Serverless Platforms on the Edge: A Performance Analysis

Hamza Javed, Adel N. Toosi[0000−0001−5655−5337], Mohammad S. Aslanpour[0000−0002−1816−6901]

Faculty of Information Technology
Monash University, Clayton, Australia
{hamza.javed, adel.n.toosi, mohammad.aslanpour}@monash.edu

Abstract. The exponential growth of Internet of Things (IoT) has given rise to a new wave of edge computing due to the need to process data on the edge, closer to where it is being produced and attempting to move away from a cloud-centric architecture. This provides its own opportunity to decrease latency and address data privacy concerns along with the ability to reduce public cloud costs. The serverless computing model provides a potential solution with its event-driven architecture to reduce the need for ever-running servers and convert the backend services to an as-used model. This model is an attractive prospect in edge computing environments with varying workloads and limited resources. Furthermore, its setup on the edge of the network promises reduced latency to the edge devices communicating with it and eliminates the need to manage the underlying infrastructure. In this book chapter, first, we introduce the novel concept of serverless edge computing, then, we analyze the performance of multiple serverless platforms, namely, OpenFaaS, AWS Greengrass, Apache OpenWhisk, when set up on the single-board computers (SBCs) on the edge and compare it with public cloud serverless offerings, namely, AWS Lambda and Azure Functions, to deduce the suitability of serverless architectures on the network edge. These serverless platforms are set up on a cluster of Raspberry Pis and we evaluate their performance by simulating different types of edge workloads. The evaluation results show that OpenFaaS achieves the lowest response time on the SBC edge computing infrastructure while serverless cloud offerings are the most reliable with the highest success rate.

Keywords: IoT · OpenFaaS · Apache OpenWhisk · AWS Lambda · Azure Functions · AWS Greengrass · Serverless · FaaS · Serverless Edge Computing

1 Introduction

In recent years, we have seen a major increase in new developments in cloud computing due the requirement of handling data at a massive scale. New technologies such as Internet of Things (IoT) and edge devices such as smartphones have created a large influx of data and network traffic that has to be processed

at scale. [27][33]. This has given rise to new technologies and computational architectures being introduced into cloud computing in order to maintain the Quality of Service (QoS) of the applications providing these services. In the past few years, deployment architectures have changed from monolithic [4] to microservice architectures that aid the developers to build large applications in a scalable manner. Microservice architectures demanded the need of Container-as-a-Service (CaaS) [13] that provided always-on servers with the ability to scale out easily. However, serverless computing provides a different approach to managing servers in the form of Function-as-a-Service (FaaS) [7] that proposes the elimination of always-on servers and transition of the services to an on-use basis.

Serverless architecture uses the approach of using ephemeral containers that can be launched instantaneously on request and be stopped upon completion of computation [22]. This requires the developers to break down their application into multiple functions as the basic units of computation and these functions are hosted in separate containers that are launched upon request and are destroyed when the function execution is over. This architecture provides the opportunity to reduce the resource usage of containers when they are idle and are not receiving requests and also reduces the cost of always-running servers [20] [24]. Instead, the cost of the infrastructure is only accumulated for the time the function was actually executed. The serverless computing model removes the need for infrastructure management and this provides a great advantage over monolithic and microservice deployment models.

The serverless computing model provides a great opportunity for edge computing, which consists of an event driven architecture and is subject to varying workloads [31] [25]. Processing data on the edge of the network provides a solution for limited network bandwidth and latency problems as well as data privacy concerns by executing code closer to devices and end users [12]. This compute model can provide support for latency-sensitive workloads [10] and help adhere to compliance and privacy regulations. Serverless computing also offers reliability and the ability to scale with no associated management overhead in an edge environment. A serverless architecture for edge computing, what we call it *serverless edge computing*, can provide the benefit of faster deployment, a smaller footprint and increased performance [5].

This book chapter focuses on the performance evaluation of serverless platforms of both public cloud and open source contributors in an attempt to analyse the viability of serverless platforms in an edge environment. In order to emulate the edge environment, we aim to set up the serverless platforms on ARM based single-board computers (SBCs), namely, Raspberry Pis and we evaluate the performance of ARM architecture-compatible serverless platforms on these devices. For small-scale infrastructures that are set up in edge environments, single board computers are commonly used. By simulating a similar environment, this study aims to provide insights on the performance of serverless platform on resource constraint devices as this research does not exist in the present literature. The analysed open-source serverless platforms are OpenFaaS,[1] Apache

---

[1] OpenFaas (2020), `https://docs.openfaas.com`

OpenWhisk,[2] AWS Greengrass.[3] We compare their performance with serverless computing services of AWS Lambda[4] and Azure Functions[5] in an attempt to research the suitability of a serverless architecture on the edge. We set up our experiments by configuring these serverless platforms on a cluster of four Raspberry Pis and subjected them to varying amounts of workloads and analysed the differences in the performance metrics such as response time and success rate for various types of workloads, e.g., CPU-intensive, memory-intensive, and disk-intensive functions.

The organization of this book chapter is as follows: Section 2 provides a background and motivation on the topic. Section 3 focuses on the related works that have been carried out in this research area and describes serverless computing platforms that we have selected for this study. Section 4 discusses our performance evaluation methodology, experimental setup and the analysis of the results. Section 5 presents the discussion on our findings. Finally, Section ?? provides a conclusion to our work and outlines future work.

2   Background

2.1   Motivation

Imagine a course-grained application with several dependencies between its services which is deployed on the cloud. The incoming workload is naturally highly dynamic, so the application requires varied amount of computing resources to respond to the dynamism over the time, so called auto-scaling [2]. This scenario applies to a great number of Web applications hosted in the cloud [3].

The first deployment solution appears resource over-provisioning: providing resources, i.e. VMs, for the application as much as the required resources for the maximum expected incoming workload. This solution will greatly guarantee Quality of Service (QoS), but resource wastage is highly likely. Inefficiency also appears more seriously when hypervisor-based machines, i.e., VMs, with large footprint are intended to host the application.

