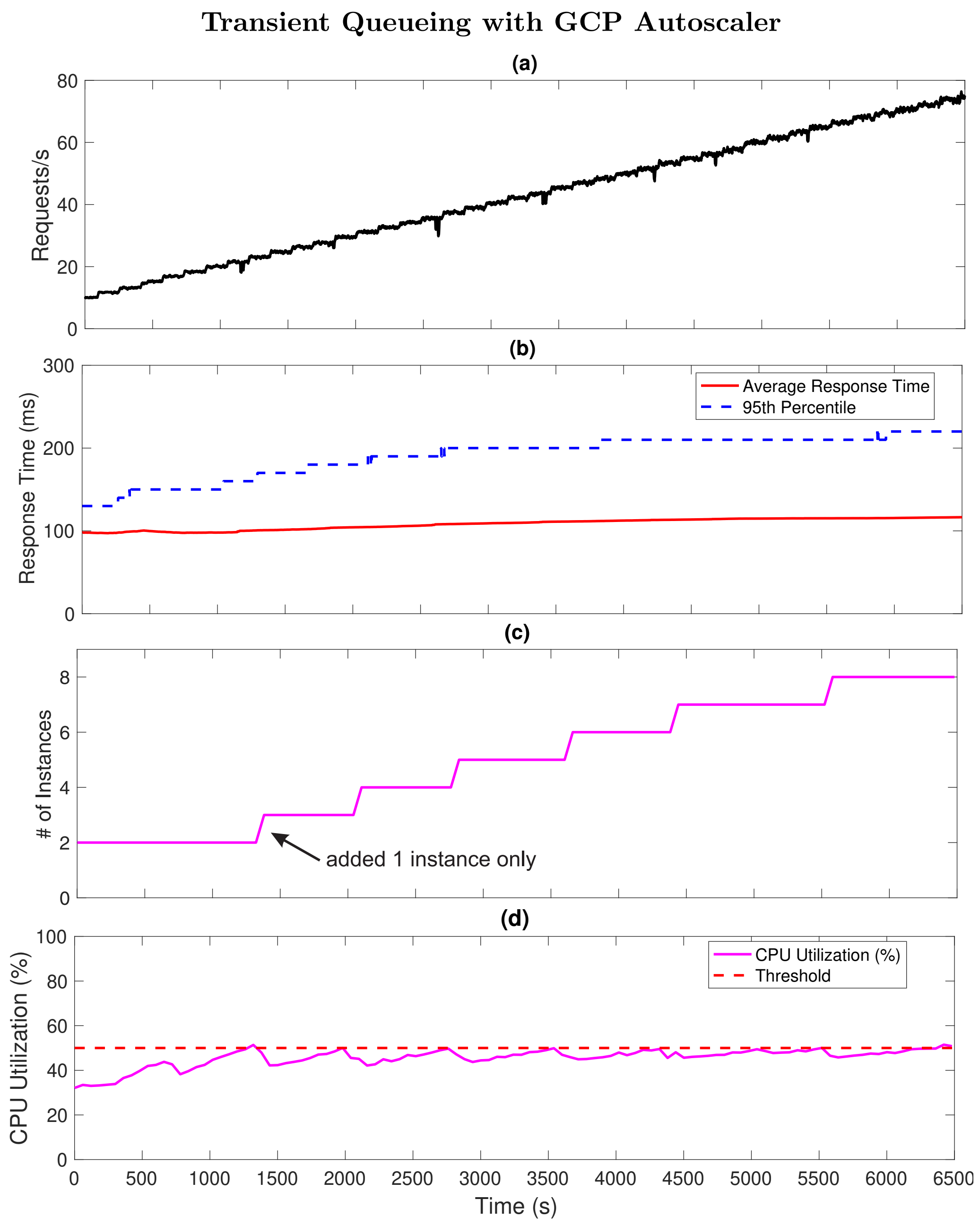

Figure 2.5: Results for the transient queueing experiment with the rule-based GCP autoscaler set to 50% using regular instances only.

Load generation: For simulating a flash crowd, we created a test scenario on Locust where the arrival rate λ increases from 10 request/s to three different picks with a maximum of 85 requests/s in a window of 33 minutes in total. We run this load against two distinct configurations of BIAS Autoscaler. The figure 2.7(a) shows the load used for the flash crowd experiment.

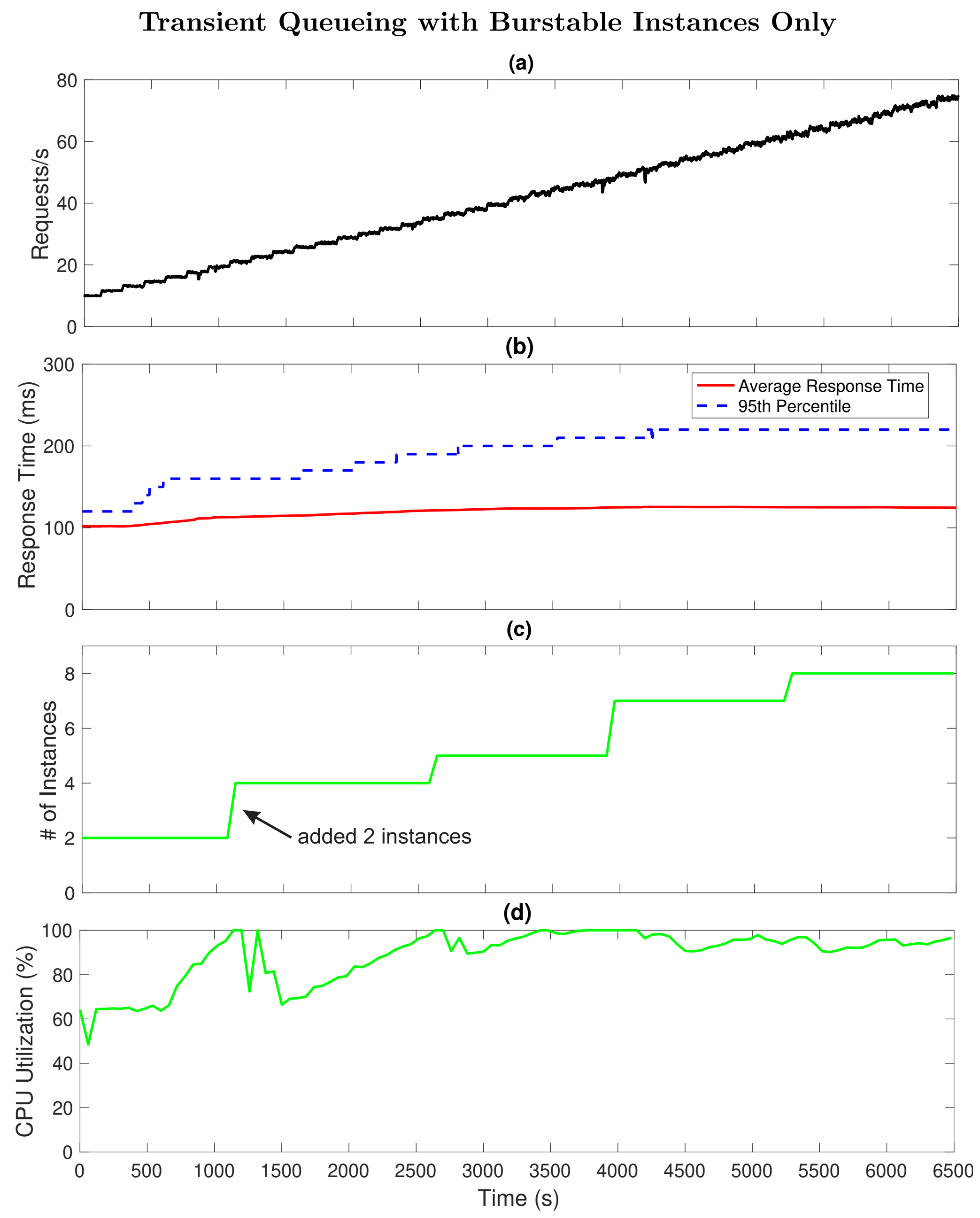

Figure 2.6: Results for the transient queueing experiment with burstable instances only.

Results: The outcome of this experiment was similar to the transient queueing benchmark test, with approximately 25% reduction in cost when replacing some conventional instances with burstable ones. Even though both performance tests where BIAS Autoscaler ran with burstable and regular instances and the one with regular instances only met all requited SLOs, the average response time achieved by the later

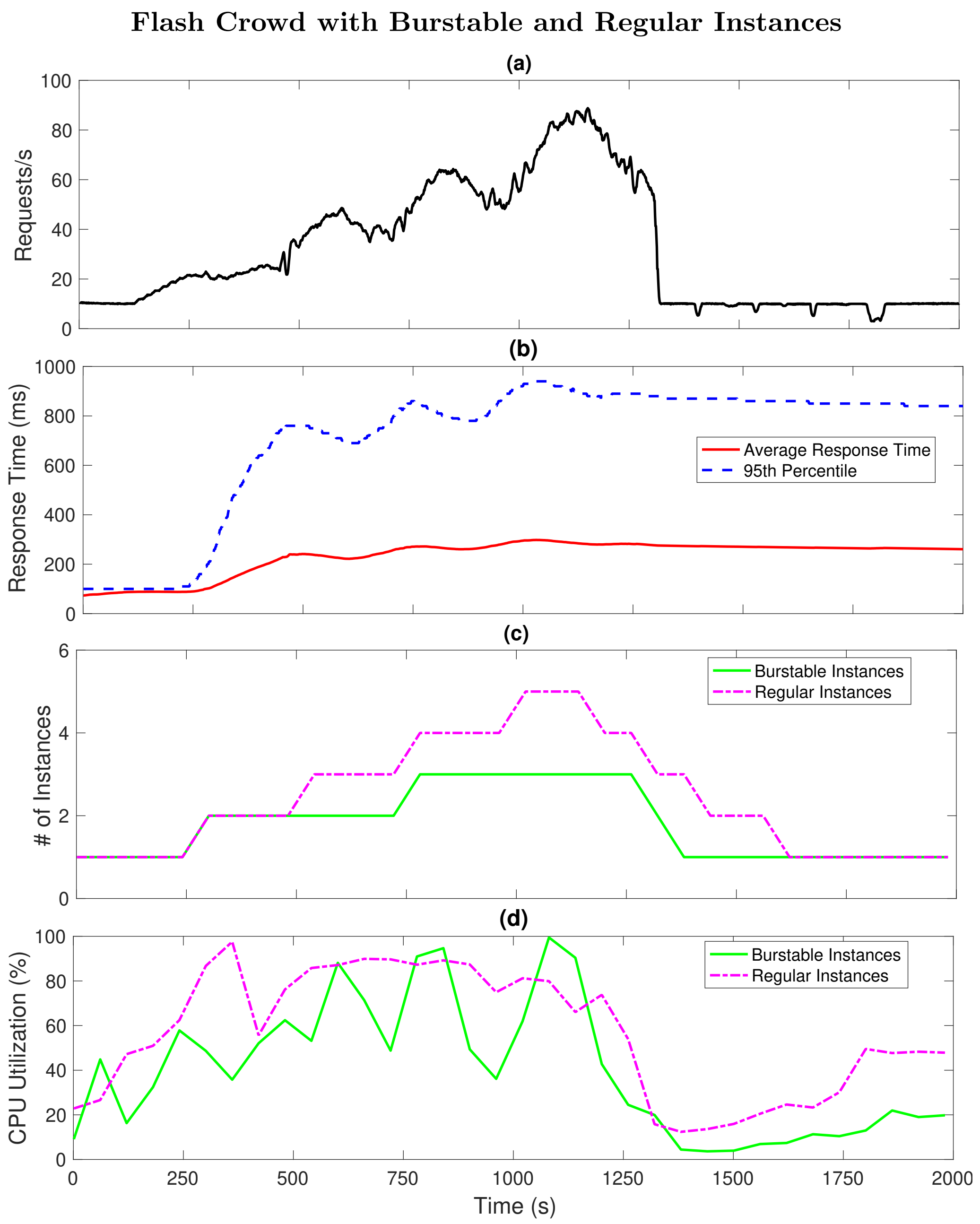

Figure 2.7: Results for the flash crowd experiment with burstable and regular instances.

configuration outperformed the former one by almost 2 times. Since we did not over-provision the cluster, the average and the 95th percentile of the response time for the flash crowd experiment running with both burstable and regular instances was almost double the figure seen on the transient queueing experiment. The average response time for the flash crowd experiment when BIAS Autoscaler ran with burstable and regular instances was 232 ms whereas for regular instances only was 141 ms.

Although the response time achieved when combining burstable and regular instances was almost two times higher than when using regular instances only, the cost savings should be considered when using this approach for scaling cloud resources. Overall, using the SR Rule algorithm for scaling regular and burstable instances was adequate for scenarios where the traffic increases at a steady and constant rate. For this application, BIAS Autoscaler was able to maintain roughly the same SLOs as for regular instances only at the same time the cost was reduced by 25%. However, the same outcome was not reached when using this strategy for handling flash crowds. For these sudden variations in traffic, a fine-granular tuning of the scaling frequency and the number of burstable instances used should be performed to avoid impacting the SLOs.

2.5 Related Work

Many works [16], [23], [24] have been done on the theoretical analysis of burstable instances, and some frameworks were proposed to AWS. The framework CEDULE developed on [16] and [24] investigates the usage of burstable instances on AWS, and proposes an adaptive scheduling framework to optimize performance and reduce cost on cloud providers. Even though the authors of CEDULE claimed it could be used in any cloud provider, they only tested it on AWS. Unlike our framework, CEDULE focuses only on token-based systems to manage burstable instances (present on AWS and Azure) while BIAS Autoscaler addresses the practical applications of these instances on non-token-like systems such as the one on GCP. Also, CEDULE is not open-source, and no information about its internal architecture is provided by its authors.

Similar to CEDULE, BurScale [15] leverages burstable instances to reduce cost and handle flash crowds on AWS. Unlike our solution, BurScale is validated and applied only on AWS, and it uses a customized load balancer. Even though BurScale is an open-source solution, there is no information on how it could be used on a non-token-like system such as GCP. When using BIAS Autoscaler on GCP, however, the load balancer used is the Google Cloud Load Balancer for controlling the traffic of the burstable instances.

The authors on [23] advocate the usage of burstable instances for workloads that do not require large amounts of computational resources to run, and that occasionally need to run with additional resources for small periods. They also propose the first analytical model for burstable instances that takes into account the QoS metrics and CPU credits of burstable instances to derive a mathematical model that maximizes cost and resource efficiency for customers and cloud providers for IaaS (Infrastructure as a Service) clouds. Although their framework can be used to model burstable instances on AWS and Azure, there is no information on how these instances can be used on GCP.

Some solutions [25], [26] were developed on the efficiency of reactive rule-based autoscaler on Google Compute Engine, whereas others [27], [28] proposed scaling policies for Google Kubernetes Engine. However, no studies were found on the practical usage of burstable instances on GCP. The analysis on [29] presents a complete overview and comparison between burstable instances from AWS and GCP, and explains in detail the token mechanisms used by AWS to manage these instances.

MRburst, which was developed on [30], is also a performance scheduler to control, among other things, the CPU utilization of burstable instances in the network level on AWS to maximize cost efficiency. However, this approach differs from ours since we control our resource utilization on the application level performing load balancing configuration changes.

2.6 Conclusion

Burstable instances can be the key to improving resource efficiency and reducing costs on the public cloud. We presented BIAS Autoscaler, an autoscaler that leverages burstable instances on the public cloud as the only of its kind fully validated and integrated on the Google Cloud Platform. We applied a known scaling model to validate our concept, and achieved promising results on both savings and resource efficiency. By replacing some of the conventional instances with burstable instances, BIAS Autoscaler was able to reduce the cost by 25% while maintaining the same service SLOs compared with traditional approaches using regular instances only.

We evaluated BIAS Autoscaler under a transient queueing and a flash crowd experiment, and showed its efficiency on Google Cloud Platform on a microservice workload. The outcome of these performance experiments showed great potential to increase resource efficiency and reduce the cost. These results demonstrated that BIAS Autoscaler can increase resource efficiency by 42% without interfering with the quality of the service when using burstable instances.

