National College of Ireland

Parallel Serverless Workloads in OpenWhisk on Kubernetes

MSc Research Project
MSc in Cloud Computing

Neeti Sharma
Student ID: 20242778

School of Computing
National College of Ireland

Supervisor: Shivani Jaswal

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Neeti Sharma
Student ID:	20242778
Programme:	MSc in Cloud Computing
Year:	2022
Module:	MSc Research Project
Supervisor:	Shivani Jaswal
Submission Due Date:	15/08/2022
Project Title:	Parallel Serverless Workloads in OpenWhisk on Kubernetes
Word Count:	5720
Page Count:	18

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Neeti Sharma
Date:	18th September 2022

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Parallel Serverless Workloads in OpenWhisk on Kubernetes

Neeti Sharma
20242778

Abstract

Recent advances in virtualisation technology have resulted in the widespread use of Serverless computing, which provides the ability to programme and administer the cloud in an autoscaling, pay-as-you-go fashion. The platform handles the invocation, execution, and scaling of a function thereby, managing the complete lifecycle of a function. The most famous offering of Serverless computing is Function as a Service (FaaS). Though FaaS provides ease of use, and flexibility it also possesses significant overheads such as vendor-lock-ins, complex pricing models, functional restrictions, and security. In this paper, an open-source serverless framework has been proposed which uses Apache OpenWhisk as FaaS on a single master multi-worker Kubernetes cluster deployed on private cloud NCI OpenStack. The results show that OpenWhisk deployed on Kubernetes works seamlessly on OpenStack with the proposed framework and performs better when the compute intensive workloads are parallelised for C++, Java and Go. The computed results not only demystify the feasibility of OpenWhisk on Kubernetes but also checks the resource utilisation and performance of the system for parallelised workloads.

1 Introduction

Serverless computing has recently become one of the most popular paradigms in the space of cloud computing because of its unique ability to provide event-driven, short-lived, stateless function execution on containers without the need for infrastructure management. With the release of AWS Lambda in the year 2015, serverless computing quickly gained popularity in different types of applications such as machine learning[1], linear algebra[2] and high-performance computing[3]. The main reason behind this demand is its pay-per-use billing strategy, reduced management overhead of the cloud and fast delivery to market.

A key feature of serverless computing is providing infrastructure management for running serverless functions, this responsibility is completely handed over to the FaaS platform. At the moment, many open-source, as well as commercial FaaS platforms, are available where the applications are coded in the form of small granular-level functions. These functions are then packaged along with dependencies and are executed on FaaS platforms based on an external trigger. When these functions are invoked, the FaaS platform creates an execution environment to securely run the functions. Most of the platforms allow developers to write serverless functions in different programming languages such as NodeJS, Python, Java and Go providing the respective language-specific

runtime within the execution environment. However, there is also a limitation in terms of the languages supported wherein, some of the FaaS platforms allow developers to build container images and run them as functions.

Because of these features, FaaS has recently gained interest in the area of high-performance computing applications. Some of the research [4] [5] showed that FaaS can be used for data analytics application which led to the further analysis [6] [7] [8] for compute-intensive workloads depicting its feasibility. However, other studies [9] portrayed challenges involving the high cost of running such applications and maintaining the state of the serverless functions.

Most of the research conducted have evaluated the compute-intensive workloads on commercial serverless platforms such as AWS Lambda, Google Cloud Functions (GCF), and Azure Functions. Though these platforms provide all the functionalities required by developers to produce faster-to-market products, they do come with several drawbacks such as vendor lock-ins, unavailability of language runtimes, complex pricing models and restrictions on security.

However, open-source FaaS frameworks give programmers the option to create applications using a variety of programming languages, preventing vendor lock-ins[10]. Additionally, developers are free from the burden of adhering to restrictions on code execution time, code size and concurrency. Hence, the primary focus of this paper is one such open-source FaaS OpenWhisk. Also, for large scale FaaS infrastructure providers the pay-as-you-go model does not provide enough control over the billing which points to the requirement of open-source FaaS in the private cloud giving more control over the cost of serverless.

Kubernetes [1] is an open source platform for managing containerized workloads which supports both Enterprise and open-source FaaS providers[11][12][13][14] but has not yet been explored enough for private cloud.