Another solution for efficiently deploying monolithic applications is to adopting an auto-scaler to dynamically respond to the workload changes by adding or removing replicas of the application, i.e. adding or removing VMs [3]. This can narrow the gap, but auto-scaling large monolithic applications with several interconnected services will not potentially be a sufficiently smart and quick action. It is not smart, as potentially only certain parts of the application may need auto-scaling, not the entire application. A monolithic application is composed of several services tightly coupled with each other and cannot be scaled individually easily. The solution is not quick also, as heavyweight hypervisor-based

---

[2] Apache OpenWhisk (2020), `https://openwhisk.apache.org/documentation.html`

[3] AWS Greengrass (2020), `https://docs.aws.amazon.com/greengrass/`

[4] AWS lambda (2020), `https://docs.aws.amazon.com/lambda/index.html`

[5] Microsoft Azure Functions (2020), `https://docs.microsoft.com/en-us/azure/azure-functions`

virtualization tends to suffer from long boot-up times for newly provisioned VMs, sacrificing QoS. Add to auto-scaling challenges that considerable complexity for deploying auto-scalers will not be practical for non-expert users.

Given the motivational scenario, this question is raised: "how we can re-design cloud platforms to properly respond to the dynamic nature of the application?" Recently, serverless computing has attracted great attention in dealing with such challenges. In the following, a historical view of serverless computing will be presented.

2.2 History of Serverless

In a nutshell, "serverless attended the reunion between technological advances such as microservices, containerization, and the idea of effortless auto-scaling and pure pay per use model [1]." From DevOps perspective, let's fork the advances in development, e.g., coding, and deployment, e.g., installation and maintenance, sides for applications and merge them into a united idea as serverless.

Development advances: monolithic applications, despite easy deployment, are not easily scaled. Hence, service-oriented architectures for applications appeared and advanced towards today's microservices that reshapes a monolithic application into loosely coupled services that can scale individually. Furthermore, the idea of fully decomposited application arose, that assumes each individual microservice can live by itself, under the name of a function. This led to Function-as-a-Services (FaaS) [1] wherein only an individual service is developed for a single unit of task. FaaS also decouples the application for them state and runs stateless functions. Note that the if the state is still required for certain applications while staying in FaaS, the state can be preserved separately by storage services. With this advance, the question is how to take advantage of FaaS in deployment side? and how to bring them in reality?

Deployment advances: given the large footprint of VMs, the containerization [1] came into the picture. This means, instead of abstracting the underlying resource from a hypervisor in order for launching an isolated computing resource as VM, one can employ the OS kernel to launch a semi-isolated container. This idea came from small footprint requirements for computing resources to efficiently provision resources for certain applications. With this in mind and the advances in development side, it is obvious that containerization will definitely suit FaaS, where single units of computation are intended to be provisioned and scaled.

One step is left to serverless. Such advances helped decoupling monolithic applications to fine-grained FaaS and help provisioning lightweight resources, i.e., containers instead of VMs for making FaaS a reality. However, the challenges of auto-scaling and maintenance are still remained unsolved and poses the question that how to benefit from such advances for efficient execution of applications?

That was a problem until Amazon introduced Lambda [1] for running functions in terms of FaaS that run on ephemeral short-lived containers. The auto-scaling and maintenance is left to the serverless platform. Precisely, serverless platforms enable developers to write their applications in terms of functions

in any language of interest. Then, the platform is responsible for provisioning resources, e.g., container, per each request (invocation) to the function. Simplistically, once a request is made to the function, the serverless platform will run a new container and after completing the task the container is terminated or stays alive for a short period of time for reusability. Functions are intended to run ephemerally and for short-running tasks. Note that the idea of FaaS and serverless appeared under the same banner historically and occasionally are used interchangeably. An added bonus with serverless offered by cloud providers is also to realize a pure pay pay use pricing model wherein one just pays for the actual execution time of tasks. In conventional cloud pricing models, customers pay for the duration of time the resources are provisioned, regardless of being actually used or not.

This provides a huge advantage over other infrastructures such as Platform-as-a-Service (PaaS) or Container-as-a-Service (CaaS) as servers do not need to be running constantly in the background accumulating costs. Functions start within milliseconds and process individual requests. If there are several simultaneous requests for your function, then the system will create as many copies of the function, that will be managed by the container orchestration system, as needed to meet demand. When demand drops, the application automatically scales down. Dynamic scaling is a benefit of FaaS, and is cost-efficient as well, because providers only charge for the resources that are used, not idle time [21]. When running on premise, this dynamic nature can also increase platform density, allowing more workloads to run and optimize resource consumption. An event-driven service that needs horizontal scaling can work well as a function, as well as RESTful applications [8]. Figure 1 shows a schematic serverless architecture.

Given the emergence of AWS Lambda serverless in 2014 as the pioneer, several companies and individual users employed this platform in practice. Other IT companies such as Microsoft (Azure Functions), Google (Google Cloud Functions) IBM (Open Whisk), etc. also attended the market and offered their own serverless platforms. Open-source platforms also came to the picture such as OpenFaaS, Kubeless, etc.
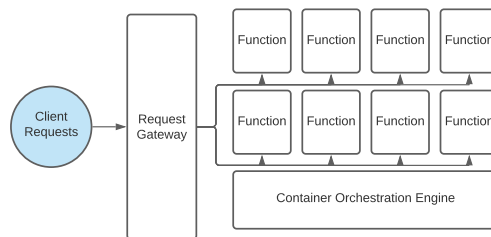

Fig. 1: Serverless Architecture

2.3 Serverless at the Edge

Given the focus area of this research —serverless edge computing— this question is raised that "Why will serverless be good practice for edge computing?"

Edge computing is intended to bring computation closer to data sources and of the biggest stakeholders of edge will be IoT applications [1]. Let's analyze how IoT applications work. Typically, a group of sensors are located in a particular area, e.g., street, farm, factory, body, etc., in order for collecting and sending data to a computing node for execution. The occurrence of such events is highly variable. The execution also would involve short-running tasks such as analyzing if a body temperature is higher than a certain degree or if the moisture content of the soil is at a certain level. Add to these characteristics of IoT applications that they will potentially face energy preservation challenges of edge nodes as well. An edge computing platform for several IoT use cases is deemed to perform on low-power devices such as SBCs that demands certain considerations for energy saving.