Chapter 3

Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda

Serverless computing has emerged in recent years as the new computing paradigm adopted by key players in the industry for software development. This new paradigm has seen rapid growth in adoption due to its unique billing model and scaling characteristics. Public cloud providers such as Amazon Web Services (AWS) offer several configurations and language runtimes for their serverless functions. Although extensively explored by the research community, this field still lacks current studies that address the many challenges developers face when leveraging serverless functions for real-world applications. One of these challenges that are often overseen by many programmers is the cold start problem which is present in any serverless application. For this reason, we propose the first study to characterize the underlying cold start impacts caused by the choice of language runtime, application size, memory size and deployment type on AWS Lambda. In this chapter, we analyze the performance of the *container-based deployment* and *ZIP-based deployment* of AWS Lambda using a variety of language runtimes and applications running with different function configurations; then we propose guidelines for developers and cloud managers to consider when deploying/managing the workloads on the cloud.

3.1 Introduction

Cloud computing has become the new standard for running applications and workloads in the industry. Most of the companies nowadays host their services on private or public clouds, and public cloud providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP) and Microsoft Azure (Azure) offer support to most of the services required by complex systems and applications. Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Service as a Service (SaaS) are the most popular cloud computing service models used in the industry [31].

Serverless computing, often referred to as Function as a Service (FaaS), is the most recent cloud computing paradigm that originated from the SaaS model. With this new computing paradigm, developers deploy small pieces of code named functions and the cloud provider is in charge of provisioning all the infrastructure and resources required to run these functions. Unlike IaaS or PaaS, where cloud admins still need to provision their cluster, FaaS does not require resource provisioning or complex configurations. The cloud provider is responsible for managing the FaaS infrastructure, allowing customers to deploy and execute their functions in an efficient and faster way compared to the older computing models. With FaaS, cloud providers charge per request, not per resource used. The value of each execution varies according to the function configuration and the region it is executed.

Most of the public cloud providers have their own FaaS platforms and services. Cloud Functions are functions offered by GCP where customers can deploy and run their code with zero server management [3]. Azure also provides their FaaS service named Azure Functions[4], IBM Cloud Functions are the serverless functions offered by IBM [32] and AWS has the AWS Lambda functions [5]. AWS is currently the only public cloud provider to offer different deployment types for their FaaS services. Previously, all major FaaS platforms, including AWS Lambda, supported only one deployment type. Customers were required to upload their function source code into the cloud platform either manually or by compacting the code and deploying it automatically. However, in December 2020, AWS introduced the new deployment type named *container-based deployment* for AWS Lambda[33]. Now, customers can build their own container using the container images provided by the company. The size limitation that was once present on all serverless platforms is now eliminated on AWS Lambda since they allow the container to be as large as 10GB.

One of the most important challenges that all FaaS platforms share is the cold start delay. When a function is deployed in the cloud, a set of sequential processes is executed when this function is first invoked. First, the function source code is fetched from storage, deployed in a container, and then initialized. Once the first request is handled, the following requests are executed faster for a certain period. Then, since the container is already initialized, the function enters into idle mode, and all following requests are executed instantaneously. However, cloud providers can decommission the function container at any time, and whenever it does, clients will experience a cold start again for their requests. This issue is often addressed by researchers as "the cold start problem" , and many works [34]–[36] try to mitigate this problem faced by FaaS. Still, most of these approaches used to reduce the cold start time of serverless functions often require additional software and architectures which are often seen as complex implementations by developers since it requires extra computational resources and maintenance. Additionally, the lack of knowledge about the benefits and drawbacks of each deployment type of AWS Lambda makes this matter even worse.

Before the release of the new *container-based deployment* on AWS Lambda, developers had to compress their source code - including the external libraries and dependencies - in a ZIP file which had a size limit of 250MB (uncompressed). This file is usually referred to as the "Function package file", and when the Lambda is executed for the first time, it imports and uncompresses this function package which is stored in an S3 Bucket, then loads the function for execution in a container. This process is time-consuming, especially for large package sizes. The new *container-based deployment* eliminates the unzip and container building process, which in turn, could lower the initialization time. For this deployment type, a Docker image is built locally and then sent to be stored on the Amazon Elastic Container Registry (ECR) for use with AWS Lambda. AWS providers base container images for all supported language runtimes and system architectures. Although this new *container-based deployment* shows many advantages over its traditional counterpart, little research has been conducted using this new approach.

In this research, we aim to fill in this knowledge gap by showing how to use application and platform knowledge to reduce the initialization time on AWS Lambda. We analyze the new *container-based deployment* and compared it against the traditional package deployment named *ZIP deployment* of AWS Lambda. We test the performance of several real-world applications under different language runtimes and architectures, and derive guidelines that developers and cloud admins can use to mitigate the cold start problem on AWS Lambda. We aim to answer three research questions:

- **RQ-1:** What is the impact of the AWS Lambda package size on the initialization time when using the *container-based deployment*?
- **RQ-2:** How does the memory allocated to the AWS Lambda function affect the initialization time when using the *ZIP deployment* compared to the *container-based deployment*?
- **RQ-3:** Does the machine learning model size have the same impact on both deployments of AWS Lambda functions?

The contributions of our research are:

- Presenting the first extensive analysis of AWS Lambda that takes into account the *ZIP* and *container-based deployment*, the language runtime, the memory and the package size of the function.

- Suggesting guidelines for reducing the cold start delay on AWS Lambda by choosing the ideal deployment type based on application knowledge.

This research is structured as followed: We start by explaining the approach we used for analyzing the performance of the *container-based deployment* and the applications tested (section 3.2), then we present the results for the execution time and cold start performance of the two deployment types under different configurations (section 3.3). The package size impact on the initialization time (section 3.3.2), the memory size (section 3.3.2), the model size (section 3.3.2), and the language runtime choice (section 3.3.2) are discussed, and guidelines are suggested. Finally, we discuss the related work (section 3.4) before concluding this research (section 3.5).

3.2 Methodology

In this section, we go over the details of the methodology proposed in this research. We answer **RQ-1** and **RQ-2** by analyzing the cold start and execution time of 13 serverless functions deployed with the *ZIP deployment* and *container-based deployment* and different memory sizes. Finally, we answer **RQ-3** by studying the response time of a machine learning function deployed with 5 distinct model sizes using the two deployment types and a wide range of memory configuration.

3.2.1 Language Runtimes and Libraries Used

The choice of libraries and language runtimes to use in our analysis was based on the most used libraries and language runtimes on AWS Lambda in the industry. We have selected a wide range of Python libraries that are used in many studies with serverless functions [37], [38]. In particular, we focus on image processing and machine learning applications using the *TensorFlow*, *Pillow* and *Sklearn* libraries which are also analyzed on [39]. The most used language runtimes on AWS Lambda are Python,

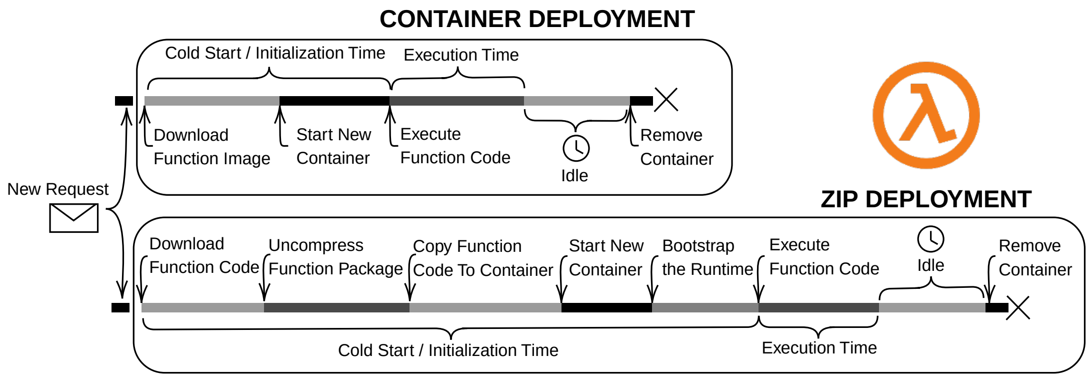

Figure 3.1: AWS Lambda lifecycle (adapted from [44]).

Node.js and Java, respectively [40], [41]. However, Node.js applications particularly have never been tested with the new *container-based deployment* on AWS Lambda.

3.2.2 Function Deployment Configuration

Fig. 3.1 shows the two deployment types of AWS Lambda in detail. The lifecycle of the *container-based deployment* of AWS Lambda is not documented by the company. However, we assumed that the AWS Lambda Manager starts the container as soon as it is fetched from storage in the ECR. This process could, in theory, speed up the start-up time for some language runtimes since it does not need to build the container image. We consider the cold start time - also referred as the initialization time - the period from the invocation of the function to the time the code is ready for execution. Although the objectives of *RQ-1* and *RQ-2* over the *ZIP deployment* have already been answered in previous studies [42], [43], the effects on the *container-based deployment* of AWS Lambda remain unknown to this date. Finally, the execution time is the time that the function takes to execute when its container is up and running.

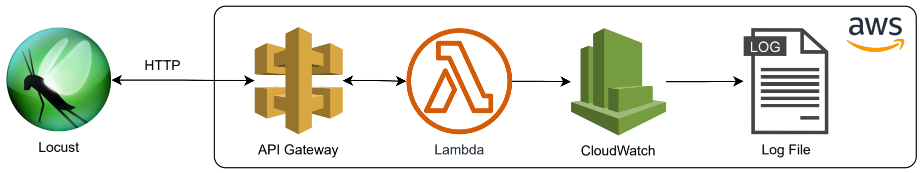

Figure 3.2: Architecture built on AWS.

3.2.3 Function Response Time Measurement

We invoke each Lambda using the AWS API Gateway through a RESTful API. In particular, we follow the same benchmark configuration adopted on [45] where AWS API Gateway is used for creating endpoints for invoking the Lambdas. AWS CloudWatch is used for storing the execution log of both the Lambda and the API Gateway call. We use the Locust[1] benchmark tool for our performance tests. Locust is an open-source, scriptable, and scalable performance testing tool that allows customized use test cases written in Python. Fig. 3.2 shows the architecture view used for each Lambda function. We use the fields *@billedDuration* and *@initDuration* from the original Lambda log file to calculate the execution time and initialization time of each execution, respectively. For the execution time analysis, we check the *@logStream* to extract the container identifier to validate the time by which the container is decommissioned and a new one is launched.

3.2.4 Function Workloads Used

We evaluate the performance of the two Lambda deployments with 13 serverless functions and 3 different language runtimes. These functions are coded in Python, Node.js and Java and tested on both arm64 and x86 architectures. A wide range of libraries and package sizes are used on each of these applications. All the 13 Lambdas are invoked by the Amazon API Gateway. Table 3.1 compiles all applications used.

Image Classifier 230: Consists of a binary image classification function using one of the most popular and comprehensive open-source machine learning libraries,

[1]https://locust.io

the *scikit-learn (sklearn)* [46]. It receives an image of any size, and it predicts which of the two classes this picture belongs to. In order to do the predictions, we first reduce the image to 59×59 pixels using the *Pillow* library, then we extract the Histogram of Oriented Gradients (HOG) and run a Support Vector Machines (SVMs) model. We train 5 SVMs models with different sizes (1MB, 7MB, 12MB, 16MB and 20MB) to evaluate the impact of the model size in the cold start of the application. These models are deployed on the Image Classifiers 230 to 249 from smallest to largest, respectively.

Linear Regression: This application uses the open-source scientific computing library for Python *SciPy*. Since its initial release in 2001, *SciPy* has become the most used library for scientific algorithms in Python [47]. We compute a simple and yet commonly used mathematical operation with this library, the least-squares regression for two sets of measurements.

Image Black and White: We convert an image to black and while using the OpenCV library. This library is widely used among developers for computer vision applications.

TF Image Classifier: It is a popular open-source serverless application that uses the *TensorFlow Lite* library. It consists of a multi-class image classifier using the on-device inference framework from TensorFlow and both the *ZIP* and *container-based deployments* are available on GitHub[2].

Resize and Feature: This function is part of the Image Classifier 230, and it performs the resize and feature extraction of an image of any size using the libraries *Pillow* and *Numpy*.

Resize: This application is a simple application used to resize an image, and it is also part of the Image Classifier 230. It uses only the *Pillow* library.

Factorial: Performs the factorial of a given number. It does not use any external libraries. There are three versions of this application, one for each language runtime.

[2]https://github.com/edeltech/tensorflow-lite-on-aws-Lambda

Table 3.1: Serverless applications.

App Name	**ZIP Size (MB)**	**Image Size (MB)**	**Architecture**	**Runtime**	**Libraries**
Python					
Image Classifier 230	230	480	arm64	Python 3.8	Pillow, Numpy, Pillow, Numpy Sklearn, Joblib, Scikit Image
Linear Regression	186	283	arm64	Python 3.8	Scipy, Numpy
Image Black and White	126	256	arm64	Python 3.8	OpenCV, Numpy, Pillow
TF Image Classifier	83	340	x86	Python 3.7	TensorFlow, Numpy, Pillow
Resize and Feature	64	210	arm64	Python 3.8	Pillow, Numpy
Resize	14	182	arm64	Python 3.8	Pillow
Factorial Python	0.004	176	arm64	Python 3.8	
Node.js					
Large Size Node	234	323	x86	Node 14	gulp-imagemin, jspdf, html-pdf, natural, text-extractor, jimp
Medium Size Node	77	204	x86	Node 14	natural, sharp
Factorial Node	0.006	148	x86	Node 14	
Java					
Large Size Java	133	227	x86	Java 11	OpenIMAJ
Medium Size Java	15	188	x86	Java 11	iTextPDF
Factorial Java	10	185	x86	Java 11	

Large Size Node.js: This application uses some of the most popular Node.js packages from the official NPM Registry website[3]. It performs image processing as well as natural language operations.

Medium Size Node.js: This function performs natural language operations using the well-known package *natural*.

Large Size Java: It is a popular open-source application that performs image processing and mathematical operations using the award-winning library *OpenIMAJ*. This application is openly available on GitHub[4].

Medium Size Java: This function converts a text input to a PDF document using the popular library *ItextPDF*.

We classify all 13 applications into different application size groups according to the language runtime aiming to answer RQ-1. The Image Classifier 230 is used to answer RQ-2 and RQ-3, and finally, the three factorial applications are analyzed to visualize the impact of the language runtime on the two deployment types.

3.3 Experimental Evaluation

We performed a combination of tests on both the AWS Lambda *ZIP* and *container-based deployments* and analyzed the execution time and initialization time of each configuration. The experiments were conducted over an extended period of time from December 28th, 2021, to January 18th, 2022. All tests were performed in the AWS region *us-east-1* where each Lambda handles one request per time. The benchmark tests were done using both the arm64 and x86 architectures. We compared the two deployments using several metrics that include the median, the 95th percentile and the average of the response time and the initialization time, as well as the cumulative distribution function (CDF) of all experiments. We first discuss the experimental setup of each test, and then present the results and discussion of the execution time

[3]https://www.npmjs.com
[4]https://github.com/eugenp/tutorials/tree/master/image-processing

followed by all three research questions and the impact of the language runtime. All the experiments are openly accessible on GitHub[5].

3.3.1 Experimental Setup and Data Collection

We first measured the execution time of all 13 applications for both Lambda deployments. Each application was invoked 400 times with requests sent every 1 second. This interval guarantees no queue is formed since the worst execution time is less than 1 second. During this period of approximately 7 minutes, none of the containers were decommissioned. All the logs from AWS CloudWatch, including the Lambda and API Gateway log files, were analyzed and validated. We studied the execution time for 128MB, 256MB, 320MB, 384MB, 448MB and 512MB memory configurations.

Then, we evaluated the cold start performance of all applications with different memory configurations. We invoked the Lambdas 20 times with requests sent every 10 minutes. In total, each test lasted 200 minutes, and it was conducted with 128MB, 256MB, 320MB, 384MB, 448MB and 512MB memory configurations. According to our benchmark tests, the idle time of AWS Lambda was less than 10 minutes for all programming languages. We use 10 minutes interval between invocations to compute the cold start time of AWS Lambda. This interval guarantees that each call is executed by a new container.

The second set of tests for the cold start were aiming to study the memory impact on the cold start time. We tested the Image Classifier 230 under 128MB, 256MB, 320MB, 384MB, 448MB, 512MB, 640MB, 704MB, 768MB, 832MB, 896MB, 960MB, 1024MB, 1088MB, 1152MB, 1216MB, 1280MB, 1344MB, 1408MB, 1472MB, 2048MB and 3008MB memory configurations. Finally, we used 5 SVMs models from different sizes to study the relationship between the model size and cold start time on both deployments. In order to do this, we used different amounts of data to train these 5 models.

