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

Jaime Dantas, Hamzeh Khazaei and Marin Litoiu
{jaimecjd,hkh,mlitoiu}@yorku.ca
Department of Electrical Engineering & Computer Science
York University
Toronto, Ontario, Canada

ABSTRACT

Edge computing is a distributed computing paradigm that brings computation resources closer to the sources of data. This recently-introduced model enables low latency applications to access computing and storage services deployed on-premises. In this paper, we present GreenLAC, a load balancer and reverse proxy for supporting serverless deployments of edge-core IoT applications. 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. We performed rigorous experiments on 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.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*.

KEYWORDS

AWS Greengrass, Edge Computing, Function as a Service, Serverless Computing, IoT

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 [8, 10] 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, IBM and Microsoft Azure now support edge computing and several IoT services at the edge. As FaaS grew in popularity, cloud providers have started to develop their own edge platforms to support serverless functions and other cloud services, and AWS became 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 [11]. 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 [1]. Finally, IBM Edge helps solve speed and scale challenges by using the computational capacity of edge devices, gateways, and networks [5]. Even though many solutions for load distribution across the edge and core nodes using AWS Greengrass have been proposed on [6, 7, 14], 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 paper, 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 deployed both at the edges and in the core cloud. The contributions of this position paper are:

- an architecture and an open-source implementation of an edge-core load balancer for serverless applications
- validation of the serverless balancer feasibility and integration and performance with a public cloud platform

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

[1] https://www.raspberrypi.org

[2] https://beagleboard.org/bone

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CASCON'22, November 15–17, 2022, Toronto, Canada
© 2022 Copyright held by the owner/author(s).

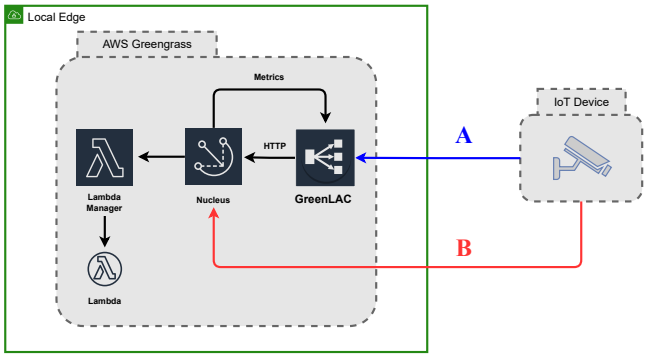

Figure 1: Deployment of GreenLAC component on AWS Greengrass.

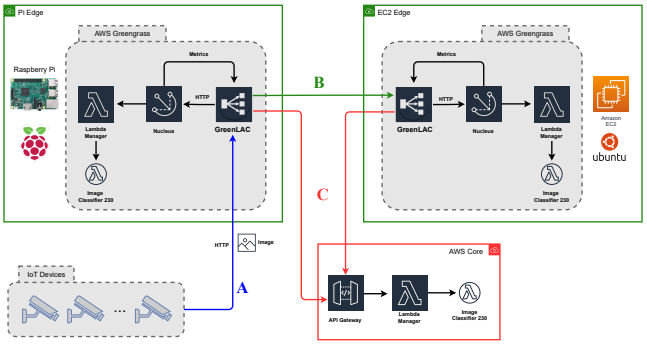

Figure 2: Deployment of GreenLAC component on two edge nodes.

component for resource-aware dynamic load balancing of serverless applications.

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].

2.1 GreenLac Architecture and Intended Use

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 monitors 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 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 1 shows the typical use 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 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 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 distribute its load across itself and the AWS Core (flow C).