Subsequently, this study fills the gap in research and benchmarks in the area of open source serverless computing. Apache OpenWhisk [2] is an open-source distributed Serverless platform which has not been explored much with Kubernetes container orchestrator on a private cloud for parallel workloads, which posts the research question.

Can OpenWhisk be deployed on Kubernetes cluster running on OpenStack to run parallel workloads for measuring speedup, response time and success rate?

The key contributions of this research paper are:

- Checking feasibility of deploying OpenWhisk on Kubernetes cluster deployed on NCI OpenStack.
- Gain insight into the underlying architecture OpenWhisk and how it runs functions with Kubernetes.
- Understanding methods to parallelize code using libraries for Java, C++ and Go.
- Gain insight into the execution times for running parallel workloads using OpenWhisk FaaS

The report is structured as follows section 2 describes and critically analysis related work done in the same field. Followed by section 3 explaining the research methodology

[1]Kubernetes : https://kubernetes.io/docs/home/

[2]OpenWhisk : `https://github.com/apache/openwhisk`

and specifications followed for completing the research. Finally, the results are discussed along with the conclusion and future work.

2 Literature review

Function-as-a-service plays a significant role in improving the performance of applications using cloud services and in reducing the cost of using them. In the following section, several related works is discussed and critically analysed.

2.1 Serverless Performance evaluations

Serverless initially construed as not using servers is sometimes considered misleading in terms of serverless computing as the functions with the application code runs on containers which are also partly physical machines or servers running on certain data centre. Cold start is a well-known problem within serverless computing[15] which represents the time taken for spinning up a new container when a warm container is not available to run a function. Several studies have attempted to discuss cold starts and lowering the start-up time to run the functions [16][17][18].

In [16] authors propose a performance based serverless architecture and compare it with existing platforms. The platform is developed with .Net using azure storage and its messaging layer. For managing and executing container service a worker service is implemented and for exposing platforms APIs a web service is developed. To benchmark their platform, they use concurrency tests which checks function invocation at scale. Their results show linear growth for upto 15 requests. AWS gives the highest throughput compared to other platforms and OpenWhisk gives the lowest throughput for up to 8 concurrent requests. Though their results discuss function expiration time, it does not broadly discuss networking, CPU, and different language runtimes available.

On the other hand, the effect of using different language runtime on performance is also conducted by few studies. The cost of function execution for different languages for AWS and Azure is done by [18]. They conclude that python is well suited for AWS Lambda and C# for azure functions. Whereas, in [19] the authors suggest that Nodejs or python runtimes are best for frontend functions as they would help in reducing latencies because these languages are less prone to cold starts than compiled languages compiled languages on the other hand give better performance for subsequent invocations.

In another attempt to reduce function start up latency, in [17] authors isolate applications within containers and functions within these applications. Due to this isolation the functions start up time reduces as the resources are allocated and deallocated faster. Another interesting area examined in this paper is of memory footprint of running concurrent functions. They make up to 50 concurrent calls to a python function showing a linear rise of memory in OpenWhisk and Greengrass whereas their platform adds an extra 1.1 MB in each call. Their research shows a way to reduce function interaction latency without considering the possibility that sandboxes could contend with each other for resources thereby increasing overall application latency.

2.2 Serverless and Parallel workloads

In terms of performance evaluation of serverless frameworks many research studies have been done exploring parallel workloads. In [20] Four major cloud platforms such as AWS, Azure ,Google Functions and OpenWhisk are studied comparing CPU performance, n/w bandwidth and throughput by parallelizing function invocation for distributed data processing. Their performance evaluation showed that AWS Lambda dominates elasticity for running concurrent function invocation. In general, serverless computing can be used for compute or data intensive workloads if the tasks are divided into small granular level tasks and run concurrently resulting in cost savings for running such workloads.

Similarly, in [21] authors propose a model to overcome issues with data exchange within serverless. The model uses information of the cluster configuration, storage and about the workload type to create the model. Other information such as workload graph is also used. For performance validation of the model, they have taken a traditional application which involves exchange of large amount data. They have completely transformed this application into serverless workload. The application used is Google's Tesseract Optical character recognition (OCR) engine. For translating this application into serverless workload they have used profiling tools specifically, valgrind and Callgrind. By using the computational graphs generated from the tools the application is split into small units and run as functions. Their results show that the performance of the data-intensive workloads can be increased by modifying the resource scheduling and deployment of the workloads. But they have not given an in-depth explanation of why parallelism did not perform so well for data-intensive workloads.