[5]https://github.com/pacslab/serverless-iot-deployment

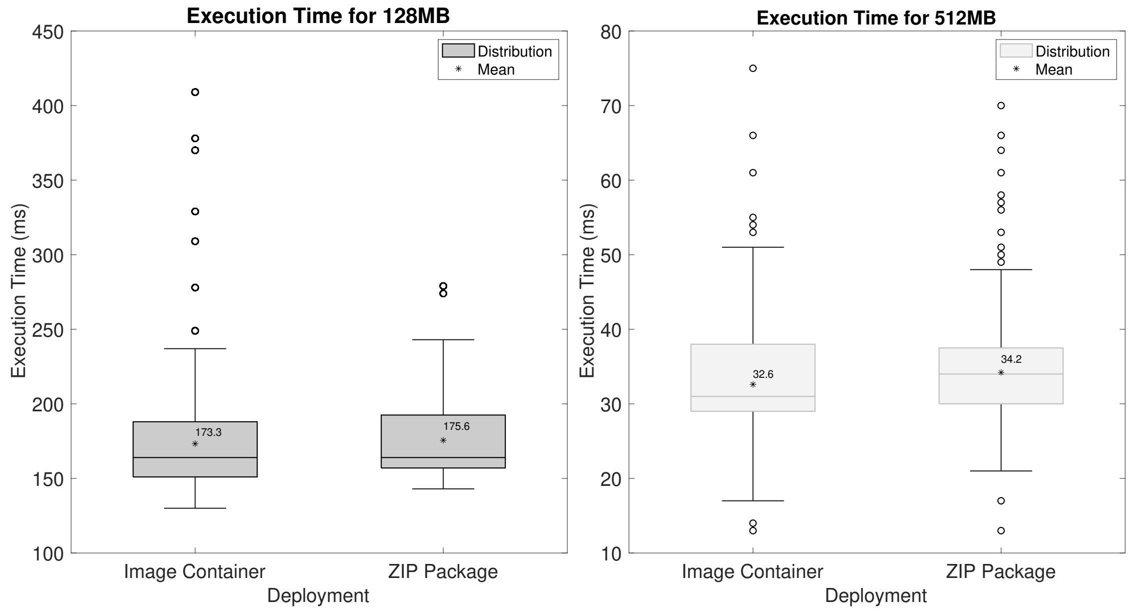

Figure 3.3: Execution time for the Image Classifier 230.

3.3.2 Results and Interpretations

We tested the execution time and cold start time of all 13 functions for both Lambda deployments, and present the results for each research question below.

Execution Time

In order to answer **RQ-1, RQ-2** and **RQ-3**, we need to first validate whether or not the execution time differ according to the deployment type. Therefore, we analyzed the execution time of all 13 applications. Fig. 3.3 shows the execution time distribution for two memory configurations of the Image Classifier 230 application. As we can see, both execution times are statistically equivalent. The same was seen for the Node.js and Java applications. Furthermore, the execution time of the AWS Lambda *ZIP* and *container-based deployments* are equal for all memory sizes and language runtimes. Although we have noticed a slight difference in the execution time in favour of the *container-based deployment*, both measurements are statically equivalent.

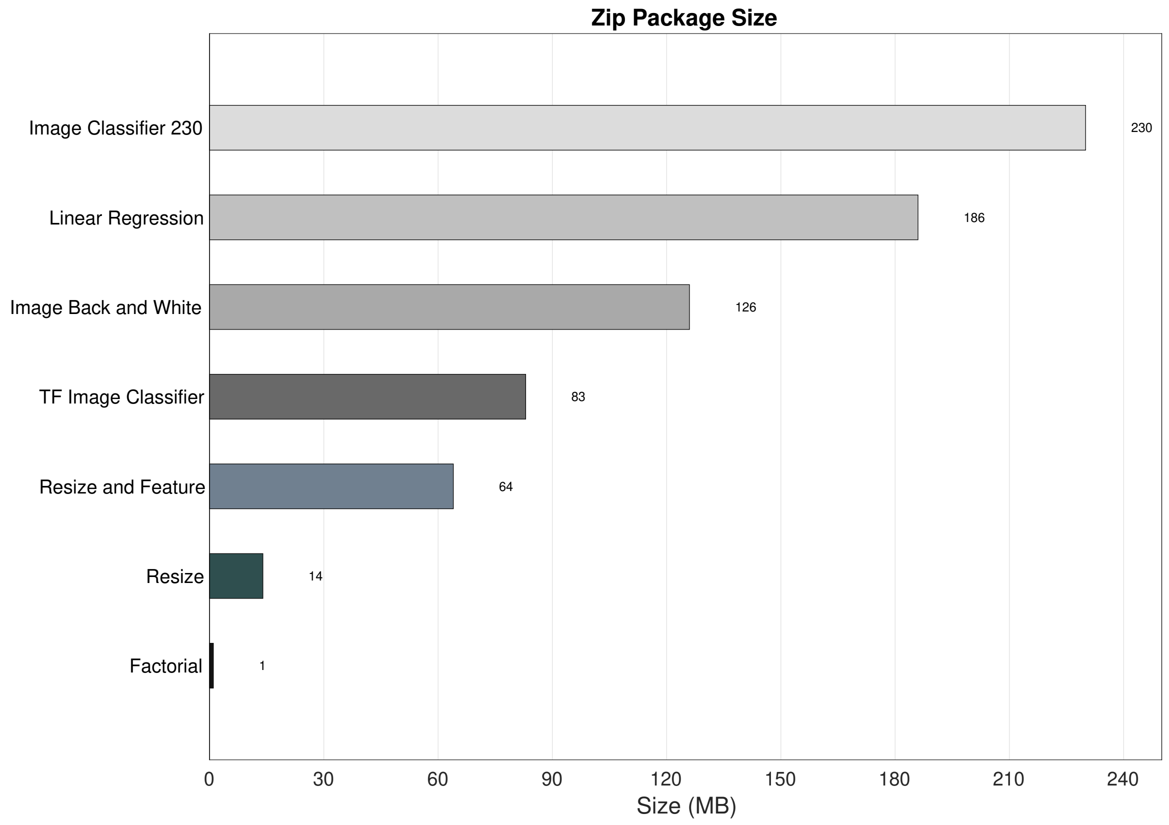

Figure 3.4: ZIP package size of all Python serverless applications.

RQ-1: What is the impact of the AWS Lambda package size on the initialization time when using the *container-based deployment*?

On **RQ-1** we want to study the impact of the package size of the Lambda function on the *ZIP* and *container-based deployments*. We first studied this on Python functions, and then moved to the other programming languages. The right chart of Fig. 3.5 shows the different sizes of the *ZIP deployment*. We measured the initialization time of both deployments, and then calculated the percentage of improvement of the *container-based deployment* over the *ZIP deployment*. This data is presented on the right chart of the Fig. 3.5.

We observe two trends with regards to the package size of **Python** applications. The *container-based deployment* shows a better performance for large package sizes and small memory configurations. For instance, the initialization time of the Image Classifier 230 was 78% smaller than the *ZIP deployment* for 128MB memory. As

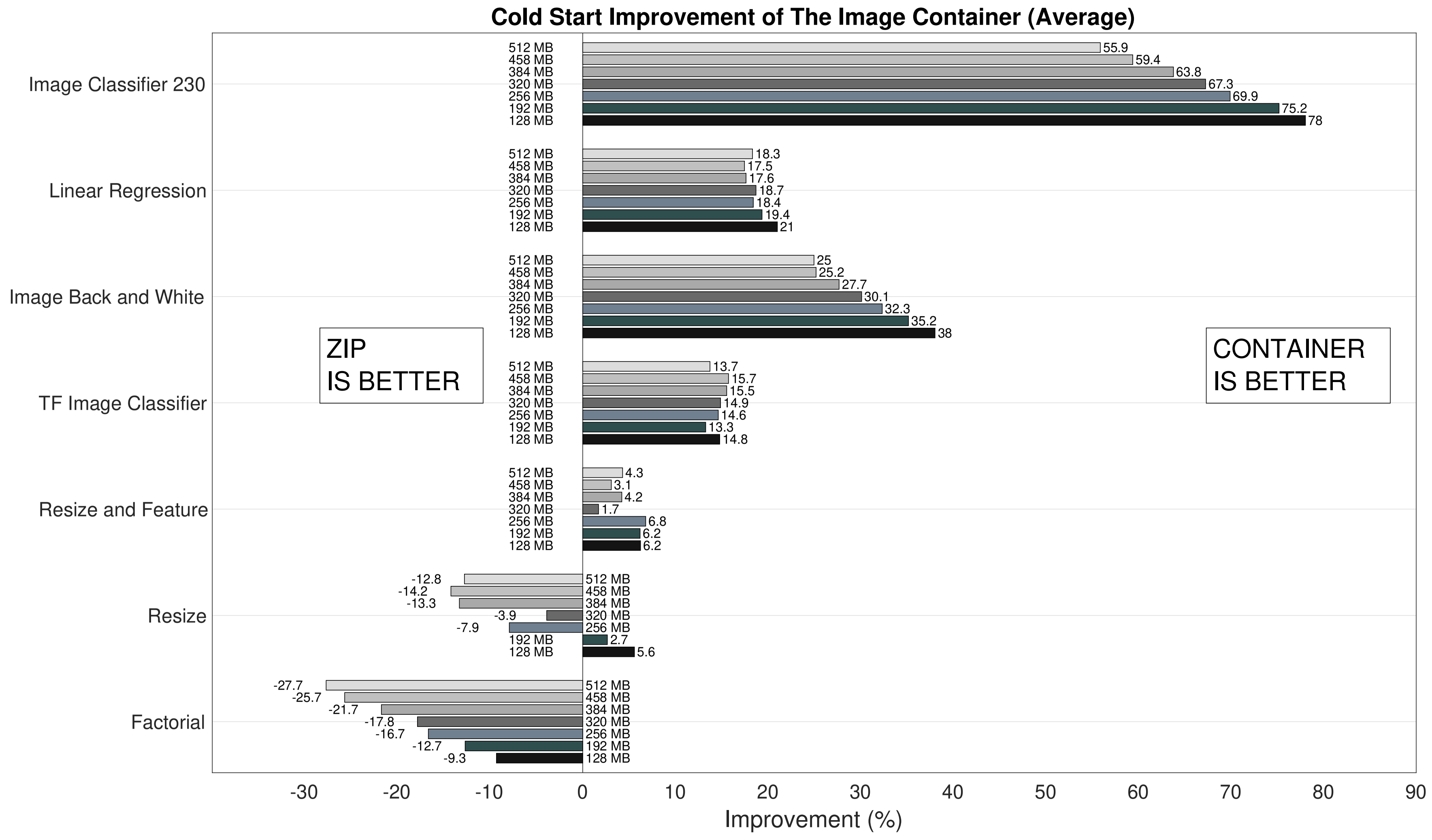

Figure 3.5: Initialization time improvement of the *container-based deployment* over the *ZIP deployment* of the Python functions.

we increase the memory size, this difference decreases and the two deployment types become similar in cold start time. For most of our tests, smaller memory configurations favoured the *container-based deployment* since this Lambda deployment type presented a smaller cold start time.

Our results contradict the academic work presented on [45] where the *container-based deployment* of Python applications showed worse initialization time than the *ZIP deployment*. Unlike our experiments, the authors on [45] used only one Python application - with a small package size - and tested it with one memory configuration only. Even though they only evaluated one specific scenario, they claimed that the *container-based deployment* was worse than the *ZIP deployment* for interpreted programming languages such as Python when it comes to cold start time. In contrast, our results showed that the size of the Lambda package - the ZIP archive or Docker container image - is, in fact, a key factor in the initialization time of these two deployments.

Another observation is that the package size impacts the cold start time of both deployments. Small package sizes had better initialization time with the *ZIP deployment*. For the Python Factorial function, which has only 4kB in size, the *container-based deployment* was worse than the *ZIP*'s for all memory configurations with figures varying from 9.3% to 27.7% worse initialization times for 128MB and 512MB, respectively. These findings are in line with the results presented on [45], and it may be due to the fact that Python functions are non-static binary programs, and consequently, all libraries and modules have to be imported dynamically which adds overhead in the initialization time of the *container-based deployment*.

One of the most important findings is the inflection point by which the *ZIP deployment* becomes better than the *container-based deployment*. This can be seen when we analyze the initialization time for the Resize application which has 14MB in size, and it uses a popular Python library for image manipulation - *Pillow*. From this data, we can see that the point of inflection is located between 192MB to 256MB where the *container-based deployment* becomes worse than the *ZIP*'s by 7.9%. This is because while for 192MB the former deployment was 2.7% better than the latter one, for 256MB the container-based was 7.9% worse than the *ZIP deployment*.

Therefore, we advocate that developers and cloud managers should choose the *container-based deployment* whenever they are using a large Python application with external libraries and dependencies. In our tests, Python applications with sizes equal to or larger than 64MB are better with the *container-based deployment* for memory configurations of up to 512MB. If, however, developers are deploying smaller Python applications of 14MB in size, for instance, further testing is necessary to find out the best deployment according to their memory needs. In our tests, Python applications with 14MB in size and up to 192MB memory are better with the *container-based deployment*. After this point, the *ZIP deployment* becomes a more suitable option. Finally, small Python applications of 1MB or less in size that have no libraries or external modules should be deployed using the *ZIP deployment* since it offers a better initialization time.

Even though both Python and Node.js are dynamically-typed languages, the performance of these languages under the *container-based deployment* is different. The left chart of Fig. 3.6 shows the percentage of improvement of the *container-based deployment* for Node.js functions. For large **Node.js** applications, the *container-based deployment* showed a similar cold start compared to the *ZIP deployment*. Both the average and 95th percentile were statically equivalent for most memory sizes. For instance, for 320MB memory, the average differs in only 0.43% in favour of the *ZIP deployment*. Although we observed a tiny improvement in the average of the *container-based deployment* for small memory sizes of 5.9 %, the 95th percentile of this deployment is in fact 10% worse than the *ZIP deployment*. Therefore, the choice of deployment has little to no impact on the initialization time of large Node.js applications.

When it comes to small packages sizes, the same outcome found for Python applica-

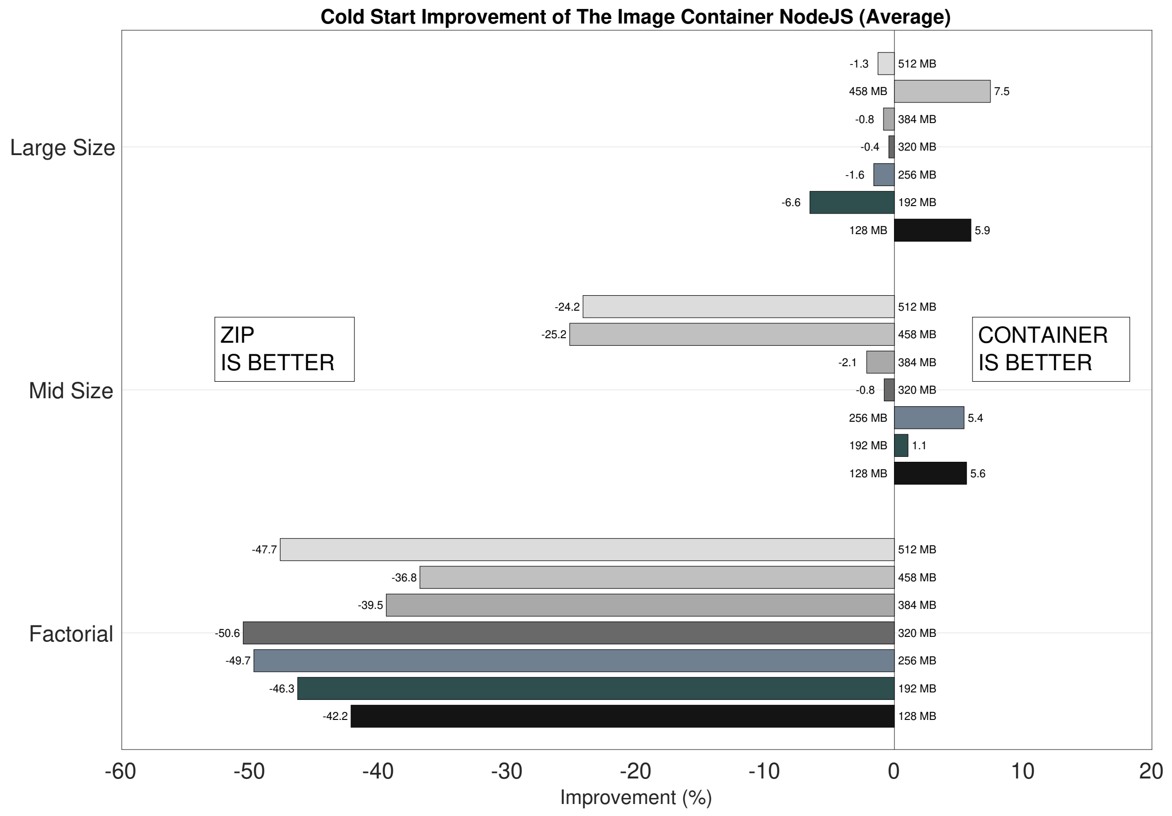

Figure 3.6: Initialization time improvement of the *container-based deployment* over the *ZIP deployment* of the Node.js functions.

tions also applies to Node.js functions. This is because the Node.js Factorial function was also faster with the *ZIP deployment*. Similar to the Python applications, as we increase the application package size, this difference becomes smaller and the two deployments perform similarly. This can be seen when we analyze the initialization time for medium size Node.js applications. However, unlike Python applications, Node.js applications performed better or equal with the *ZIP deployment* on all package sizes. Therefore, we recommend that cloud admins use this type of deployment for Node.js applications of any package size up to the 250MB limit.