The architecture of GreenLAC is composed of four modules. Figure 3 shows the architecture of GreenLAC. IoT devices send

[3] https://github.com/pacslab/GreenLAC
[4] https://spring.io/projects/spring-boot

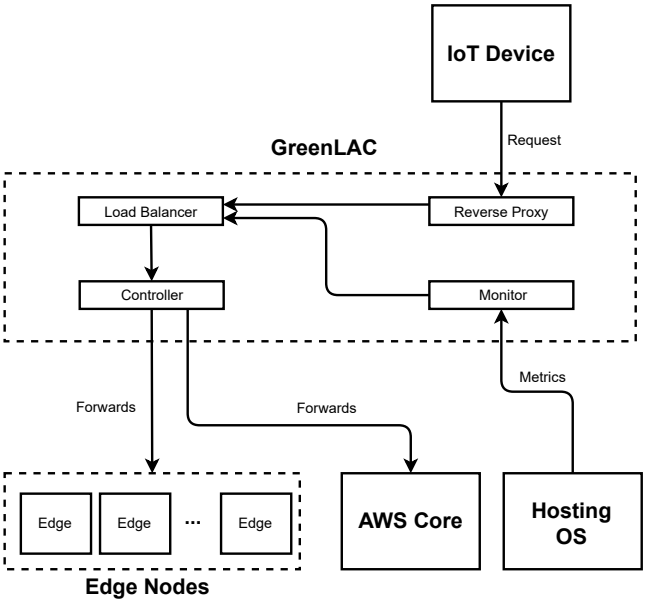

Figure 3: Architecture of the GreenLAC component.

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 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.

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.

Algorithm 1 Load balancer policies

Input: $cpu_{current}, mem_{current}, cpu\_MAX, mem\_MAX$
Input: $remoteEdgesList$
Output: node target

if $cpu_{current} > cpu\_MAX$ *or* $mem_{current} > mem\_MAX$ **then**
  $should\_scale \leftarrow True$;
else
  $should\_scale \leftarrow False$;
end if
if $should\_scale = True$ **then**
  if $policy = Edge - Core - PC$ **then**
    $target \leftarrow core$;
  end if
  if $policy = Edge - Core - PE$ **then**
    $target \leftarrow roundRobin(remoteEdgesList)$;
  end if
else
  $target \leftarrow local\_edge$;
end if
return $target$

3 PRELIMINARY 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 which

is available on GitHub[5]. It consists of a binary image classification function using one of the most popular and comprehensive open-source machine learning libraries, the *scikit-learn (sklearn)* [9]. 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. All the experiments are openly accessible on GitHub[6].

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.

3.1.1 Experimental Setup. We created the architecture shown in Figure 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-1*, and at the core 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(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%.

3.1.2 Experimental Results and Discussion. GreenLAC enabled the execution of the same AWS Lambda on both the edge node and the core cloud. Figure 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(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(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× 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

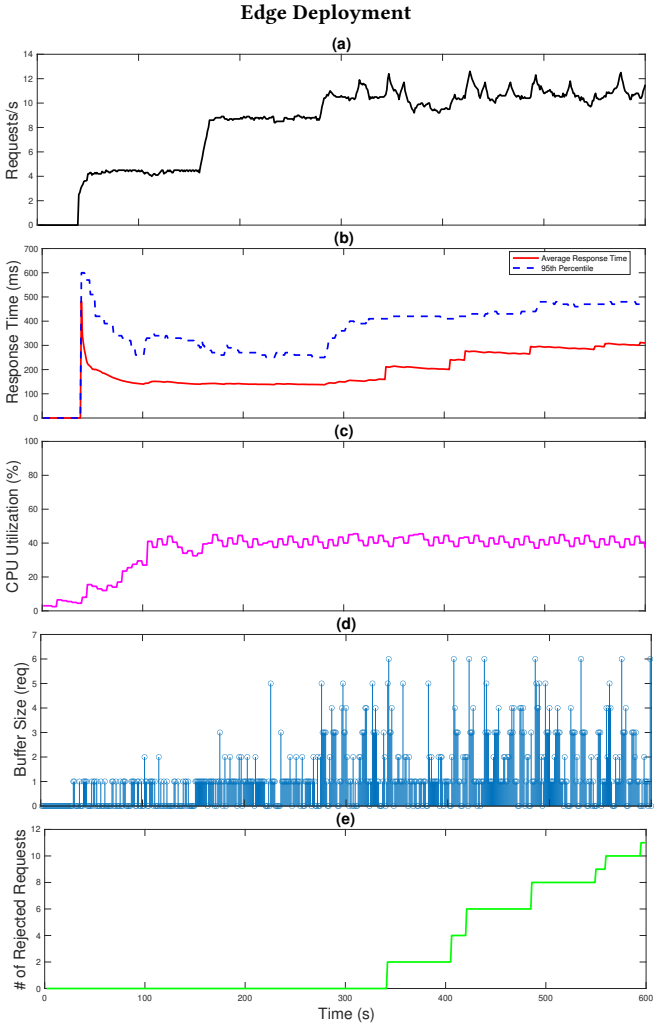

Figure 4: Experimental results for the single edge deployment.