Another research where parallelism is explored is done by [22]. Authors have used a trigger-based orchestrators to schedule massively parallel workloads which follow fork join model. For this, they have used and compared existing orchestration services such as AWS steps functions, OpenWhisk Composer and Azure Durable functions. For benchmarking up to 320 parallel tasks are used. The outcome of the experiments shows a stable behaviour for AWS step functions, which is not the case for durable functions which shows inconsistent behavior after 20 concurrent tasks. OpenWhisk proves to be best choice for fork-join workflows as it shows smooth growth. But their results do not indicate the impact scheduling parallel tasks on such scale which could downgrade the performance of the application overall. In any case, their study throws light on the OpenWhisk composer and it's in-depth architecture.

By examining the architecture and parallel performance of four major cloud providers, the authors of [23] contend that most commercial FaaS platforms are not naturally suited for parallelism. By analysing the inherent design of these platforms, they have identified components such as function invocation, function management, virtualization which highly affect parallelism. Their experiments prove that different platforms follow different architecture which affects how parallelism is handled by them. Similar to the study [4] discussed earlier, they provide proof that AWS works well for parallelism; however, Azures performance is lower it inherently groups functions into fewer instances to improve resource utilisation. Their result provide evidence that resource scheduling and virtualization are the two most important factors influencing parallelism in FaaS platforms, a field that requires further investigation.

Similarly, [24] authors have explored parallelism in FaaS platforms with the difference of using parallel code within a container showing how the memory and CPU configuration

of the framework effects the performance of parallel code. They analyse data for different types of workloads in different language runtime on AWS, GCF and GCR. Similar to [21] they take existing applications and parallelize that code using profilers. There results clearly show that the platform users can save cost upto 80% based on the language runtime used and memory configuration of the container.

2.3 Open Source FaaS and Kubernetes

The majority of FaaS providers use virtualization with containers. These containers are needed to be managed based on the workload. There are many container orchestration platforms available, but Kubernetes [3] is the most widely used platform for orchestration. Hence, in the proposed study Kubernetes is used for orchestrating containers. Kubernetes has also been subjected to evaluation in some studies[13] [25]. Both these studies evaluate performance of containers managed by Kubernetes. Where authors in [25] explore commercial cloud platforms, in [13] its feasibility is checked on private cloud with bare metal setup. Both the studies show that there is almost no impact of using Kubernetes as orchestrator on the performance which usually depends on deployment chosen for the serverless setup.

Some studies have also extended Kubernetes towards edge computing. Given the heterogenous nature of edge devices authors in [14] and [12] show methods to improve the resource scheduling at the edge with the help of Kubernetes. The authors in [14] use the features provided by Kubernetes to define node capabilities based on workload which is used by kube-scheduler. The labels and selectors are pre-assigned before running the workloads. This does not work well with heterogenous nature of the edge devices as the workload type may change during runtime. Hence, [12] propose an advanced solution by replacing the Kubernetes default scheduler itself with a custom scheduler skippy. This helps in influencing and implementing a domain specific logic for scheduling which fits the criteria of edge devices. Both the studies have used labels and node selectors only for edge devices. Additionally, only OpenFaaS platform is considered for such scenario.

Mobile/IoT with serverless computing is explored by [26] where a combined architecture of edge and serverless computing is proposed. By creating a hybrid model, the decision of running a serverless function in local edge network or cloud network is made based on the history of the execution time of running functions. An edge proxy is presented between the serverless function and the requestor, this proxy analyses the functions historical data and then decides where the function will be executed. Their work presents the feasibility of OpenFaas with IoT environment.