Unlike the results seen with the dynamically-typed languages Python and Node.js, the performance of **Java** functions, which is a is a statically-typed language, is quite distinct. The right chart of Fig. 3.7 shows the percentage of improvement of the *container-based deployment* for Java applications. Overall, the *ZIP deployment* out-

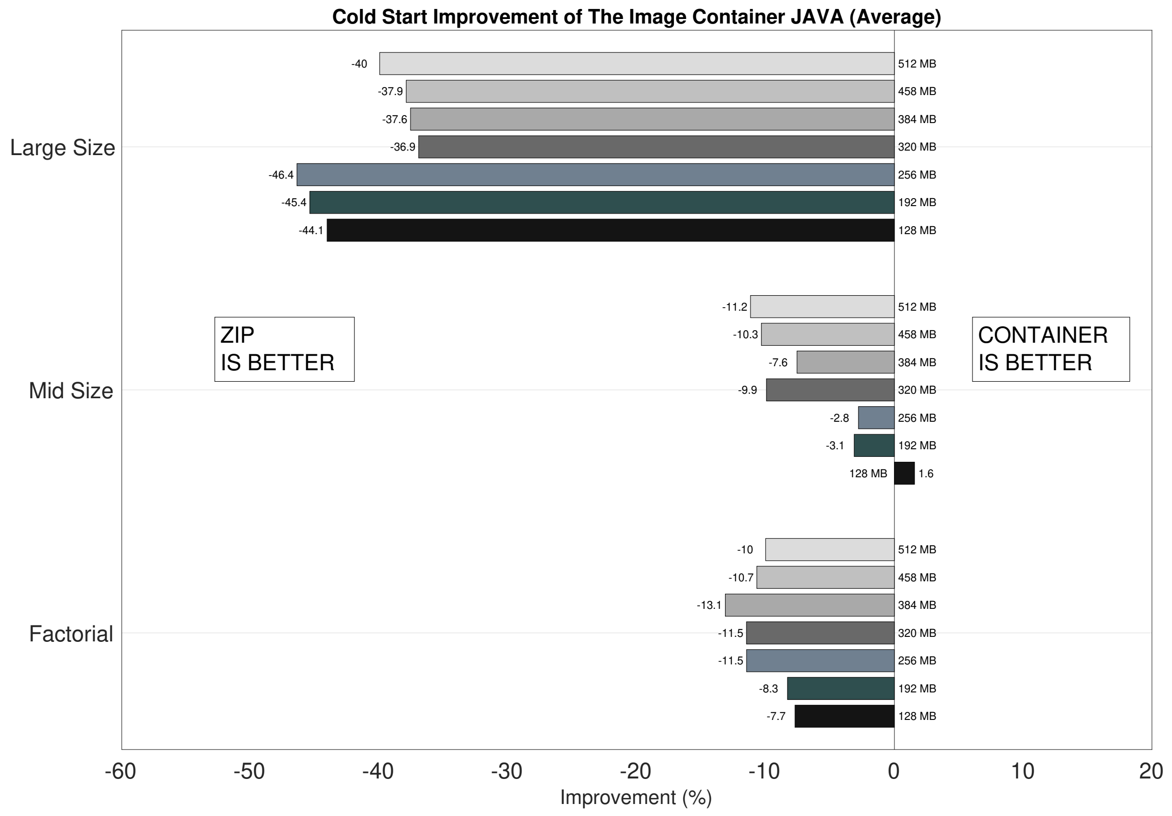

Figure 3.7: Initialization time improvement of the *container-based deployment* over the *ZIP deployment* of the Java functions.

performed the *container-based deployment* on all memory and package sizes. In fact, we have noticed the opposite trend where larger applications performed significantly better with the *ZIP deployment*. For instance, the average cold start time of the *container-based deployment* was 46.4% worse than the *ZIP*'s for the large size Java function with 256MB memory configuration. As we decrease the package size, this difference also decreases but for almost all configurations the *ZIP deployment* is preferable.

Our results contradict the findings on [45] where a Golang function - also a statically-typed language - is analyzed. The authors on [45] advocate that statically-typed languages such as Golang and Java have similar initialization times on both deployments. However, they only tested one package size and did not test Java applications. From our findings, we can also conclude that the *ZIP deployment* should be used for Java

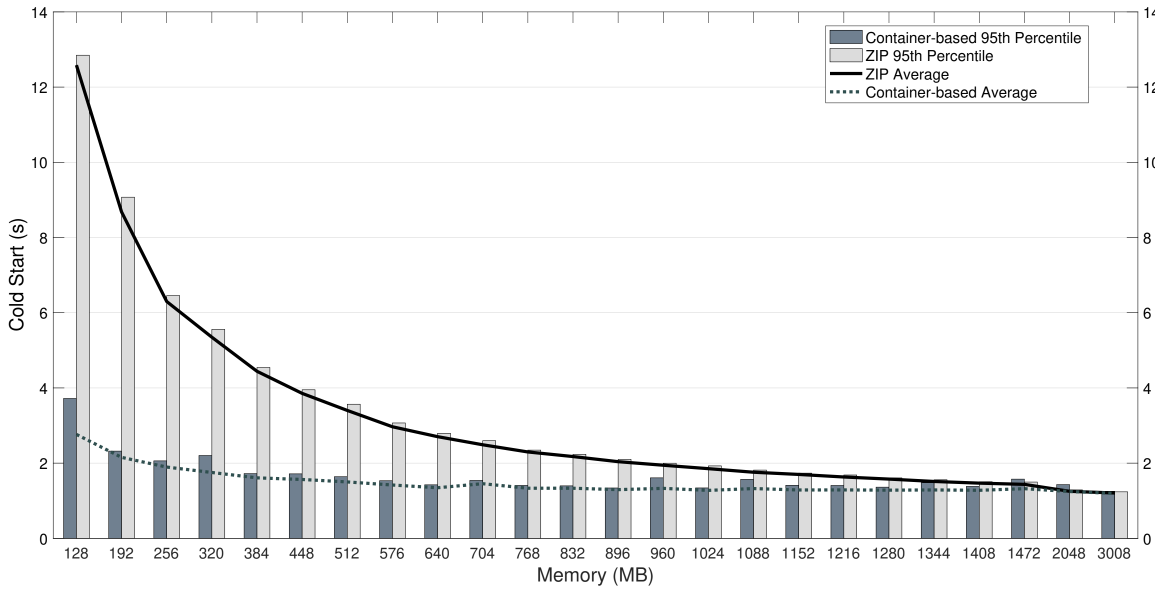

Figure 3.8: Initialization time of the Image Classifier 230 under different memory configurations.

applications of any package size up to the 250MB limit.

Recommendations of RQ - 1

- *Python applications with large package sizes have faster initialization time with the container-based deployment while small applications are better with the ZIP deployment. Our experiments show that the inflection point is 64MB in size.*
- *Node.js applications with small package sizes have faster initialization time with the ZIP deployment while medium and large size ones have approximately equivalent cold start time to some extent.*
- *Java applications of any package sizes have faster initialization time with the ZIP deployment.*

RQ-2: How does the memory allocated to the AWS Lambda function affect the initialization time when using the *ZIP deployment* compared to the *container-based deployment*?

Moving on to **RQ-2**, we wanted to see if the memory size had the same impact in the cold start time of the two different deployments for all three language runtimes. First, we analyzed only **Python** applications. Fig. 3.8 shows the Image Classifier 230 under a wide range of memory configurations. As expected, the *container-based deployment* presented a significantly smaller initialization time for memories from

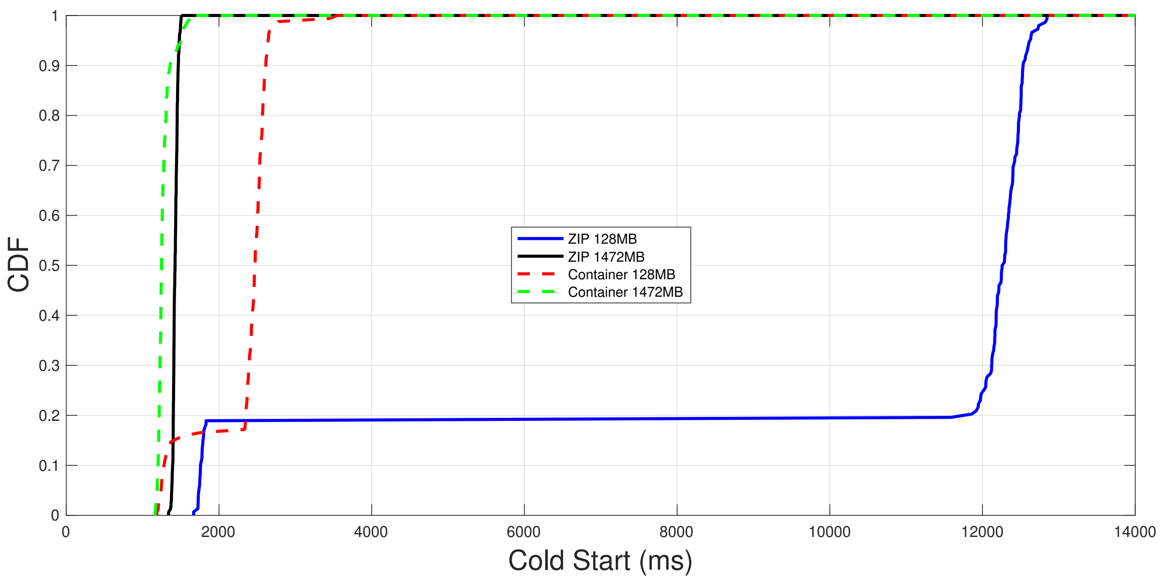

Figure 3.9: CDF of the Image Classifier 230 initialization time.

128MB to 512MB. As we increase the memory size, this difference gets smaller and the two deployments become equivalent eventually.

This behaviour is seen when we analyze the average and 95th percentile initialization time for the 1472MB memory configuration. Even though the average initialization time of the *container-based deployment* is 8% better, the 95th percentile is 5% worse. Thus, we assumed this is the point where the two deployments reach the same minimum value. In order to further investigate the initialization time of the two deployments for very small and very large memories, we plotted the CDF of 200 executions for 128MB and 1472MB memory configurations on Fig. 3.9.

From the CDF on Fig. 3.9, we can see that the contained-based deployment with 128MB memory configuration had a cold start average 4.5× smaller than the *ZIP deployment*. However, they have near-identical cold start CDFs for 1472MB memory configuration. Thus, we conclude these two deployments have the same initialization time for memory configurations larger or equal to 1472MB. Therefore, developers should consider the size of the application and the memory allocated to the Lambda to decide on what deployment type to choose. The analysis conducted on [42] and [43] also suggests that using large memory sizes can reduce the cold start effect in AWS

Lambda when using the *ZIP deployment*. Thus, our results validate this behaviour for the *container-based deployment* as well.

However, function memory has little to almost no impact on the initialization time of both **Node.js** and **Java** applications. This is especially true for small size applications such as the Factorial one. In both these languages, the memory played no role in the performance of the cold start. The same was also seen for large-size applications where the performance of all memory configurations was similar. However, for medium-size applications, the memory size does affect the performance of the cold start. This is because both the Node.js and Java applications showed a slightly better performance in favour of the *container-based deployment* for small memory configuration. As we increase the memory size of the medium size Node.js function, for example, this difference becomes smaller and after 256MB the *ZIP deployment* becomes better. This behaviour was the same observed for medium Python applications.

As a result, the impact of the Lambda memory on both Node.js and Java functions depends on the package size. We recommend that developers and cloud admins should choose the *container-based deployment* only if the application package size is between 15-77MB for Node.js applications under small memory configurations. For both small and large functions, Node.js and Java functions should use the *ZIP deployment* instead. Finally, we tested both the arm64 and x86 architectures on all applications, and they are equally in performance. Thus, the architecture does not influence the cold start time of Lambda applications.

Recommendations of RQ - 2

- *Small memory sizes, e.g., 128MB, make the initialization time of the container-based deployment faster when deploying medium and large size Node.js and Python applications compared to the ZIP deployment.*
- *Very large memory sizes, e.g., 1472MB for Python applications, make the initialization time of both deployment types approximately equivalent.*

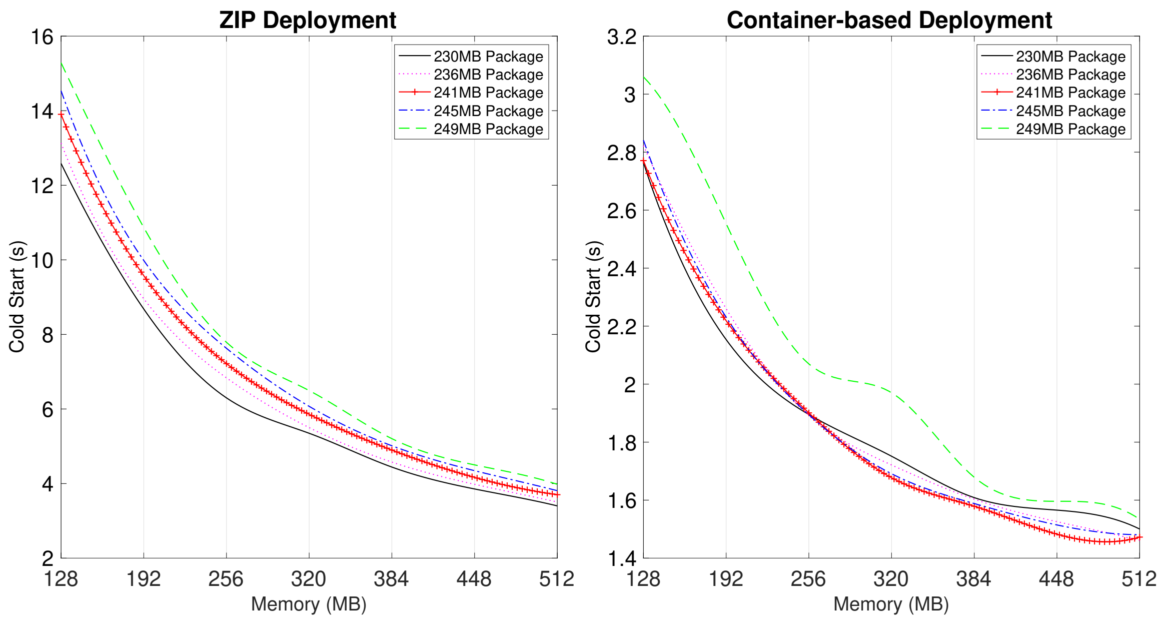

Figure 3.10: Model size impact in the initialization time of the *ZIP deployment* (left) and *container-based deployment* (right).