Obviously, IoT applications with event-driven nature and short-running tasks characteristics will match with serverless platforms functionality. Serverless tends to avoid always-on deployment of applications and instead tends provision containers per requests and to terminate after execution. Hence, this will be a huge step towards energy saving on edge nodes as well.

Theoretically, serverless appears practical for adaptation in edge. This idea has attracted cloud serverless providers to re-design their platform to support edge specific requirements as well. AWS Lambda was once again of the dominant in this move and AWS is now offering relative services such as Greengrass. In academia, researchers also made effort to assess the feasibility of serverless at the edge for different IoT use cases, which will be elaborated in the next section.

3 Related Works

Function-as-a-Service (FaaS) is an event-driven computing execution model that runs in stateless containers and those functions manage server-side logic and state through the use of services [29] [9]. It allows developers to develop, deploy, and manage those application services as functions without having to maintain their own infrastructure. FaaS provides developers with the flexibility to develop event-driven applications without managing servers. Serverless infrastructure has the ability to scale to zero, i.e. when there are no requests, the servers are stopped and are only running when they are needed [11]. Serverless computing is a relatively new area and has received a lot of attention in recent years [30] [16]. This is due to the potential the serverless computing architecture offers and the need for being validated by research. The work of Van Eyk et al. [30] provides a detailed description and explanation of the serverless computing architecture along with the evolution of cloud computing that led to the rise of serverless computing. Lee et al. [16] performed a performance analysis and comparison of the serverless computing services from major cloud providers such as AWS,

Azure, GCP and IBM Cloud and analysed the performance of these services in a production environment.

The work done by Mohanty et al. [18] researched multiple open source serverless platforms and comprehensively compared the features and architecture of each platform in order to identify the most suitable platform for production environments. They further elaborated on their research by also comparing the performance of these platforms when running on a Kubernetes cluster. Another study done by Palade et al [23] is related to our work as they focus on the performance of open source serverless computing platform at the edge. The serverless platforms that were compared in that study are Kubeless, OpenFaaS, Knative and Apache OpenWhisk. However, their study does not take into account the computational limitations of edge devices as the platforms are setup on Desktop grade machines and are used to compute the data that is being generated by edge devices. Their study also evaluates qualitative features of these platforms, such as programming language support, container orchestration engine support, monitoring support and CLI interfaces, and the quantitative evaluation, which measures response time, throughput and success rate, and does not include resource intensive tasks. This book chapter provides insights into how a serverless architecture on the edge looks like when we use resource constrained devices such as Raspberry Pis.

Baresi et al. [5] discuss the adoption of serverless architectures on the edge and proposes a new serverless platform that can be suitable specifically for an edge environment. Lloyd et al. [17] discusses the performance of AWS Lambda and Azure Functions and compares it across multiple metrics including latency and the effect of warm and cold starts on serverless functions. Shillaker et al. [28] evaluates the performance of OpenWhisk and evaluates the response times at varying levels of concurrency. However, none of the works analysed the performance of AWS Lambda function running on the edge using AWS Greengrass and none of the works explicitly compare the performance of both open source and public cloud serverless platforms that is a significant deciding factor in the viability of serverless platforms on the edge. Moreover, to the best of our knowledge, this is the first work that evaluates the performance of open source serverless platforms that are set up on resource constraint edge devices like Raspberry Pis.

In this book chapter, we compare the performance of *AWS Lambda*, *AWS Greengrass* (running local lambda functions), *Azure Functions*, *Apache OpenWhisk*, and *OpenFaaS*. In the following subsections, we briefly describe each of these severless platforms. Table 1 also provides the qualitative evaluation of the various serverless platforms that we analysed in this book chapter.

3.1 AWS Lambda

Amazon Web Services (AWS) offers their serverless computing service called Lambda. AWS Lambda was the pioneer of public cloud serverless computing services as AWS was the first major cloud computing provider to provide a serverless compute service. AWS Lambda follows the FaaS architecture and allows the developers to focus on their application and not worry about manag-

ing infrastructure [32]. AWS Lambda supports multiple programming languages such as JAVA, Go, Python, Ruby. etc. and the cost model is based the computational expenses of the function execution i.e. the cost is only accumulated for the amount of time that the function is executed. AWS Lambda also provides its Lambda@Edge service which utilizes the AWS Edge Locations in an attempt to reduce latency for its customers and the functions are executed closer to the application which results in lower response times. AWS allows Lambda functions to be run on local edge devices using AWS Greengrass service which is discussed in the next section.

3.2 AWS Greengrass

AWS Greengrass is a service provided by AWS that allows its customers to run local compute, messaging, and multiple AWS services on a local edge device. AWS Greengrass allows the developer to set up a Greengrass core device that acts as the primary point of connection between the edge setup and the cloud servers. Other devices can either be added as worker nodes for computation or as 'Things' that are devices that are able to communicate with the core device. AWS Greengrass primarily uses MQTT [19] protocol to communicate between the core devices and the IoT things that are registered in the AWS Console. Greengrass is relevant to our study because AWS allows compute services such as AWS Lambda to be run on edge devices in order for the computation to be closer to the application and user. We use ARM devices to set up our serverless platforms. Luckily, Greengrass natively supports ARM architecture and allows the Greengrass Core to be set up on a Raspberry Pi along with providing the ability to add multiple Raspberry Pi devices as worker nodes for the core device.

3.3 Azure Functions

Microsoft Azure provides Azure Functions as its serverless compute service and follows the same paradigm of serverless computing as discussed previously. The underlying infrastructure is managed by Azure Function providing the developer to focus on their code and building their applications following the serverless development paradigm [15]. Azure Function use authentication to keep out unwanted entities from the system. Azure Functions is a relatively new service, so there are fewer studies that have analysed its performance and compared it to other public cloud serverless offerings.