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 5 shows the number of 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.

Another important feature of GreenLAC is the concurrent request customization feature. Figure 4(d) shows the current buffer size, and Figure 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.

[5] https://github.com/pacslab/serverless-iot-deployment/tree/main/lambda/image-classifier-230

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

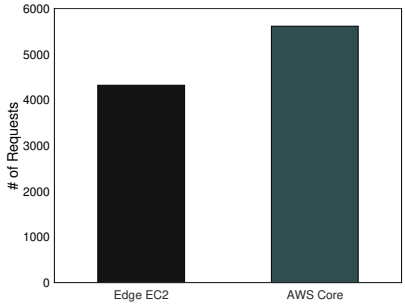

Figure 5: Request distribution for the edge deployment.

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.

3.2.1 Experimental Setup. We created the architecture shown in Figure 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 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%.

3.2.2 Experimental Results and Discussion. Once again, GreenLAC was able to execute AWS Lambdas on both the edge nodes and the core cloud. Figure 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 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

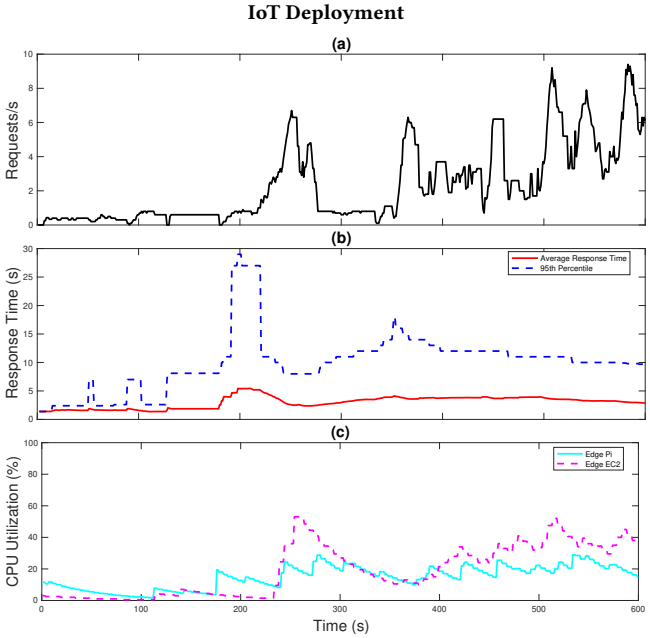

Figure 6: Experimental results for the IoT deployment.

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.

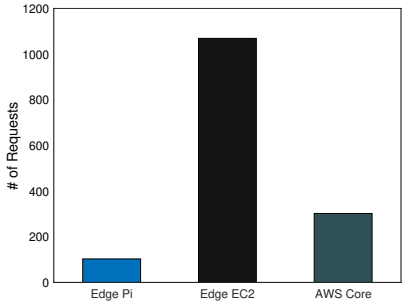

Figure 7: Request distribution for the IoT deployment.

We can see a better picture of the performance of GreenLAC when we analyze the number of requests processed by each node on Figure 7. We can see that the majority of the requests sent to the Pi edge, approximately 72.5%, were processed in 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 RELATED WORK

Many studies have proposed serverless platforms for edge computing and IoT. In [2], 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 [12] and LaSS [13] are also serverless computing frameworks for edge computing which relay on Docker and OpenWhisk (for [13]). 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 [4] 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 [3] 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 [3] 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 core with AWS Greengrass were proposed on [6, 7, 14]. CEVAS was proposed on [14], 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 [7] and [6] 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.

5 CONCLUSION AND FUTURE WORK

In this paper, 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.