RQ-3: Does the machine learning model size have the same impact on both deployments of AWS Lambda functions?

Finally, **RQ-3** investigates the effect of the model size in the cold start of the functions. We tested the 5 SVMs models and included them inside the original application Image Classifier 230. The 5 applications were named Image Classifier 230MB, 236MB, 241MB, 245MB and 249MB from smallest to largest, respectively. Fig. 3.10 shows the effect the model size has on the two deployments studied. As we expected, the larger the model size, the higher the cold start time for the *ZIP deployment*. Also, the curve concave decreases as we increase the memory size for all model sizes under the *ZIP deployment*. The *container-based deployment*, however, was slightly less susceptible to the model size changes. While the variation of the model size caused a 22% change in the cold start time of the *ZIP deployment*, this figure was only 11% for the *container-based deployment* under 128MB memory configuration.

Therefore, changes in the model size of machine learning applications are less likely to cause variations in the cold start time of *container-based deployment* compared with the *ZIP*'s. This is especially true when we analyze the cold start of the Image

Classifiers 241 and 245. The two curves for these applications are approximately equivalent for the *container-based deployment*. For example, for 256MB memory, these two applications had only 0.33% difference in the initialization time of the *container-based deployment* whereas for the *ZIP* version this figure was 5.39%, which is roughly 17 times more than the former deployment.

Recommendations of RQ - 3

- *Variations in the model size are less likely to cause impact in the initialization time of the container-based deployment compared to the ZIP deployment.*

Language Runtime Impact

Previous works have shown that the choice of the language runtime affects the cold start time of AWS Lambda [48], [49]. We extend **RQ-1** and **RQ-2** by analyzing the impacts of the language runtime in the cold start of serverless applications. We analyzed the average and 95th percentile initialization time of the Factorial function in Python, Node.js and Java. Fig. 3.11 compiles the cold start time of these applications on both the *ZIP* and *container-based deployments* for 128MB memory configurations.

Our findings show a noticeable impact of the language runtime in the cold start of applications. As can be seen, both deployments had the same trend where Node.js functions were the fastest followed by Python and Java, respectively. Java applications have the worse initialization time among all three languages, with figures approximately 4 times higher than the Node.js ones for the *ZIP deployment*. For the *container-based deployment*, this figure is around 3 times which is still representative. Our results follow the work present on [48], [49] where is shown that Java functions have a high trail latency compared to Python applications. However, our work showed that Node.js applications have the best performance which contradicts the works [48], [49] since they claim that Node.js functions are in fact worse than Python applications. One of the reasons may be due to the fact that these academic benchmark tests were performed using an older version of all language runtimes analyzed.

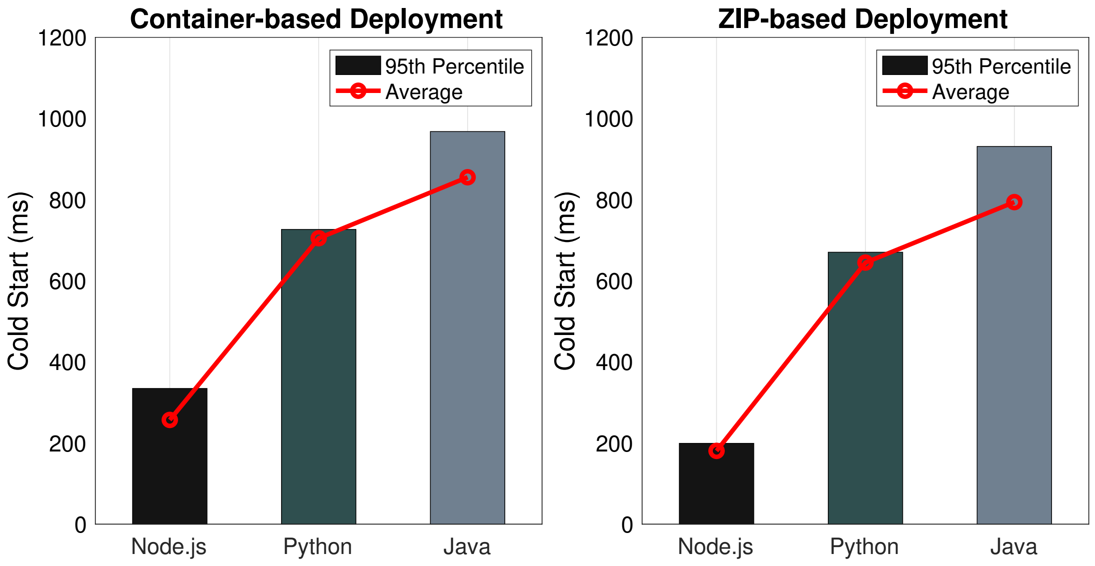

Figure 3.11: Factorial function under different language runtimes and 128MB memory configuration.

Our result is surprising since it also contradicts the work on [45] where their authors state that the language runtime has negligible implications for cold-start delays for the *ZIP deployment*. As seen on the right chart of Fig. 3.11, the impact of the programming language is significant for the 128MB memory configuration. One of the possible reasons that can explain this opposite outcome is the memory configuration allocated to the Lambda. While we tested a wide range of memory sizes - from 128MB to 3GB - the authors on [45] only tested it with 2GB which may bias their findings. However, the analysis performed on [42] and [43] also suggests that using dynamically-typed languages such as Node.js and Python can reduce the cold start effect in AWS Lambda when using the *ZIP deployment*. Our study shows that this impact is also seen with the *container-based deployment* in a corresponding degree.

In conclusion, the *container-based deployment* had a better initialization time for larger Python applications and small memory configurations. Also, it is more suited to deploy machine learning applications with embedded models. Python applications larger or equal to 64MB in size should be deployed as containers whenever they are running with 512MB or less memory. Additionally, very large Python applications of 230MB or more in size are faster when using container-based with up to 1472MB of memory. Finally, both Node.js and Java applications have, in general, faster or equal initialization time when using the *ZIP deployment*, especially for large memory configurations.

3.4 Related Work

Prior work includes a number of frameworks and FaaS architectures developed to mitigate the cold start time present in most of the public cloud providers nowadays. WLEC [50] and Pigeon [51] are alternative approaches to AWS Lambda that reduce the initialization time, and can be integrated with other serverless providers. Application knowledge is used on [52] for reducing the duration of cold start by implementing a lightweight choreography middleware for FaaS. However, all these works are complex architectures that require extra computational resources and maintenance in order to be used with Lambda functions. Cloud admins and developers could use our insights instead, and change the Lambda deployment type according to their applications to significantly reduce the initialization time.

Many benchmark tests addressed the impact of the memory size and package size on the cold start of serverless functions. Daniel et al. [53] studied the cold start time of a wide range of applications on AWS Lambda, Google Cloud Functions, Microsoft Azure Functions, and IBM Cloud Functions. This type of performance analysis was also investigated on [53]–[55], however, since no cloud provider supported container-based function deployment at the time, no prior work was developed using this new development type. Additionally, the open-source benchmark suite for characterizing serverless platforms ServerlessBench developed on [56] evaluates the cold start time of functions with different sizes on AWS Lambda using the *ZIP deployment*. Similar to our results, the authors on [56] advocate that large-size functions suffer from longer initialization time due to larger data transmission and package import overhead. However, only Python applications are studied on [56], and our work is the first to show that these impacts could be mitigated by changing the deployment type for some language runtimes.

The size limit of 250MB of AWS Lambda is challenging for most of the machine learning applications since they use large libraries and models. AMPS-Inf [57] is a framework that solves this problem by partitioning customized libraries and modules across a number of Lambda functions. Although this solution can be used for large Lambda functions, AWS Lambda now supports Docker container images of up 10GB in size, and as demonstrated on our benchmark tests, this deployment is ideal for large workloads. The choice of language runtime is also discussed on [38], [48], [49], however, our study is the first of its kind to address the *container-based deployment* with Node.js and other programming languages. In addition to the size limitation of the *ZIP deployment*, many Internet of Things (IoT) and latency-critical applications often face performance issues when using AWS Lambda due to the cold start problem. The survey on [58] shows that despite the many challenges the cold start delay brings to these applications, they still use serverless functions platforms such as AWS Lambda. Now, developers and companies can use the insights we present in this research to mitigate the cold start problem these functions face.

The only study found that partially investigates the use of container-based Lambdas is the STeLLAR benchmarking framework [45]. STeLLAR is an open-source serverless benchmarking framework that enables an accurate performance characterization of serverless deployments. It evaluates the cold start time of the *ZIP* and *container-based deployments* of Lambda applications with two types of applications. However, unlike our benchmark tests, the authors on [45] only tested their applications with a 2GB memory configuration. Their findings show that for small Python applications, the *ZIP deployment* is recommended based on the initialization time. Contrary to the aforementioned work, our findings suggest that memory, package size and language runtime are pivotal when choosing the best deployment type for better cold start performance. Additionally, unlike the work presented on [56] and our study where real-world applications were used and all functions imports and uses external libraries and dependencies, a random file is used for simulating the different application sizes on the analysis performed by STeLLAR.

As demonstrated in this research, the impact of choosing a compiled or interpreted language runtime on the cold start is tremendous. The work on [59] came with one hypnotizes why Java functions have the worst initialization time of all three language runtimes tested. They claim it is because Java applications need more resource-intensive environments for starting the JVM, which overcharges the already busy CPU. This may explain why large Java functions performed badly with the *container-based deployment*. The authors on [59] also stated that this effect is smaller for higher memory settings. Our work is additional to this analysis since we demonstrate that medium-size Java applications when running with small memory configurations may present a lower cold start time when deployed as a container instead of a ZIP package.

3.5 Conclusion

Over the past five years, serverless computing has grown exponentially in popularity, and it is now one of the most used software architectures in the industry. However, not much focus was given to the underlying problems this model brings to light. The cold start problem is one of the biggest challenges that comes when using serverless functions. Additionally, the many choices around language runtime, memory configuration and deployment types one can have for deploying their workloads on the cloud make this problem even more complex. To address this issue, we presented the first extensive study about the impacts caused by the language runtime, memory allocation and function package size in the initialization time of AWS Lambda when using the two deployment types available, the *ZIP deployment* and the recently introduced *container-based deployment*.

We proposed guidelines for Python, Node.js and Java serverless functions under different memory configurations and application sizes according to which deployment type presents the best cold start performance. Developers can use these insights to achieve lower initialization times when deploying their applications on AWS Lambda by using application and platform knowledge.

Chapter 4

GreenLAC: Resource-Aware Dynamic Load Balancer for Serverless Edge Computing Platforms with AWS Greengrass

In this chapter, we present GreenLAC, a load balancer and reverse proxy for supporting serverless deployments of edge-core IoT applications with AWS Greengrass. GreenLAC enables applications running on resource-constrained edge nodes to leverage neighbours edge nodes and the core cloud for real-time processing of workloads. It monitors the host hardware and distributes requests according to predefined configurations to prevent failures at the edge. GreenLAC is completely integrated with one of the most popular edge runtimes and cloud services for IoT, the AWS Greengrass, and it supports the automatic deployment of serverless functions in the AWS core. We performed rigorous experiments on both AWS cloud and with a real-world application deployment using IoT devices and edge nodes. Preliminary results show that GreenLAC can increase the processing capabilities of hardware-restricted edge machines when using core resources in combination with serverless functions for IoT applications.

4.1 Introduction

In the past decade, we have witnessed the rapid growth of smart applications running at end-user devices and IoT gadgets. Latency-sensitive IoT applications require computing resources to be closer to customers and to end devices for real-time processing tasks. Edge, fog, and mobile edge computing [60], [61] address this extension of traditional cloud computing for this in-demand need. Edge computing brings computational power in close proximity to data sources and IoT devices. Edge devices can be used to reduce the data processing delay faced when using traditional cloud computing in the core. Moreover, IoT devices and gadgets such as Raspberry Pi[1] and BeagleBone[2] can be expanded to execute other computation tasks as well, enabling them to be processing edge nodes.

Public cloud providers such as AWS and Microsoft Azure now support edge computing and several IoT services at the edge. AWS was the pioneer to bring core cloud services to the edge with their service named AWS Greengrass which allows customers to execute important cloud services to on-premises edges. AWS Lambda functions and Docker containers - the same image containers used by ECS - can be deployed at the edge node and run offline or online. They can communicate with other services in the core, and users can deploy workloads at the edge using the AWS console. The same Lambda function can be deployed both at the edge and in the core, and complex serverless applications can be built using both the core and edge nodes [62]. However, local Lambdas running at the edge cannot be scaled automatically nor forward requests to the core under edge saturation. Microsoft Azure also supports executing core services at the edge using their Azure IoT Edge. The same Azure Function can be deployed at the edge and in the core simultaneously. Users can also deploy containers to the IoT edge and run complex serverless applications [63]. Even though many solutions for load distribution across the edge and core nodes

[1]https://www.raspberrypi.org
[2]https://beagleboard.org/bone

using AWS Greengrass have been proposed on [64]–[66], they often require complex architectures and implementations, and there is no open-source component for AWS Greengrass that supports dynamic load balancing of serverless applications running on edges and in the core simultaneously.

In this research, we introduce GreenLAC, an open-source component for AWS Greengrass that performs load balancing and resource monitoring on edge computing. GreenLAC is used as a reverse proxy at the edge nodes, and it is integrated with AWS Lambdas deployed both at the edges and in the core cloud. The proposed component has been validated by extensive experimentation on AWS and on-premises clouds using embedded architectures with a real-world IoT application. To the best of our knowledge, GreenLAC is the first open-source AWS Greengrass component for resource-aware dynamic load balancing of serverless applications.

4.2 GreenLAC

In this section, we explain the architecture and design of GreenLAC, and show how it uses the core resources for running serverless functions and for handling spikes in the workload. We evaluate GreenLAC on several architectures on both AWS and on a private cloud. The source code and deployment instructions are openly accessible on GitHub[3].

4.2.1 System Architecture

AWS Greengrass Lambda Autoscaler Core (GreenLAC) is a ready-to-use component for AWS Greengrass with minimal requirements and configurations needed. It can be used with any device or server running AWS Greengrass, and it supports load distribution to multiple edge nodes and core clouds. To the best of our knowledge, GreenLAC is the first open-source component for AWS Greengrass that allows load distributions among edge and core nodes with fallback and scaling features. It moni-

[3]https://github.com/pacslab/GreenLAC

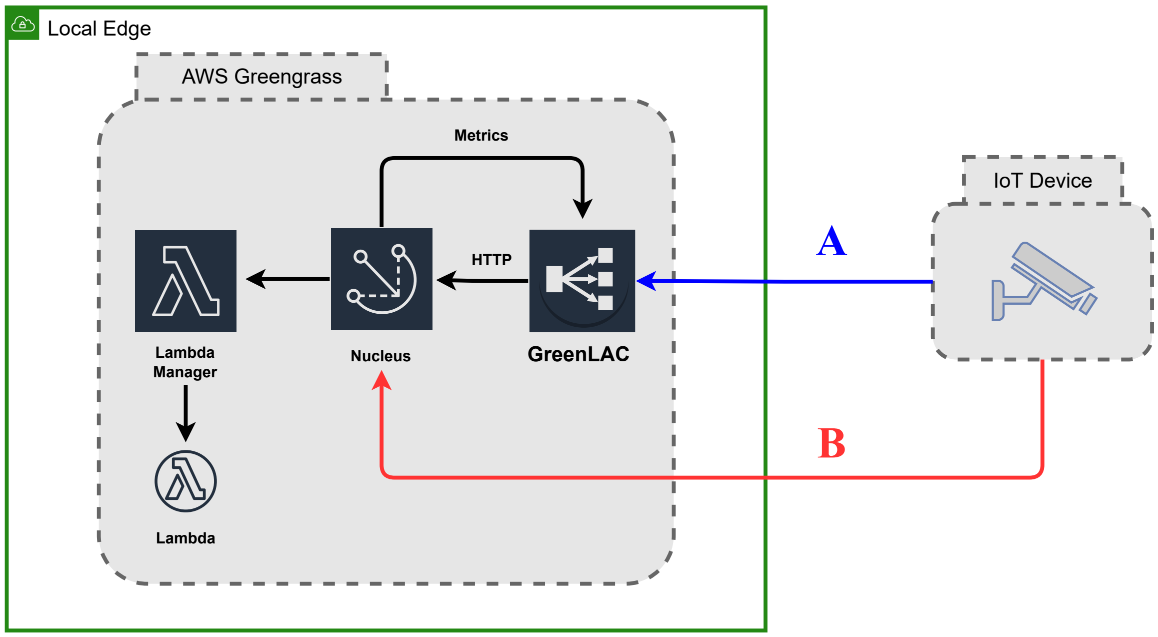

Figure 4.1: Deployment of the GreenLAC component on AWS Greengrass.

tors the hardware usage of the hosting OS to prevent resource starvation by the system and AWS Greengrass on the edge nodes. By fetching real-time CPU and memory usage, GreenLAC can distribute requests among edge and core nodes to preserve the SLAs and SLOs of the services. It was primarily designed to work with AWS Lambda functions running on AWS Greengrass at the edge or in the core. However, GreenLAC also supports microservice applications when using container deployments with the Docker component on AWS Greengrass. It was built using the Java programming language with the Spring Boot Framework[4].