3.4 Apache OpenWhisk

Apache OpenWhisk is an open-source serverless platform currently in incubation. IBM Cloud has built their IBM Cloud Functions on top of Apache OpenWhisk utilizing the power of OpenWhisk's services. OpenWhisk has multiple components and relatively consumes more resources due to the added components [26]. For the purposes of this research, we will be utilizing the Lean

	OpenFaaS	OpenWhisk	AWS Lambda	AWS Greengrass	Azure Functions
Characteristics	Allows the development of serverless functions in multiple languages and natively supports Docker. Easy deployment. Auto-scaling according to demand. Portable as it runs on existing hardware and can be deployed on both public and private cloud.	Writes functional logic called actions which can be scheduled and triggered via HTTP. Highly scalable and resilient.	Lambda runs your code on high-availability compute infrastructure and performs all the administration of the compute resources, including server and operating system maintenance. Capacity provisioning and automatic scaling. Code and security patch deployment. Code monitoring and logging. Fault Tolerance.	Runs AWS Lambda functions on the device to respond quickly to local events, interact with local resources, and process data to minimize the cost of transmitting data to the cloud. Support local publish/subscribe messaging between components. Highly secure.	Easily develop and run massively parallel real-time analytics on multiple streams of data – including IoT – using Azure Stream Analytics. With no infrastructure to manage, process data on demand, scale instantly and only pay per job.
Programming Languages Supported	Go, NodeJS, Python, Java, Ruby, PHP, and C#	NodeJS, Go, Java, Scala, PHP, Python, Ruby, and Swift	Java, Go, PowerShell, NodeJS, C#, Python, and Ruby	Java, NodeJS, Python	Java, Python, TypeScript, F#, C#, Powershell, and JavaScript
Intended Infrastructure	Kubernetes, Docker Swarm, extendable to other orchestrators, Public Cloud supported	No orchestrator required, Kubernetes supported, Public Cloud supported.	NA	Docker Compose	NA
Virtualization	Docker	Docker	Micro-VM/firecracker	Micro-VM/firecracker	Docker
Triggers	HTTP, faas-cli	HTTP, OW-CLI	S3, SNS, DynamoDB, CloudWatch, Config Rules, API Gateway, Greengrass.	MQTT Event, AWS Greengrass CLI	Azure Event Hub and Azure Storage, Web Triggering, trigger types and scheduled types.
Billing	NA	NA	Price according to the memory allocation. $0.00001667 per GB-second.	Monthly Price per device. $0.16 per month	Price according to the memory allocation. $0.000016 per GB-second.
Limitations	Does not provide authentication	Open Whisk does not have many options for triggering actions since it is a bit difficult to integrate them. Also, there is a problem in the execution of the concurrent systems.	Maximum function runtime is 15 minutes, and the default timeout is 3 seconds which makes it unsuitable for long-running workloads. The payload for each invocation of a Lambda function is limited to 6MB, and memory is limited to just under 3GB.	Lack of programming language support currently. Steep learning curve. Limited support for integration with other AWS services.	Maximum function runtime is 10 minutes. Memory usage limit is 1.5 GB. This memory is shared among all Functions in the application.
Communication Pattern	Request/Reply	Request/Reply	Request/Reply	Publish/Subscribe and Request/Reply	Request/Reply
State	Stateless	Stateless	Stateless	Stateless	Stateless

Table 1: Qualitative Evaluation of Serverless platforms

OpenWhisk branch of the Apache OpenWhisk platform as it is less resource intensive and more suitable for edge environments compared to its full version. Lean OpenWhisk supports the same number of languages as its master branch but is not natively compatible with the ARM architecture. OpenWhisk can be run on Kubernetes or Docker compose [14]. For the purposes of this study, we will be setting up Lean OpenWhisk on Docker compose as it consumes less system resources. In order for it to be compatible, custom Docker images of the Lean OpenWhisk platform have to be created for the ARM architecture as it currently only supports x64 and x86 architectures.

3.5 OpenFaaS

OpenFaaS is a lightweight open-source serverless platform that provides native compatibility with the ARM architectures and is easy to set up and start running production workloads. OpenFaaS is compatible with multiple system architectures and can be run on top of Kubernetes and Docker Swarm. OpenFaaS follows the same serverless compute paradigm as the other serverless platforms and provides services for resource and performance monitoring and supports multiple programming such as Python, JavaScript, Ruby etc. Faasd is a variant of OpenFaaS that utilizes Containerd as the container orchestration system and is lighter than OpenFaaS. However, faasd does not support setup on multiple Raspberry Pi devices in a cluster which is why we use OpenFaaS in our experiments.

4 Performance Evaluation

We analyze the performance metrics including response times and success rate for the requests measured for each of the aforementioned platforms for the sake of comparison. The response times consists of the time it takes a request to reach the server, the time to execute the function on the server, and the time it takes the response to be delivered back to the client. Success rate measures as the percentage of requests that were executed successfully. As displayed in Figure 2, the local functions are run on a cluster of 4 Raspberry Pis, whereas calls are made to OpenFaas, OpenWhisk and AWS Greengrass in our serverless setup on the local network and also to the cloud functions in AWS Lambda and Azure Functions from the same test machine. This allows us to compare the performance of running local functions as compared to cloud functions which will be a good deciding factor on the viability of running serverless workloads on the edge or running the workloads on the cloud.

4.1 Experimental Setup

We run the experiments on a cluster of four Raspberry Pis, with our requests being generated by a test machine running JMeter on the same local network. For the purpose of accuracy, the experiments are run on an isolated network so that there is no other network overhead that interferes with the network. These
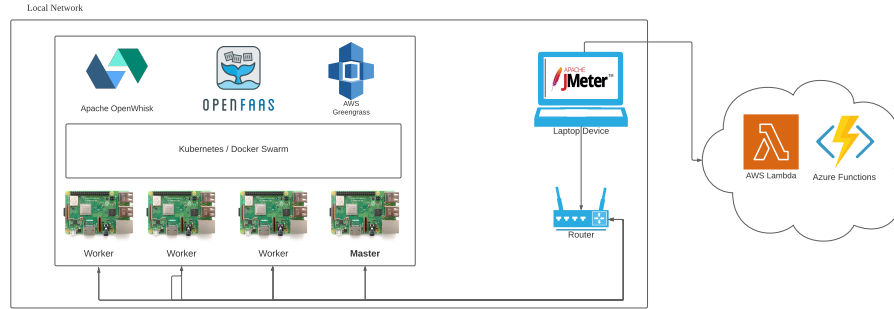
Fig. 2: Deployment setup

tests have been carried in the city of Melbourne, Australia. The deployment of each platform is done individually in order to make sure no other services are consuming the system resources. This is especially important due to the limited resources available on the edge devices and compute power plays an essential role in our results. We use Raspberry Pi Model 3B+ devices running Raspberry Pi OS which is a Linux distribution designed specifically for Raspberry Pi devices.