In future work, we plan to deploy GreenLAC on multiple edge nodes and evaluate the performance cost and trade-offs when using this AWS Greengrass component on embedded systems. Enabling dynamic load balancing policies through the use of models at runtime is also a promising direction. Furthermore, we intend to integrate GreenLAC to the AWS Greengrass repository on GitHub to make this component available to the industry through the AWS Console.

REFERENCES

[1] Microsoft Azure. 2019. *IoT on the Edge and in the Cloud.* shorturl.at/wASU7

[2] Luciano Baresi and Danilo Filgueira Mendonça. 2019. Towards a Serverless Platform for Edge Computing. In *2019 IEEE International Conference on Fog Computing (ICFC).* 1–10. https://doi.org/10.1109/ICFC.2019.00008

[3] Zhenyu Fan, Wang Yang, Fan Wu, Jing Cao, and Weisong Shi. 2021. Serving at the Edge: An Edge Computing Service Architecture Based on ICN. *ACM Trans. Internet Technol.* 22, 1, Article 22 (oct 2021), 27 pages. https://doi.org/10.1145/3464428

[4] Raphael Hetzel, Teemu Kärkkäinen, and Jörg Ott. 2021. Actor: Stateful Serverless at the Edge. In *Proceedings of the 1st Workshop on Serverless Mobile Networking for 6G Communications* (Virtual, WI, USA) *(MobileServerless'21).* Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3469263.3470828

[5] IBM. 2021. *Overview of Edge computing.* https://www.ibm.com/docs/en/eam/3.2.1?topic=overview-edge-computing

[6] István Pelle, János Czentye, János Dóka, András Kern, Balázs P. Gerő, and Balázs Sonkoly. 2021. Operating Latency Sensitive Applications on Public Serverless Edge Cloud Platforms. *IEEE Internet of Things Journal* 8, 10 (2021), 7954–7972. https://doi.org/10.1109/JIOT.2020.3042428

[7] István Pelle, Francesco Paolucci, Balázs Sonkoly, and Filippo Cugini. 2021. Latency-Sensitive Edge/Cloud Serverless Dynamic Deployment Over Telemetry-Based Packet-Optical Network. *IEEE Journal on Selected Areas in Communications* 39, 9 (2021), 2849–2863. https://doi.org/10.1109/JSAC.2021.3064655

[8] Pawani Porambage, Jude Okwuibe, Madhusanka Liyanage, Mika Ylianttila, and Tarik Taleb. 2018. Survey on multi-access edge computing for internet of things realization. *IEEE Communications Surveys & Tutorials* 20, 4 (2018), 2961–2991.

[9] Sebastian Raschka and Vahid Mirjalili. 2017. Python Machine Learning: Machine Learning and Deep Learning with Python. *Scikit-Learn, and TensorFlow. Second edition ed* (2017).

[10] Ju Ren, Deyu Zhang, Shiwen He, Yaoxue Zhang, and Tao Li. 2019. A survey on end-edge-cloud orchestrated network computing paradigms: Transparent computing, mobile edge computing, fog computing, and cloudlet. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–36.

[11] Amazon Web Services. 2022. *AWS IoT Greengrass.* https://aws.amazon.com/greengrass/

[12] Fedor Smirnov, Chris Engelhardt, Jakob Mittelberger, Behnaz Pourmohseni, and Thomas Fahringer. 2021. 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* (Leicester, United Kingdom) *(UCC '21).* Association for Computing Machinery, Article 10, 10 pages. https://doi.org/10.1145/3468737.3494103

[13] Bin Wang, Ahmed Ali-Eldin, and Prashant Shenoy. 2021. LaSS: Running Latency Sensitive Serverless Computations at the Edge. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (Virtual Event, Sweden) *(HPDC '21).* Association for Computing Machinery, 239–251. https://doi.org/10.1145/3431379.3460646

[14] Miao Zhang, Fangxin Wang, Yifei Zhu, Jiangchuan Liu, and Zhi Wang. 2021. *Towards Cloud-Edge Collaborative Online Video Analytics with Fine-Grained Serverless Pipelines.* Association for Computing Machinery, New York, NY, USA, 80–93. https://doi.org/10.1145/3458305.3463377