Figure 4.1 shows the deployment of the GreenLAC component on AWS Greengrass inside a local edge node. This deployment can be performed using either the AWS Console through AWS IoT or by the automation script provided using the AWS CLI. GreenLAC works by intercepting HTTP requests sent by IoT devices to the edge node, and distributing them according to user rules and hardware constraints. It forwards requests to either the local edge node, remote edge nodes or the core according to the current state of the local edge node. The flow B of figure 4.1 shows the typical use

[4]https://spring.io/projects/spring-boot

case scenario of AWS Greengrass where an IoT device sends requests to the Nucleus component which then forwards them to the Lambda Manager. When we deploy the GreenLAC component, these requests are sent to GreenLAC instead as shown in flow A of figure 4.1. GreenLAC forwards requests to any available edge node or to the core.

In order to use multiple edge nodes and clouds for load distribution, developers are required to deploy their serverless applications, AWS Lambdas or container-based microservices, in the corresponding AWS IoT Groups and on AWS Lambda. Users can also configure AWS ECS in combination with AWS ECR to create clusters to be used with GreenLAC when using microservices.

Since the Lambda Manager component of AWS Greengrass is not open source, there is no feasible way to communicate with Lambda functions without using an interface for communication. GreenLAC requires that Lambda functions running on AWS Greengrass expose a TCP port for HTTP communication. The Greengrass SDK allows Lambda functions to communicate with other Lambdas using HTTP or MQTT communication, and it is available in all supported language runtimes. Additionally, AWS API Gateway needs to be configured to accept requests to Lambda functions in the core cloud.

To prevent failures and application crashes, GreenLAC monitors the CPU and memory utilization of the hosting device and redistributes requests whenever these two metrics reach a predefined threshold. Cloud engineers can use the GreenLAC component on each edge node of their cluster, and all the configuration is managed through the AWS Console. Figure 4.2 shows the deployment of GreenLAC in two edge nodes, one running AWS Greengrass inside an EC2 instance, and another one with a Raspberry Pi device. This arrangement allows the GreenLAC deployed on the Local Pi Edge to distribute the load that comes from the IoT devices connected to it (flow A) to three different sources: Local Pi Edge, remote EC2 edge (flow B) and AWS Core (flow C). On this configuration, the GreenLAC on the EC2 Edge will

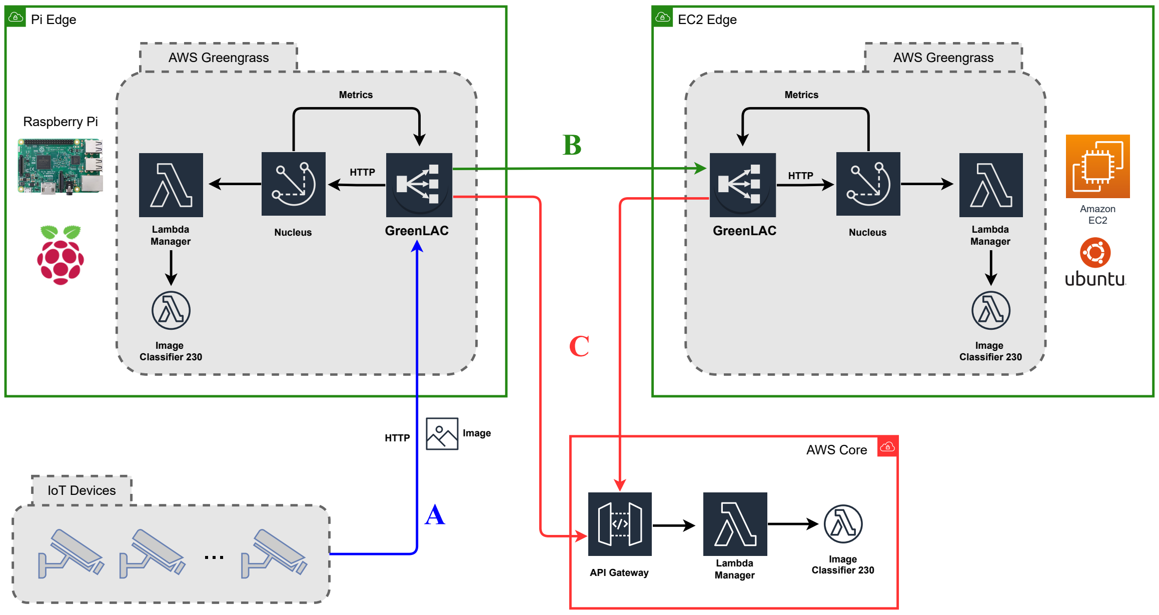

Figure 4.2: Deployment of the GreenLAC component on two edge nodes.

distribute its load across itself and the AWS Core (flow C).

The architecture of GreenLAC is composed of four modules. Figure 4.3 shows the architecture of GreenLAC. IoT devices send requests to the reverse proxy, and based on the load balancing policy and current OS metrics, the controller forwards the requests to either the edges nodes or the core. GreenLAC can be deployed on multiple edge nodes or on a single node.

Reverse Proxy: GreenLAC exposes all local endpoints that are running inside AWS Greengrass on its reverse proxy interface. It is a generic RESTful controller that accepts any payload and any HTTP method. The API path present in the request is used to forward it to the corresponding service. This interface is generic, and it does not require any customization when used with a new serverless function. The reverse proxy can reject requests whenever the service is saturated. This is because users can set the size limit of the internal buffer for concurrent requests.

Monitor: The monitoring interface is in charge of acquiring metrics of CPU and memory from the hosting operating system. The metrics are provided to the load balancer interface to be used when choosing which node the requests should be sent

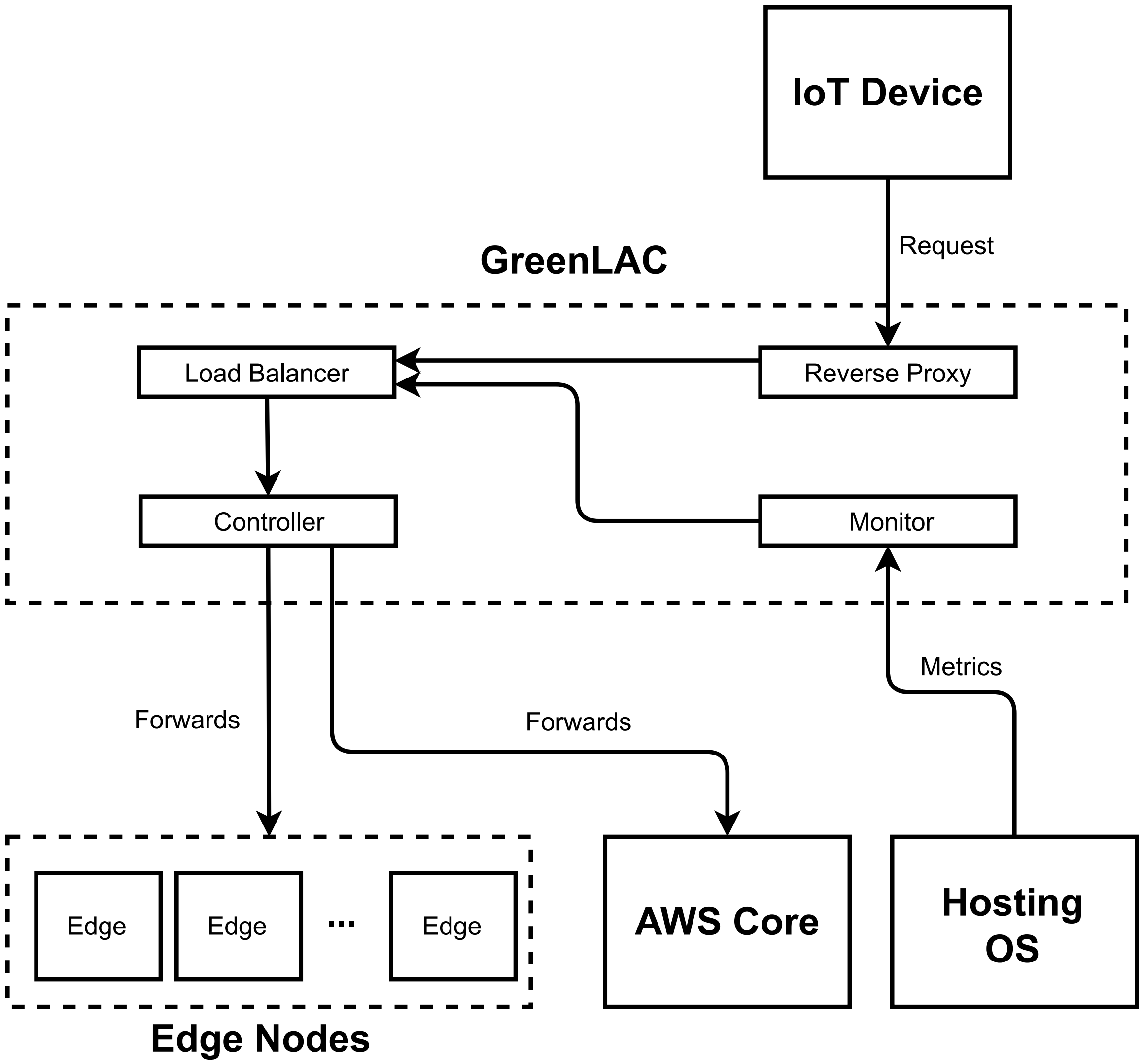

Figure 4.3: Architecture of the GreenLAC component.

to. By default, GreenLAC fetches the average utilization of CPU and memory each second, and it aggregates them by one-minute intervals. However, this configuration is changeable.

Load Balancer: The load balancing policy is applied, and the request is sent to the controller to be distributed accordingly.

Controller: The requests are distributed according to the predefined target node. The controller is also responsible for identifying and handling failures when sending requests to other neighbour nodes. It has a fallback service that redirects requests that are rejected by an edge node to the core or in case of an error, i.e., HTTP response code 5.x.x. All steps are logged, and users can check the real-time log with the provided AWS Greengrass log file.

4.2.2 Load Balancing Policies

GreenLAC has four load balancing policies to be used for edge computing deployments. Users can also implement customized policies using the generic interface provided. The algorithm 1 shows two of these policies. Requests can be forwarded to the core and edges based on the following policies:

Edge-Core-PC: The Edge-Core Priority Core policy is the default algorithm used by GreenLAC. As shown on Algorithm 1, requests are sent to the core whenever the local edge reaches its saturation point.

Edge-Core-PE: With The Edge-Core Priority Edge policy users can choose to use local and remote edges for distributing requests. GreenLAC will send requests to the multiple edges defined in the configuration file using the Round-robin scheduling algorithm whenever it needs to redistribute requests. In this strategy, the core is never used for processing requests, which guarantees the privacy of the processed data.

Core: All requests are forwarded to the core when using this policy.

Edge: All requests are forwarded to the local edge when using this policy.

4.3 Experimental Evaluation

We conducted two different experiments to evaluate the performance of GreenLAC for workload redistribution and resource management on edge-core architectures. In order to evaluate the performance of our component, we created a network of virtual IoT sensors using the Locust benchmark tool, and we analyzed the performance of our component with an IoT application running on a single edge node and on multiple edge nodes using an embedded system. The IoT application used is the Image Classifier 230 presented on section 3.2.4, and all the experiments are openly

Algorithm 1 Load balancer policies

```
Input: $cpu_{current}, mem_{current}, cpu_{\_MAX}, mem_{\_MAX}$
Input: $remoteEdgesList$
Output: node target
 1: if $cpu_{current} > cpu_{\_MAX}$ or $mem_{current} > mem_{\_MAX}$ then
 2:     $should\_scale \leftarrow True$;
 3: else
 4:     $should\_scale \leftarrow False$;
 5: end if
 6: if $should\_scale = True$ then
 7:     if $policy = Edge - Core - PC$ then
 8:         $target \leftarrow core$;
 9:     end if
10:     if $policy = Edge - Core - PE$ then
11:         $target \leftarrow roundRobin(remoteEdgesList)$;
12:     end if
13: else
14:     $target \leftarrow local\_edge$;
15: end if
16: return $target$
```

accessible on GitHub[5].

4.3.1 Single Edge Deployment

We deployed the GreenLAC component on a single edge node, and evaluated QoS metrics such as the 95th percentile and the average response time as well as the error rate.

Experimental Setup

We created the architecture shown in figure 4.1 which consists of the deployment of the Image Classifier 230 on one edge node and in the core cloud. The edge node is an EC2 instance with 4GB of memory and 2vCPUs running the Ubuntu OS created in the AWS region *ca-central-1.* We deployed AWS Greengrass Core at the edge node, and created one single AWS Lambda using the ZIP deployment of the Image Classifier 230 in the AWS Console. We then deployed it in the core in the AWS region *us-east-*

[5]https://github.com/pacslab/GreenLAC/tree/main/experiments

1, and at the edge using the AWS Greengrass Console. We invoke the Lambdas in the AWS core using the AWS API Gateway through a RESTful API. The Lambda concurrency limit of both the local edge and AWS core was set to 100. For the IoT sensors, we simulated a set of smart sensors that send HTTP requests with a JPEG image of dimensions 430×500 pixels for a 10-minute period. The workload used is a sequence of 3 consecutive linear increases and then a random pattern as demonstrated in figure 4.4(a). The load balancing policy was the *Edge-Core-PC*, and the buffer size was set to 5 concurrent requests. Finally, the CPU and memory threshold of the EC2 edge was set to 40%.

Experimental Results and Discussion

GreenLAC enabled the execution of the same AWS Lambda on both the edge node and the core cloud. Figure 4.4 shows the results of running the GreenLAC component on the EC2 edge. When we analyze the average and 95th percentile response time (figure 4.4(b)) of $t = 50s$, we can notice the effect of the cold start delay of the AWS Lambda on the first requests. Note that the CPU utilization of the EC2 edge (figure 4.4(c)) is kept at around 40% during all the time of the experiment. If we analyze the CPU utilization, response time and workload at $t = 120s$ and $t = 280s$, we can see that the workload increased around $2.4\times$ while the CPU utilization and 95th percentile response time was roughly the same at 39% and 330ms, respectively. GreenLAC was able to keep the same QoS metrics and hardware constraints because it sent part of the requests to be processed in the AWS core cloud.