Testbed Setup Each Raspberry Pi has a 1.4GHz 64-bit quad-core ARM processor and 1GB RAM. Unless stated otherwise, we used the default deployment settings for each platform, for example, Docker Swarm as the container orchestration system for OpenFaas. [6] We use Apache's Lean OpenWhisk offering that utilizes no Kafka, Zookeeper and no Invokers as separate entities which is suitable for the ARM architecture. The Lean OpenWhisk branch of the Apache OpenWhisk project does not provide compatibility with the ARM architecture of the Raspberry Pi. Each component of OpenWhisk runs as a Docker container and these Docker images have been configured for the ARM architecture in this docker repository.[7] We created the ARM compatible Docker images to set up OpenWhisk on each Raspberry Pi in the cluster. This setup does not support multiple Raspberry Pi devices in a cluster, so we use an NGINX Load Balancer to distribute our HTTP requests to the Raspberry Pi cluster.

We use AWS Greengrass to set up our cluster which involves setting up a Greengrass Group on the AWS Management Console and an AWS Greengrass IoT Core on the master Raspberry Pi. The other Raspberry Pi devices are registered as non-core devices and communicate with the core over localhost. The region that the remote Lambda functions are running on Greengrass is set as *ap-southeast-2 i.e. Asia Pacific (Sydney) region*. For our AWS Lambda setup, the region is set as *ap-southeast-2 i.e. Asia Pacific (Sydney, New South Wales) region* in order to maintain consistency and minimize network latency. For our

---

[6] OpenFaas has switched to Kubernetes as the officially supported orchestration system. This update was provided by OpenFaas after the experiments were conducted in this book chapter.

[7] Docker (2020), `https://docs.docker.com/`

Azure Functions setup, the region is set to *Australia East* which is also the *New South Wales region*. Azure does offer the Australia South East region that is the Victoria region, but for the purposes of comparison with AWS Lambda, the Australia East region is selected. Table 2 shows the average ping results from the test machine to the AWS and Azure servers in Sydney and also the ping results to the raspberry pis on the local cluster. This provides us with the time associated with network latency in our experiments.

Destination	Ping
Raspberry Pi Cluster (local network)	4ms
AWS Sydney Region (ap-southeast-2)	42ms
Azure Sydney Region (Australia East)	51ms

Table 2: Ping results from the test machine

We use Apache JMeter version 5.3 to generate HTTP requests that invoke the functions deployed on each platform. We run JMeter on a Windows Machine that has a quad-core Core i7 CPU and 8GB RAM. This machine is set up on the same local network as the Raspberry Pi Cluster in order to minimize latency. The JMeter tool is set up to send 1000 requests with different levels of concurrency (5, 10 and 15). These concurrent requests affect the number of simultaneous requests received by each platform. These concurrency levels were decided based on the compute power available to the platforms and each experiment is replicated 3 times in order to maintain statistical accuracy. Specifically for AWS Greengrass, we use the JMeter MQTT plugin which extends the JMeter capability to send MQTT requests instead of HTTP requests as AWS Greengrass requires the user to send MQTT requests.

Test Functions The three types of functions that we use to test our serverless platforms are CPU-intensive function, memory-intensive functions and disk-intensive functions. We use JavaScript for our functions as it is supported by all of our test platforms. The workloads designed for testing these platforms are representative of the various types of workloads that are generated in an edge environment. Due to the lack of research on serverless platforms' performance on resource-constrained devices, this workload has been designed to test how each platform performs when subjected to an emulated workload. In order to simulate a CPU-Intensive function, we multiply two square matrices of 128 by 128 dimensions that has a complexity of $O(n^3)$. The execution time of this function varies on the computational power available to the compiler. For the memory-intensive function, we utilize the *memoize()* function of javascript that allows us to store the expensive results of our functions in the cache and retrieve them quickly from the cache if the same result occurs [6].

To simulate a disk-intensive workload, we created a function to unzip 10 zipped files of size 2.5MB onto the disk. This utilizes both read and write operations for the function. The disk-intensive function requires an extensive setup especially for the cloud functions that we are executing using AWS Lambda and Azure Functions. This is because these services are event-driven and make use

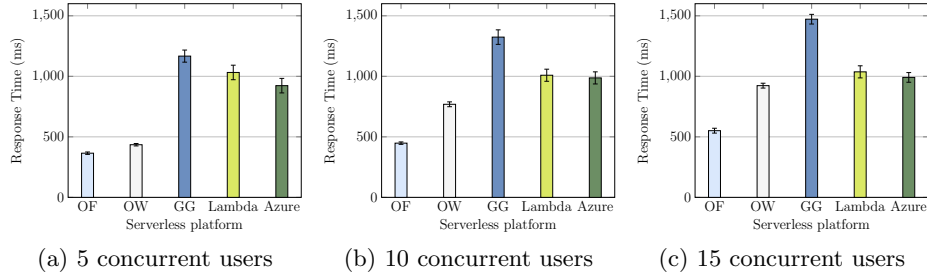(a) 5 concurrent users (b) 10 concurrent users (c) 15 concurrent users

Fig. 3: Median response time with standard error bars on CPU-Intensive tasks for OpenFaas (OF), OpenWhisk (OW), AWS Greengrass (GG), AWS Lambda (Lambda), Azure Functions (Azure) with various number of concurrent users.

of ephemeral containers that are stateless and do not have access to file system that can be shared across all other functions. While a temporary file system for the individual function is included, this ephemeral storage is not intended for durable storage which is why need to attach a shared file system for the functions to access shared data. For AWS Lambda, we created an EFS File Systemon the AWS Console and provided read/write permissions for Lambda. Then we added the EFS File System in the Lambda configuration to as Local Mount PathAzure Functions follows a similar procedure to AWS Lambda. We used Azure Blob Storage to provide file access to our Azure Function. For the other platforms, we placed the zip files on each Raspberry Pi in the cluster as the devices do not share a file system. In future work, we would like to test these platforms on actual IoT application workload traces in order to gain better insight on the performance of each serverless platform.

4.2   Results

In order to compare the performance of each platform, we measure the response times of each request under different levels of load. The number of concurrent requests directly affects the performance of platforms which alludes to the number of function instances being created in order to handle the requests. The function instances consume the system resources and we want to measure the effect of this on the performance of each platform. Our main aim is to find out the performance issues for each platform and how it handles different types of serverless workloads i.e. CPU-intensive, memory-intensive and disk-intensive functions.

CPU-Intensive Functions Figure 3 shows the median response times for each platform for the 1000 requests that we sent under different concurrency. These response times include the time taken to send the request, the time taken to execute the function and the time taken to receive the result to our test machine. This is an asynchronous request so it waits for the function to be executed and then receives the response. In each of the results, the lowest response time recorded is for the OpenFaaS platform. We observe that the lowest response

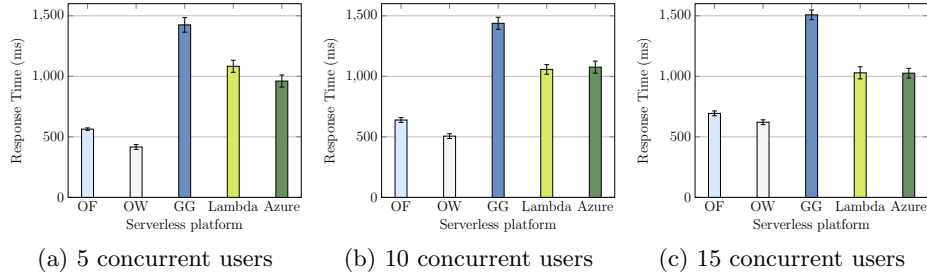(a) 5 concurrent users   (b) 10 concurrent users   (c) 15 concurrent users

Fig. 4: Median response time with standard error bars on Memory-Intensive tasks for OpenFaas (OF), OpenWhisk (OW), AWS Greengrass (GG), AWS Lambda (Lambda), Azure Functions (Azure) with various number of concurrent users.

time is when we send 5 concurrent requests to OpenFaaS with values around 400ms. OpenWhisk performs fairly similarly with values around 450ms. These response times increase as we increase the number of concurrent requests to 10 where OpenWhisk is impacted considerably more than OpenFaaS as the response time increases to 750ms. OpenFaaS response time rises to 448ms when dealing with 10 concurrent requests. The response times for OpenWhisk rise even more when subject to 15 concurrent requests as the response time is 923ms. As compared to OpenFaaS, the response time remain fairly stable when we increase the concurrent requests.

	OpenFaaS	OpenWhisk	AWS Greengrass	AWS Lambda	Azure Functions
CPU	100%	99.1%	100%	100%	100%
Memory	99.5%	99.3%	100%	100%	100%
Disk	99.9%	99.6%	100%	100%	100%

Table 3: Request Success Rate for platforms under different workloads

As shown in Table 3, OpenWhisk records the worst success rate in our experiments for CPU-intensive workloads. We recorded 8 failed requests out of 1000 as a result of an increase in the system's resource consumption. OpenFaaS records a perfect success rate with no failed requests deducing that it handles CPU-intensive workloads very well under various loads. These results show that performance of OpenWhisk decreases significantly as we increase the number of concurrent requests and the reason for that is because OpenWhisk consumes more compute power to run its components as compared to OpenFaaS which is more lightweight. If we analyze the response times for AWS Greengrass, we can compare the rise in latency is not that significant when we increase the number of concurrent requests accordingly. However, AWS Greengrass records the highest response times out of all the platforms, due to the time it takes for the request to reach the AWS server to trigger the function on the core device. AWS Lambda and Azure Functions performs very similarly for each level of concurrency and show very slight changes in latency which is predictable as these platforms do not have as limited resources as compared to the other platforms and provide highest level of scalability. For our CPU-intensive functions, AWS Greengrass,

AWS Lambda and Azure Functions and OpenFaas do not have a single failed request as compared to OpenWhisk.

Memory-Intensive Functions Figure 4 demonstrates the results for the performance of each serverless platform for memory-intensive functions. The response times were recorded for each platform until varying concurrent requests in order to observe the effect it has on the performance. The lowest response times, in each of the tests for memory-intensive functions, were for Apache OpenWhisk that performed considerably better than the other platforms and the results were quite stable with only a gradual increase in response time when we increased the number of concurrent requests. The lowest median response time for OpenWhisk is 416ms for 5 concurrent requests and it reaches a maximum of 621ms for 15 concurrent requests. OpenFaaS, however, demonstrated higher response times than OpenWhisk for memory-intensive functions but the response time varied less between different concurrent requests which proposes that this increase in latency is less likely to be caused by OpenFaaS ability to scale and is more likely attributed to memory management for function execution in the platform.

The highest response times were for AWS Greengrass that maintained its values between 1400ms to 1500ms between varying concurrent requests. We observed that AWS Greengrass took more time for memory-intensive functions than CPU-intensive functions but is affected less by change in concurrent results. In fact, there is only a 13ms difference between the results we send 5 concurrent requests and 10 concurrent requests and Greengrass only witnesses a slight increase in latency when we increase the concurrent requests to 15. Both AWS Lambda and Azure Functions perform very similarly with little changes in latency when we increase the number of concurrent requests. Both of the platforms provided similar results to our CPU-intensive tasks. This is attributed to both of the cloud serverless offerings' ability to scale indefinitely and also due to the higher computational power available to the functions. However, as the functions are not being executed locally and need to communicate with the cloud servers, the latency is much higher as compared to the platforms that are set up locally.