The efficiency of GreenLAC is once again validated when we analyze the minimum and maximum CPU utilization and workload after $t = 105s$. These figures are 32.5% and 45.5% for the CPU utilization and 4 req/s and 12.6 req/s for the load. The average CPU utilization for this period was 40.9%, which demonstrates the efficacy of GreenLAC at keeping the desired hardware constraints at the edge. It achieved this by forwarding most of the requests to the AWS core. Figure 4.5 shows the number of

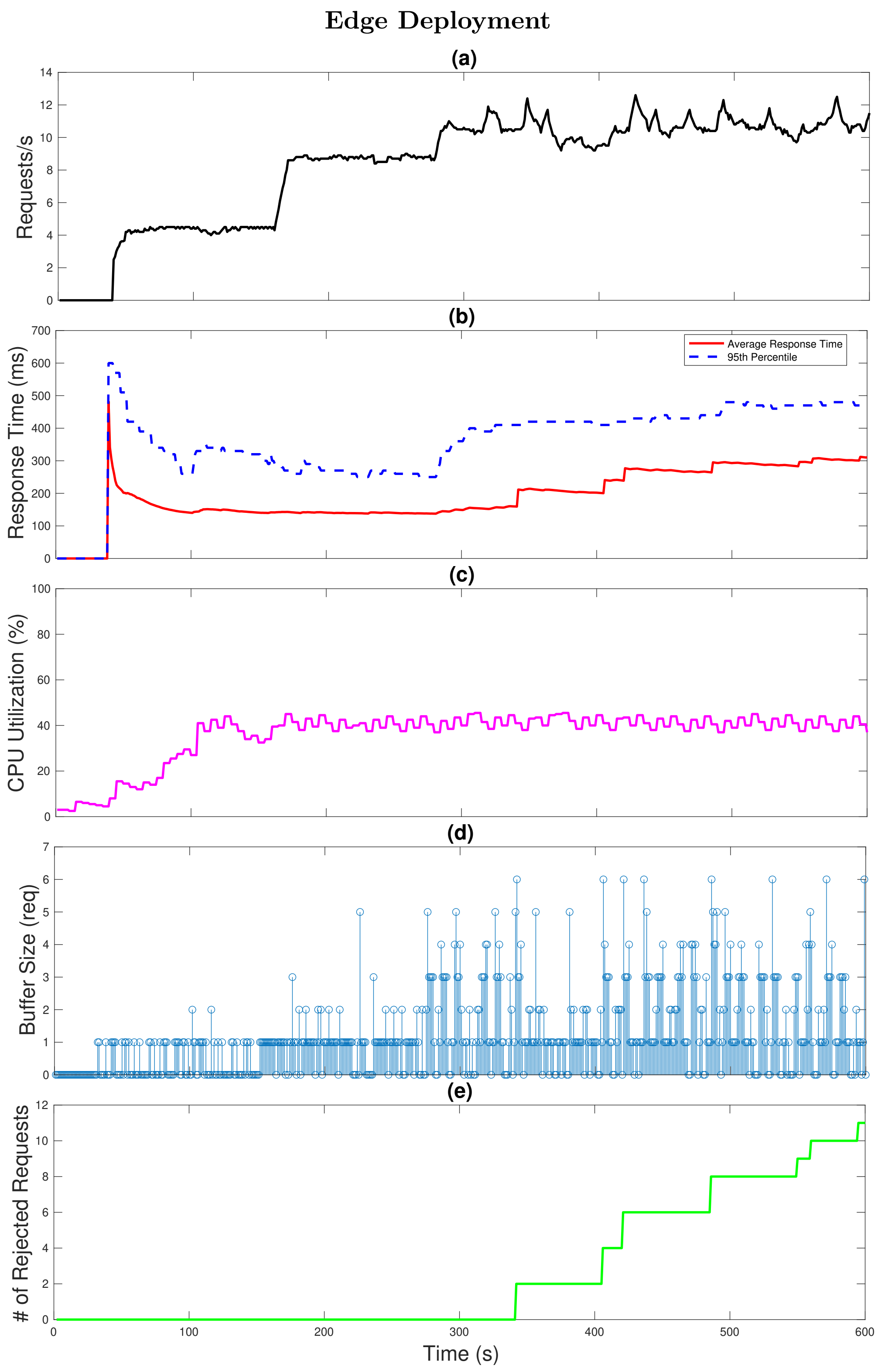

Figure 4.4: Experimental results for the single edge deployment.

requests processed. Note that approximately 57% of the total requests were sent to the core cloud. This allowed the extension of the edge processing capabilities without affecting the QoS metrics.

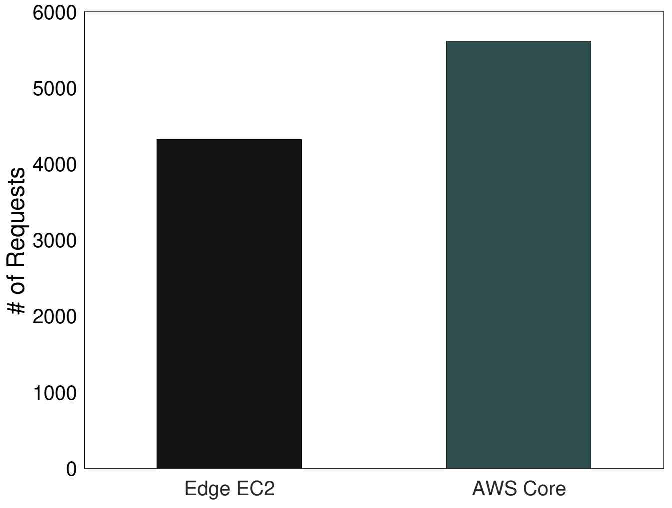

Figure 4.5: Request distribution for the single edge deployment.

Another important feature of GreenLAC is the concurrent request customization feature. Figure 4.4(d) shows the current buffer size, and figure 4.4(e) shows the number of requests that were rejected due to the maximum concurrent request constraints. Note that GreenLAC rejected requests to avoid resource starvation at the edge node. For this experiment, whenever the buffer size reaches more than 5 requests in the queue, GreenLAC starts to reject the following requests with the service unavailable error (HTTP code 503). In total, 11 requests were rejected and the maximum value of the buffer size was 6 requests. This demonstrated the importance of having controlling mechanisms to avoid unexpected server errors.

4.3.2 IoT Deployment

We used AWS Greengrass in combination with GreenLAC in an edge node running in the small single-board computer Raspberry Pi, and another remote edge node running on an EC2 instance.

Experimental Setup

We created the architecture shown in figure 4.2 which consists of the deployment of the Image Classifier 230 on two edge nodes and in the core cloud. The first edge node, named EC2 edge, is an EC2 instance with 4GB of memory and 2vCPUs running the Ubuntu OS in the AWS region *ca-central-1*. The second edge node, named Pi edge, is a Raspberry Pi 2 Model B with 1GB of memory and 4vCPUs running the Raspbian Buster OS hosted on a private cloud in Canada. We deployed the AWS Greengrass Core and the AWS Lambdas as explained in the previous experiment. The Lambda concurrency limit of both the edges and AWS core was also set to 100. For the IoT sensors, we simulated the same client configuration described in the previous experiment. The workload used, however, is a sequence of 2 consecutive linear increases and then a random pattern as demonstrated in figure 4.6(a). The buffer size was set to 100 concurrent requests. The load balancing policy of the Pi edge was the *Edge-Core-PE* while the EC2 edge was set to *Edge-Core-PC*. This means that the Pi Edge can send requests to the EC2 edge and to the core, whereas the EC2 edge can only send requests to the core. Finally, the CPU and memory threshold of the EC2 edge was set to 40% and the CPU utilization limit of the Pi edge was set to 20%. Due to the limited memory available in the Pi edge, the memory threshold was set to 90%.

Experimental Results and Discussion

Once again, GreenLAC was able to execute AWS Lambdas on both the edge nodes and the core cloud. Figure 4.6 shows the results of running the GreenLAC component on the EC2 and Pi Edges. Unlike the performance seen with the previous experiment, when we use an embedded system for edge-core deployments, the QoS metrics are heavily impacted due to the hardware limitations of the Pi edge. However, when we use GreenLAC to forward part of the load to a more robust edge node, we are able to increase the overall processing capabilities of the embedded hardware. This

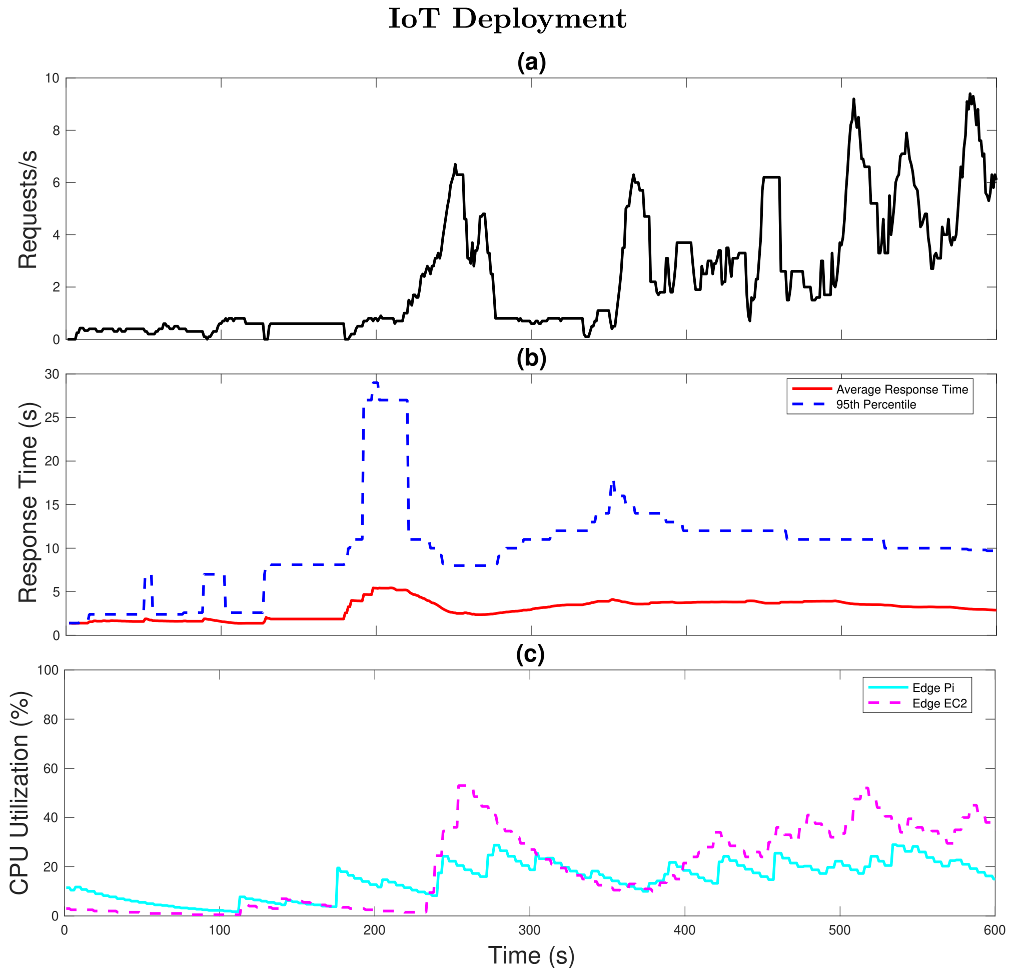

Figure 4.6: Experimental results for the IoT deployment.

is especially true when we analyze both edge nodes under their maximum load at $t = 583s$. This point is important because requests were being processed on all edges and the core cloud simultaneously. The arrival rate reached its peak of 9.4 req/s while the CPU utilization of the Pi and EC2 edge nodes were approximately 21% and 45%, respectively. This shows the efficacy of GreenLAC in keeping the desired hardware constraints even when the system is under maximum load. For this experiment, none of the requests were rejected and the buffer size never reached its maximum capacity.

We can see a better picture of the performance of GreenLAC when we analyze the number of requests processed by each node on figure 4.7. We can see that the majority of the requests sent to the Pi edge, approximately 72.5%, were processed in

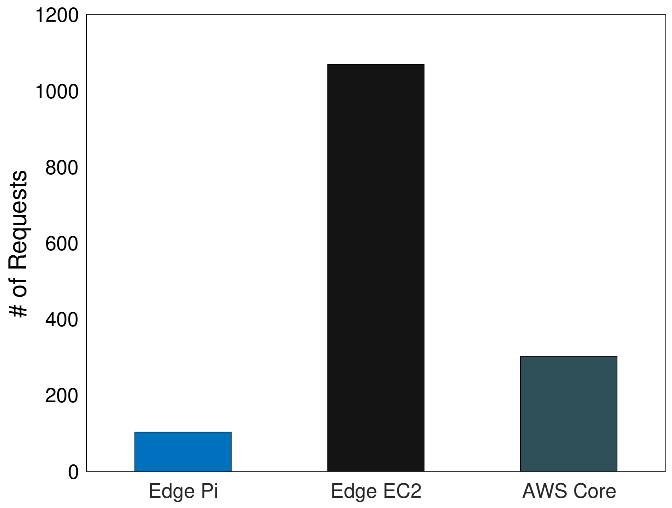

Figure 4.7: Request distribution for the IoT deployment.

the EC2 edge, followed by the AWS core with 20.5%, and lastly the Pi edge itself with only 7% of the requests. As a result of this dynamic request distribution, GreenLAC was able to keep the average CPU utilization of the Pi edge at 19.5% after $t = 300s$.

4.4 Related Work

Many studies have proposed serverless platforms for edge computing and IoT. In [67], the authors propose a new approach for edge computing named SEP by extending OpenWhisk — a state-of-the-art FaaS platform - for addressing latency-sensitive applications. Although the approach taken by the authors is innovative, it requires the use of Docker and other complex software such as Kafka, which makes it unsuitable for IoT-based edge computing environments. Apolo [68] and LaSS [69] are also serverless computing frameworks for edge computing which relay on Docker and OpenWhisk (for [69]). While they perform the orchestration of serverless functions, including AWS Lambdas, in the edge and core nodes, they are complex and require extra integration and control on the infrastructure. GreenLAC, on the other hand, is integrated with AWS Greengrass and it does not require any extra systems but the component itself which can be deployed using the AWS Console or CLI.

Recent studies investigated the use of serverless computing platforms for microcontrollers and IoT devices at the edge. μActor is proposed on [70] to address the computations challenges in the entire leaf-edge-cloud continuum. They use lightweight and long-lived stateful objects that communicate via message passing that can be executed by process virtual machines. Even though these agents can be used with IoT devices, they also need to be deployed in the core cloud and unlike GreenLAC, there is no out-of-the-box integration with AWS or AWS Lambda without the installation of extra components in the core cloud. Another edge computing architecture is proposed on [71] with the introduction of a completely new network architecture named Information-Centric Networking (ICN). The ICN strategy is to acquire hardware and network information for calculating the best interface to forward requests to the appropriate edge server achieve load balancing. Even though the authors on [71] validate their framework on a serverless application running on both robust edge nodes and Raspberry Pi nodes, ICN is a non-trivial network implementation that requires extra software and infrastructure resources to be implemented.

Some frameworks for deploying and scaling AWS Lambda at the edge with AWS Greengrass were proposed on [64]–[66]. CEVAS was proposed on [64], and it is a new serverless infrastructure paradigm for online video analysis orchestration of the cloud and edge resources based on the resource demand and cloud cost. While CEVAS and GreenLAC share the same orchestration mechanisms for deploying AWS Lambdas at the edge nodes and core cloud through AWS Greengrass, CEVAS' controller requires high CPU usage which poses a challenge when using it with microcontrollers or low processing power edge nodes. István et al. propose on [65] and [66] the automated deployment and dynamic reconfiguration of serverless functions at either the edge or cloud using AWS Greengrass. Although their framework holds a great promise for the future of edge serverless computing, it is not open source and there is no further documentation on how users can extend it to generic applications. GreenLAC, in contrast, is openly available and it works with both AWS Lambda and container-based microservices with Docker and ECS.

4.5 Conclusion

In this chapter, we presented and evaluated GreenLAC which is an AWS Greengrass component for redistributing load across multiple edge nodes and the core cloud according to customized load balancing policies. We analyzed and evaluated the deployment of a real-world IoT serverless application on both edge-core deployments. The proposed component can be used to extend the processing capabilities of hardware-restricted edges nodes to more robust edges nodes and the core cloud. It can also be deployed in combination with embedded systems such as Raspberry Pi and local edges running AWS Greengrass Core. Preliminary results show that GreenLAC enables IoT developers to deploy and use their serverless functions on multiple nodes simultaneously.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we addressed three main areas of cloud computing: scaling cloud systems, serverless computing and edge computing. More specifically, we focused on improving the performance, reducing the cost and mitigating some of the challenges present in cloud computing. As a result, we provide to the research community and the industry two open-source solutions as well as invaluable insights regarding serverless platforms.

First, in Chapter 2, we proposed an autoscaler that leverages burstable instances on the Google Cloud for scaling Google Compute Engines which achieved promising results by replacing some of the traditional instances with burstable instances. Previous studies have shown that this strategy is efficient in reducing the cost of cloud systems, and we presented the first open-source and fully-integrated autoscaler for Google Cloud.

Secondly, in Chapter 3, we addressed one of the biggest open challengers on serverless applications, the cold start delay. By using application and platform knowledge, we presented guidelines that developers and cloud managers can adopt to mitigate this issue. We investigated the impacts caused by the runtime language, memory allocation and function package size in the initialization time of AWS Lambda when using the two deployment types available nowadays.