Disk-Intensive Functions Figure 5 shows the results for the evaluation of each platform for disk-intensive functions. The lowest response times for this experiment are recorded by OpenFaaS and the platform scaling very well when we increased the number of concurrent requests. The response time only increased by 50ms between 10 and 15 concurrent requests which demonstrates the new functional replicas were able to handle the increased load easily. However, we noticed that OpenWhisk recorded higher response times as compared to OpenFaaS and when we increased the load from 10 to 15 concurrent requests, it caused an increase in response time of 200ms. The success rates for OpenFaaS and OpenWhisk were also very acceptable as we recorded very few failed requests as compared to the other function types. AWS Greengrass showed similar trends to OpenWhisk as the response times for 5 and 10 concurrent requests
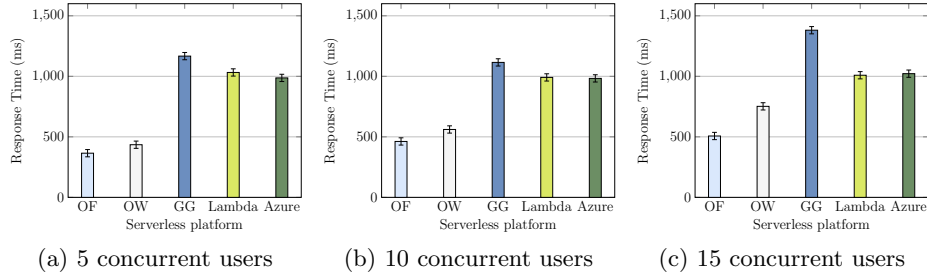
(a) 5 concurrent users   (b) 10 concurrent users   (c) 15 concurrent users

Fig. 5: Median response time with standard error bars on Disk-Intensive tasks for Open-Faas (OF), OpenWhisk (OW), AWS Greengrass (GG), AWS Lambda (Lambda), Azure Functions (Azure) with various number of concurrent users.

are very similar but an increase in load to 15 requests caused the latency to increase by 220ms. This can be attributed to the creation of new function replicas in both cases of OpenWhisk and AWS Greengrass. AWS Lambda and Azure Functions both displayed similar results for this experiment, as we did not notice any changes in latency caused due to an increase in load or the function type. The file storages for both platforms were configured on the same server as the functions, which shows us that there is minimum latency for disk read and write operations. The results for the success rates for AWS Greengrass, AWS Lambda and Azure Functions were the same as the previous experiments as they recorded perfect success rates with no failed requests.

5   Discussion

Our experimental results show that OpenFaaS performs considerably better on the edge than the other platforms. We can attribute this to the lean architecture of the platform which suits very favorably to edge devices and consumes fewer resources which is very suitable for edge devices. Along with that, it provides native support for the ARM architecture, along with the ability to scale easily by adding more raspberry pi devices. However, the results vary for high loads especially for CPU-intensive and memory-intensive workloads. We found out that the performance of Apache OpenWhisk is inferior to OpenFaaS in our setup as OpenWhisk does not provide full support for the ARM architecture and the lean version restricts its performance when handling increased workloads. This performance bottleneck may be attributed the high memory usage of the platform that leads to resource starvation for new containers. Furthermore, the ability to add more worker nodes with ease is an integral part of edge computing setup, which OpenWhisk does not support natively.

AWS Greengrass performed very consistently across our experiments, but the increased response times are a considerable factor when making the decision to deploy this platform on the edge. The cause of these increased response times is the added latency of the requested being routed to the AWS cloud servers. The increased latency for CPU-intensive tasks also suggests that the functions take longer to compute due to resource starvation. AWS Greengrass seamlessly

extends AWS to edge devices so they can act locally on the data they generate, while still using the cloud for management, analytics, and durable storage. It enables the connected devices to run AWS Lambda functions, keep device data in sync, and communicate with other devices securely. AWS Greengrass provides support for ARM architecture and provides the ability to scale out by adding more worker devices. AWS Greengrass provides an opportunity for further research to be conducted by testing its performance in an isolated edge without connectivity to the cloud.

If we want consistent performance regardless of the amount of workload, the cloud serverless offerings of AWS Lambda and Azure Functions provide a stable service and can scale according to the increased load. This demonstrates that for workloads that require increased computational capacity, along with the ability to handles a workloads at an increased scale, the cloud serverless offerings are very suitable. However, the overall latency to cloud is higher than the edge devices as experiments show and running serverless platforms such as OpenFaaS on the edge provides promising prospects to meet the stringent latency requirements of emerging real-time applications such as autonomous vehicles. Nevertheless, our work provides the groundwork for comprehensively benchmarking the performance of serverless platforms on the edge, specifically on devices with ARM architecture.

6   Conclusions and Future Work

Performing serverless computations on the edge proposes numerous advantages in an event-driven architecture, allowing us to generate more data on the edge without having to worry about managing its applications. This book chapter analysed the performance of serverless platforms on the edge, specifically on ARM architecture edge devices such as Raspberry Pis. We performed comprehensive performance tests on the compatible serverless platforms of OpenFaaS, Apache OpenWhisk and AWS Lambda functions running locally in the AWS Greengrass platform on edge devices, and compared their performance to cloud serverless offerings of AWS Lambda and Azure Functions to research the viability of setting up a serverless architecture in an edge environment. Based on that, the results demonstrated that OpenFaaS is the most suitable platform for an edge setup as it provides a lightweight architecture with support for simple and rapid scaling. We used the metrics of response time and success rate for each platform to compare the performance on how each platform would cope in an edge setup. AWS Greengrass provides a promising opportunity in this environment for further research due to its native support for ARM architecture and its development support from AWS. For future research, we aim to analyse other suitable serverless platforms and develop ARM architecture support and further research the viability of serverless architecture on the edge.

Bibliography

[1] Aslanpour, M.S., Toosi, A.N., Cicconetti, C., Javadi, B., Sbarski, P., Taibi, D., Assuncao, M., Gill, S.S., Gaire, R., Dustdar, S.: Serverless edge computing: Vision and challenges. In: 2021 Australasian Computer Science Week Multiconference. ACSW '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437378.3444367, `https://doi.org/10.1145/3437378.3444367`

[2] Aslanpour, M.S., Toosi, A.N., Gaire, R., Cheema, M.A.: Auto-scaling of web applications in clouds: A tail latency evaluation. In: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC). pp. 186–195 (2020). https://doi.org/10.1109/UCC48980.2020.00037

[3] Aslanpour, M.S., Toosi, A.N., Taheri, J., Gaire, R.: AutoScaleSim: A simulation toolkit for auto-scaling Web applications in clouds. Simulation Modelling Practice and Theory **108**, 102245 (apr 2021). https://doi.org/10.1016/j.simpat.2020.102245, `https://linkinghub.elsevier.com/retrieve/pii/S1569190X20301738`

[4] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless computing: Current trends and open problems. In: Research Advances in Cloud Computing, pp. 1–20. Springer (2017)

[5] Baresi, L., Mendonça, D.F.: Towards a serverless platform for edge computing. In: 2019 IEEE International Conference on Fog Computing (ICFC). pp. 1–10. IEEE (2019)

[6] Berube, D.: Speeding up function calls with memoize. Practical Ruby Gems pp. 215–220 (2007)

[7] Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: Serverless programming (function as a service). In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 2658–2659. IEEE (2017)

[8] Chard, R., Skluzacek, T.J., Li, Z., Babuji, Y., Woodard, A., Blaiszik, B., Tuecke, S., Foster, I., Chard, K.: Serverless supercomputing: High performance function as a service for science. arXiv preprint arXiv:1908.04907 (2019)

[9] Eismann, S., Scheuner, J., van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C., Iosup, A.: Serverless applications: Why, when, and how? IEEE Software (2020). https://doi.org/10.1109/ms.2020.3023302, `http://dx.doi.org/10.1109/MS.2020.3023302`

[10] Glikson, A., Nastic, S., Dustdar, S.: Deviceless edge computing: extending serverless computing to the edge of the network. In: Proceedings of the 10th ACM International Systems and Storage Conference. pp. 1–1 (2017)

[11] Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. pp. 225–236 (2019)

[12] Hellerstein, J.M., Faleiro, J., Gonzalez, J.E., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: Serverless computing: One step forward, two steps back. arXiv preprint arXiv:1812.03651 (2018)

[13] Hussein, M.K., Mousa, M.H., Alqarni, M.A.: A placement architecture for a container as a service (caas) in a cloud environment. Journal of Cloud Computing **8**(1), 7 (2019)

[14] Kuntsevich, A., Nasirifard, P., Jacobsen, H.A.: A distributed analysis and benchmarking framework for apache openwhisk serverless platform. In: Proceedings of the 19th International Middleware Conference (Posters). pp. 3–4 (2018)

[15] Kurniawan, A., Lau, W.: Introduction to azure functions. In: Practical Azure Functions, pp. 1–21. Springer (2019)

[16] Lee, H., Satyam, K., Fox, G.: Evaluation of production serverless computing environments. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD). pp. 442–450. IEEE (2018)

[17] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S.: Serverless computing: An investigation of factors influencing microservice performance. In: 2018 IEEE International Conference on Cloud Engineering (IC2E). pp. 159–169. IEEE (2018)

[18] Mohanty, S.K., Premsankar, G., Di Francesco, M., et al.: An evaluation of open source serverless computing frameworks. In: CloudCom. pp. 115–120 (2018)

[19] MQTT: Mqtt protocol docs (2020), `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html`

[20] Nastic, S., Rausch, T., Scekic, O., Dustdar, S., Gusev, M., Koteska, B., Kostoska, M., Jakimovski, B., Ristov, S., Prodan, R.: A serverless real-time data analytics platform for edge computing. IEEE Internet Computing **21**(4), 64–71 (2017)

[21] Nguyen, H.D., Zhang, C., Xiao, Z., Chien, A.A.: Real-time serverless: Enabling application performance guarantees. In: Proceedings of the 5th International Workshop on Serverless Computing. pp. 1–6 (2019)

[22] Nupponen, J., Taibi, D.: Serverless: What it is, what to do and what not to do. In: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 49–50. IEEE (2020)

[23] Palade, A., Kazmi, A., Clarke, S.: An evaluation of open source serverless computing frameworks support at the edge. In: 2019 IEEE World Congress on Services (SERVICES). vol. 2642, pp. 206–211. IEEE (2019)

[24] Pinto, D., Dias, J.P., Ferreira, H.S.: Dynamic allocation of serverless functions in iot environments. In: 2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC). pp. 1–8. IEEE (2018)

[25] Premsankar, G., Di Francesco, M., Taleb, T.: Edge computing for the internet of things: A case study. IEEE Internet of Things Journal **5**(2), 1275–1284 (2018)

[26] Quevedo, S., Merchán, F., Rivadeneira, R., Dominguez, F.X.: Evaluating apache openwhisk-faas. In: 2019 IEEE Fourth Ecuador Technical Chapters Meeting (ETCM). pp. 1–5. IEEE (2019)

[27] Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE internet of things journal **3**(5), 637–646 (2016)

[28] Shillaker, S.: A provider-friendly serverless framework for latency-critical applications. In: 12th Eurosys Doctoral Workshop. p. 71 (2018)

[29] Taibi, D., El Ioini, N., Pahl, C., Niederkofler, J.: Patterns for serverless functions (function-as-a-service): A multivocal literature review. In: Ferguson, D., Helfert, M., Pahl, C. (eds.) CLOSER 2020 - Proceedings of the 10th International Conference on Cloud Computing and Services Science. vol. 1, pp. 181–192. SCITEPRESS (2020). https://doi.org/10.5220/0009578501810192

[30] Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uță, A., Iosup, A.: Serverless is more: From paas to present cloud computing. IEEE Internet Computing **22**(5), 8–17 (2018)

[31] Varghese, B., Wang, N., Barbhuiya, S., Kilpatrick, P., Nikolopoulos, D.S.: Challenges and opportunities in edge computing. In: 2016 IEEE International Conference on Smart Cloud (SmartCloud). pp. 20–26. IEEE (2016)

[32] Villamizar, M., Garces, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., et al.: Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). pp. 179–182. IEEE (2016)

[33] Yu, W., Liang, F., He, X., Hatcher, W.G., Lu, C., Lin, J., Yang, X.: A survey on the edge computing for the internet of things. IEEE access **6**, 6900–6919 (2017)