Finally, in Chapter 4, we developed GreenLAC, which is an open-source component for AWS Greengrass that allows local edge applications to leverage cloud and remote edge resources for data processing. It enables load redistributing across multiple edge nodes and the core cloud according to customized load balancing policies. Preliminary results show promising potential for limited embedded edge nodes and latency-sensitive IoT applications.

5.2 Contributions

In this research, we aimed at providing tools and insights that can help improve the performance, cost, and efficiency of cloud computing platforms. The primary contributions of this work can be listed as follows:

- Design and development of BIAS Autoscaler, a highly-customizable autoscaler for Google Cloud.
- Provide in-depth analysis on the different deployment strategies of AWS Lambda.
- Design and development of GreenLAC, a dynamic load balancer for edge computing and IoT systems.

5.3 Future Work

The studies of this thesis open up several opportunities for future research.

- Make BIAS Autoscaler fully compatible with AWS using a customized load balancer. Also, proposing a performance modelling of burstable instances on GCP would be of utmost importance. Many models have been developed for the token-like system on AWS, but no study performed analytical modelling of burstable Google Compute Engines on Google Cloud.

- Launch an open-source benchmark framework to allow the automatic performance characterization of workloads for AWS Lambda using the data acquired from our analysis.
- Submit the source code of GreenLAC to the AWS Greengrass repository on GitHub to make this component available to the industry through the AWS Console.

Bibliography

[1] A. W. Services. (2020). "Hybrid cloud the most popular deployment path - study," [Online]. Available: https://datacenternews.asia/story/hybrid-cloud-the-most-popular-deployment-path-study (visited on 02/14/2022).

[2] M. Azure. (2022). "What is cloud computing?" [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/ (visited on 03/02/2022).

[3] G. Cloud. (2022). "Cloud functions," [Online]. Available: https://cloud.google.com/functions (visited on 02/14/2022).

[4] M. Azure. (2022). "Azure functions," [Online]. Available: https://azure.microsoft.com/en-us/services/functions/ (visited on 02/14/2022).

[5] A. W. Services. (2022). "Aws lambda," [Online]. Available: https://aws.amazon.com/lambda/ (visited on 02/14/2022).

[6] D. Prot. (2022). "Internet of things statistics for 2022 - taking things apart," [Online]. Available: https://dataprot.net/statistics/iot-statistics/ (visited on 02/14/2022).

[7] A. W. Services. (2021). "Burstable performance instances," [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html (visited on 04/18/2021).

[8] G. Cloud. (2021). "Shared-core machine types," [Online]. Available: https://cloud.google.com/compute/docs/machine-types#sharedcore (visited on 04/18/2021).

[9] M. Azure. (2021). "B-series burstable virtual machine sizes," [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable (visited on 04/18/2021).

[10] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, pp. 153–167, ISBN: 9781450350853. DOI: 10.1145/3132747.3132772. [Online]. Available: https://doi.org/10.1145/3132747.3132772.

[11] M. Dabbagh, B. Hamdaoui, M. Guizani, and A. Rayes, "Toward energy-efficient cloud computing: Prediction, consolidation, and overcommitment," *IEEE Network*, vol. 29, no. 2, pp. 56–61, 2015. DOI: 10.1109/MNET.2015.7064904.

[12] A. W. Services. (2021). "Amazon ec2 on-demand pricing," [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls (visited on 04/18/2021).

[13] G. Cloud. (2021). "Compute engine on-demand pricing," [Online]. Available: https://cloud.google.com/compute/all-pricing#sharedcore (visited on 04/18/2021).

[14] ——, (2021). "Cloud load balancing," [Online]. Available: https://cloud.google.com/load-balancing (visited on 03/14/2021).

[15] A. F. Baarzi, T. Zhu, and B. Urgaonkar, "Burscale: Using burstable instances for cost-effective autoscaling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19, Santa Cruz, CA, USA: Association for Computing Machinery, 2019, pp. 126–138, ISBN: 9781450369732. DOI: 10.1145/3357223.3362706. [Online]. Available: https://doi.org/10.1145/3357223.3362706.

[16] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Cedule: A scheduling framework for burstable performance in cloud computing," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*, 2018, pp. 141–150. DOI: 10.1109/ICAC.2018.00024.

[17] G. Cloud. (2021). "Google compute engine," [Online]. Available: https://cloud.google.com/compute (visited on 04/12/2021).

[18] ——, (2021). "Google kubernetes engine," [Online]. Available: https://cloud.google.com/kubernetes-engine (visited on 04/15/2021).

[19] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Distributed, robust auto-scaling policies for power management in compute intensive server farms," in *2011 Sixth Open Cirrus Summit*, 2011, pp. 1–5. DOI: 10.1109/OCS.2011.6.

[20] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "Ensure: Efficient scheduling and autonomous resource management in serverless environments," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, pp. 1–10. DOI: 10.1109/ACSOS49614.2020.00020.

[21] H. .-. Lin and C. S. Raghavendra, "An analysis of the join the shortest queue (jsq) policy," in *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*, 1992, pp. 362–366. DOI: 10.1109/ICDCS.1992.235020.

[22] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action.* Cambridge University Press, 2013.

[23] Y. Jiang, M. Shahrad, D. Wentzlaff, D. H. K. Tsang, and C. Joe-Wong, "Burstable instances for clouds: Performance modeling, equilibrium analysis, and revenue maximization," *IEEE/ACM Transactions on Networking*, vol. 28, no. 6, pp. 2489–2502, 2020. DOI: 10.1109/TNET.2020.3015523.

[24] R. Pinciroli, A. Ali, F. Yan, and E. Smirni, "Cedule+: Resource management for burstable cloud instances using predictive analytics," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 945–957, 2021. DOI: 10.1109/TNSM.2020.3039942.

[25] V. Podolskiy, A. Jindal, and M. Gerndt, "Iaas reactive autoscaling performance challenges," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 954–957. DOI: 10.1109/CLOUD.2018.00144.

[26] M. N. A. H. Khan, Y. Liu, H. Alipour, and S. Singh, "Modeling the autoscaling operations in cloud with time series data," in *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW)*, 2015, pp. 7–12. DOI: 10.1109/SRDSW.2015.20.

[27] A. Abdel Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1411–1415. DOI: 10.1109/CSCI49370.2019.00264.

[28] A. A. Khaleq and I. Ra, "Intelligent autoscaling of microservices in the cloud for real-time applications," *IEEE Access*, vol. 9, pp. 35 464–35 476, 2021. DOI: 10.1109/ACCESS.2021.3061890.

[29] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis, "Using burstable instances in the public cloud: Why, when and how?" *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, Jun. 2017. DOI: 10.1145/3084448. [Online]. Available: https://doi.org/10.1145/3084448.

[30] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "It's not a sprint, it's a marathon: Stretching multi-resource burstable performance in public clouds (industry track)," in *Proceedings of the 20th International Middleware Conference Industrial Track*, ser. Middleware '19, Davis, CA, USA: Association for Computing Machinery, 2019, pp. 36–42, ISBN: 9781450370417. DOI: 10.1145/3366626.3368130. [Online]. Available: https://doi.org/10.1145/3366626.3368130.

[31] I. Cloud. (2021). "Iaas vs. paas vs. saas," [Online]. Available: https://www.ibm.com/cloud/learn/iaas-paas-saas (visited on 02/15/2022).

[32] ——, (2022). "Ibm cloud functions," [Online]. Available: https://cloud.ibm.com/functions/ (visited on 03/24/2022).

[33] A. W. Services. (2020). "Aws lambda - container image support," [Online]. Available: https://aws.amazon.com/blogs/aws/new-for-aws-lambda-container-image-support (visited on 02/14/2022).

[34] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188. DOI: 10.1109/UCC-Companion.2018.00054.

[35] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean lambdas with large libraries," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, IEEE, 2017, pp. 395–400.

[36] H. Puripunpinyo and M. Samadzadeh, "Effect of optimizing java deployment artifacts on aws lambda," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2017, pp. 438–443.

[37] I. Stancin and A. Jovic, "An overview and comparison of free python libraries for data mining and big data analysis," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 977–982. DOI: 10.23919/MIPRO.2019.8757088.

[38] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 30–44, ISBN: 9781450381376. DOI: 10.1145/3419111.3421280. [Online]. Available: https://doi.org/10.1145/3419111.3421280.

[39] P. Vahidinia, B. Farahani, and F. S. Aliee, "Cold start in serverless computing: Current trends and mitigation strategies," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–7. DOI: 10.1109/COINS49042.2020.9191377.

[40] Denodo. (2020). "Denodo global cloud survey 2020," [Online]. Available: https://www.denodo.com/en/document/whitepaper/denodo-global-cloud-survey-2020 (visited on 03/02/2022).

[41] Datadog. (2021). "The state of serverless," [Online]. Available: https://www.datadoghq.com/state-of-serverless/ (visited on 01/21/2022).

[42] E. Şamdan. (2018). "Dealing with cold starts in aws lambda," [Online]. Available: https://medium.com/thundra/dealing-with-cold-starts-in-aws-lambda-a5e3aa8f532 (visited on 01/10/2022).

[43] E. Samdan. (2017). "A cloud guru news," [Online]. Available: https://acloudguru.com/blog/engineering/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda (visited on 01/10/2022).

[44] A. W. Services. (2018). "Become a serverless black belt - optimizing your serverless applications," [Online]. Available: https://pages.awscloud.com/Become-a-Serverless-Black-Belt---Optimizing-Your-Serverless-Applications%5C_0205-SR%5C_OD.html (visited on 02/14/2022).

[45] D. Ustiugov, T. Amariucai, and B. Grot, "Analyzing tail latency in serverless clouds with stellar," English, in *2021 IEEE International Symposium on Workload Characterization (IISWC'21)*, 2021 IEEE International Symposium on Workload Characterization, IISWC 2021 ; Conference date: 07-11-2021 Through 09-11-2021, United States: Institute of Electrical and Electronics Engineers (IEEE), Sep. 2021. [Online]. Available: http://www.iiswc.org/iiswc2021/index.html.

[46] S. Raschka and V. Mirjalili, "Python machine learning: Machine learning and deep learning with python," *Scikit-Learn, and TensorFlow. Second edition ed*, 2017.

[47] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, *et al.*, "Scipy 1.0: Fundamental algorithms for scientific computing in python," *Nature methods*, vol. 17, no. 3, pp. 261–272, 2020. DOI: https://doi.org/10.1038/s41592-019-0686-2.

[48] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *2018 Usenix Annual Technical Conference USENIX ATC 18)*, 2018, pp. 923–935.

[49] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 467–481, ISBN: 9781450371025. [Online]. Available: https://doi.org/10.1145/3373376.3378512.

[50] K. Solaiman and M. A. Adnan, "Wlec: A not so cold architecture to mitigate cold start problem in serverless computing," in *2020 IEEE International Conference on Cloud Engineering (IC2E)*, 2020, pp. 144–153. DOI: 10.1109/IC2E48712.2020.00022.

[51] W. Ling, L. Ma, C. Tian, and Z. Hu, "Pigeon: A dynamic and efficient serverless and faas framework for private cloud," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1416–1421. DOI: 10.1109/CSCI49370.2019.00265.

[52] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in faas services," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 134–143, ISBN: 9781450368667. [Online]. Available: https://doi.org/10.1145/3341105.3373909.

[53] D. Kelly, F. G. Glavin, and E. Barrett, *Serverless computing: Behind the scenes of major platforms*, 2020. arXiv: 2012.05600 `[cs.DC]`.

[54] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, and S. Lanka, "Sequoia: Enabling quality-of-service in serverless computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 311–327, ISBN: 9781450381376. DOI: 10.1145/3419111.3421306. [Online]. Available: https://doi.org/10.1145/3419111.3421306.

[55] S. Horovitz, R. Amos, O. Baruch, T. Cohen, T. Oyar, and A. Deri, "Faastest - machine learning based cost and performance faas optimization: 15th international conference, gecon 2018, pisa, italy, september 18–20, 2018, proceedings," in. Jan. 2019, pp. 171–186, ISBN: 978-3-030-13341-2. DOI: 10.1007/978-3-030-13342-9_15.

[56] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 30–44, ISBN: 9781450381376. DOI: 10.1145/3419111.3421280. [Online]. Available: https://doi.org/10.1145/3419111.3421280.

[57] J. Jarachanthan, L. Chen, F. Xu, and B. Li, "Amps-inf: Automatic model partitioning for serverless inference with cost efficiency," in *50th International Conference on Parallel Processing*. New York, NY, USA: Association for Computing Machinery, 2021, ISBN: 9781450390682. [Online]. Available: https://doi.org/10.1145/3472456.3472501.

[58] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?" *IEEE Software*, vol. 38, no. 1, pp. 32–39, 2021. DOI: 10.1109/MS.2020.3023302.

[59] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 181–188.

[60] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for internet of things realization," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 2961–2991, 2018.

[61] J. Ren, D. Zhang, S. He, Y. Zhang, and T. Li, "A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet," *ACM Computing Surveys (CSUR)*, vol. 52, no. 6, pp. 1–36, 2019.

[62] A. W. Services. (2022). "Aws iot greengrass," [Online]. Available: https://aws.amazon.com/greengrass/ (visited on 02/28/2022).

[63] M. Azure. (2019). "Iot on the edge and in the cloud.," [Online]. Available: https://devintxcontent.blob.core.windows.net/showcontent/Speaker%5C%20Presentations%5C%20Fall%5C%202019/AzureAIConf_2019%5C%20-%5C%20IoT%5C%20Edge.pdf (visited on 02/28/2022).

[64] M. Zhang, F. Wang, Y. Zhu, J. Liu, and Z. Wang, "Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines," in *Proceedings of the 12th ACM Multimedia Systems Conference*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 80–93, ISBN: 9781450384346. [Online]. Available: https://doi.org/10.1145/3458305.3463377.

[65] I. Pelle, F. Paolucci, B. Sonkoly, and F. Cugini, "Latency-sensitive edge/cloud serverless dynamic deployment over telemetry-based packet-optical network," *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 9, pp. 2849–2863, 2021. DOI: 10.1109/JSAC.2021.3064655.

[66] I. Pelle, J. Czentye, J. Dóka, A. Kern, B. P. Gerő, and B. Sonkoly, "Operating latency sensitive applications on public serverless edge cloud platforms," *IEEE Internet of Things Journal*, vol. 8, no. 10, pp. 7954–7972, 2021. DOI: 10.1109/JIOT.2020.3042428.

[67] L. Baresi and D. Filgueira Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*, 2019, pp. 1–10. DOI: 10.1109/ICFC.2019.00008.

[68] F. Smirnov, C. Engelhardt, J. Mittelberger, B. Pourmohseni, and T. Fahringer, "Apollo: Towards an efficient distributed orchestration of serverless function compositions in the cloud-edge continuum," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC '21, Leicester, United Kingdom: Association for Computing Machinery, 2021, ISBN: 9781450385640. DOI: 10.1145/3468737.3494103. [Online]. Available: https://doi.org/10.1145/3468737.3494103.

[69] B. Wang, A. Ali-Eldin, and P. Shenoy, "Lass: Running latency sensitive serverless computations at the edge," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21, Virtual Event, Sweden: Association for Computing Machinery, 2021, pp. 239–251, ISBN: 9781450382175. DOI: 10.1145/3431379.3460646. [Online]. Available: https://doi.org/10.1145/3431379.3460646.

[70] R. Hetzel, T. Kärkkäinen, and J. Ott, "Actor: Stateful serverless at the edge," in *Proceedings of the 1st Workshop on Serverless Mobile Networking for 6G Communications*, ser. MobileServerless'21, Virtual, WI, USA: Association for Computing Machinery, 2021, pp. 1–6, ISBN: 9781450386036. DOI: 10.1145/3469263.3470828. [Online]. Available: https://doi.org/10.1145/3469263.3470828.

[71] Z. Fan, W. Yang, F. Wu, J. Cao, and W. Shi, "Serving at the edge: An edge computing service architecture based on icn," *ACM Trans. Internet Technol.*, vol. 22, no. 1, Oct. 2021, ISSN: 1533-5399. DOI: 10.1145/3464428. [Online]. Available: https://doi.org/10.1145/3464428.