Similarly, authors in [27] also merge serverless and edge by introducing a serverless-edge framework using OpenWhisk. Here a prototype is presented a FaaS platform is created with distributed fog network. Based on the current load, functions are offloaded to other nodes distributing the traffic. Their work identifies various bottlenecks in the setup related to latency, node sync up, computations but they fail to provide sufficient clarity on how functions are offloaded to other fog nodes. They describe a regional load balancer which performs this task but don't provide the details of how many load balancers are present. Nevertheless, the whole process of offloading adds extra latency to the application.

Some of the studies have addressed the issues within commercial cloud platforms and

[3]Kubernetes : https://kubernetes.io/docs/home/

evaluated of opensource frameworks. Authors in [10] evaluate the performance of 3 major open source serverless platforms namely Fission, Kubeless and OpenFaaS deployed on Kubernetes by characterizing the ratio of completed tasks and response time for different types of workloads. The main reason for choosing Kubernetes is because it is supported on all the platforms under test. For evaluation 2 types of experiments are used first, a simple function which returns a string. Second a compute intensive function. These experiments fail to evaluate the throughput of the platform under test. Furthermore, they fail to assess the performance of OpenWhisk which also supports Kubernetes as they could not bring up the set up.

Similarly, [28] also evaluated the performance of open source FaaS Kubeless, Knative, OpenFaas along with OpenWhisk on a single node Kubernetes cluster with multi-layer edge platform. They evaluated qualitative and quantitative metrics of the frameworks in the edge environment. Both [10] and [28] conclude that when Kubernetes is integrated with Kubeless it outperforms other platforms attributing to its simple architecture which uses native components of Kubernetes. Based on their outcomes it is evident that OpenWhisk is not well suited for edge computing. However, they have not assessed OpenWhisk with Kubernetes in a cloud only environment. Moreover, their setup is built on-premises using a single master-worker node leaving the area of private cloud unexplored.

In contrast, private cloud for open source FaaS is explored by [29]. The authors create a fault tolerant, highly available Kubernetes cluster on private cloud OpenStack and use OpenFaaS for triggering serverless functions. For benchmarking authors have used scaled down workloads traces from azure dataset. They classify the logistic relation of response time when concurrent functions are used.

OpenWhisks performance is measured by [30] by running different types of test functions and a server based application. To compare the performance of OpenWhisk they deploy this server-based application on Kubernetes cluster with VMs identical to OpenWhisk setup. The results help in identifying different bottlenecks of OpenWhisk. But they have only used OpenWhisk's default configuration setup which does not help in identifying the feasibility of OpenWhisk with Kubernetes.

2.4 Research Niche

Existing work discussed above shows that even though there is significant research done for serverless computing very few studies have evaluated the performance of Open-source frameworks. Most of the research is done with the focus on saving cost and memory utilization of commercial cloud providers for parallel workloads. Moreover, the compute intensive parallel workloads are split to run on different containers using container orchestration. Another research gap observed is that existing work have used open-source FaaS for edge devices with less resources rather than using VMs. OpenWhisk performance has been subject to very few studies with no research found relating to OpenWhisk deployed on Kubernetes cluster in a private cloud setup.

Hence, the proposed research evaluates OpenWhisk's feasibility on Kubernetes to run parallel workloads. This work addresses the gaps in existing research studies by parallelizing workloads using external libraries like OpenMP, Valgrind and running these workloads on NCI OpenStack using only open-source technologies such as OpenWhisk, Kubernetes, Prometheus, OpenStack, kubeadm, Grafana for 3 different languages C++, Java and Go.

3 Research Methodology and Specification

In this section, the research methodology used for this research is discussed. Figure 1 shows the steps followed for the evaluation. The first step is to create 4 OpenStack instances. Once the instances are created Kubernetes is installed using kubeadm with 1 master and 2 worker nodes. Then, OpenWhisk is installed on the master node as controller and worker nodes as invokers, the fourth node is used for metrics collection. Next step in the research is the code profiling. To evaluate the framework 3 different types of workloads are selected [31] [4] [5]. All 3 workloads are translated in C++, Java, Go. Next the C++ code is run through a profiler Valgrind to get the areas that can be parallelized. After these steps the code is run as actions in OpenWhisk. In the following sections the architecture of the framework is discussed followed by the explanation of serverless workloads used for evalaution in this work. Consequent subsections also discuss the language runtimes used for translating and parallelizing the workloads. Finally, the benchmarking workflow is described.

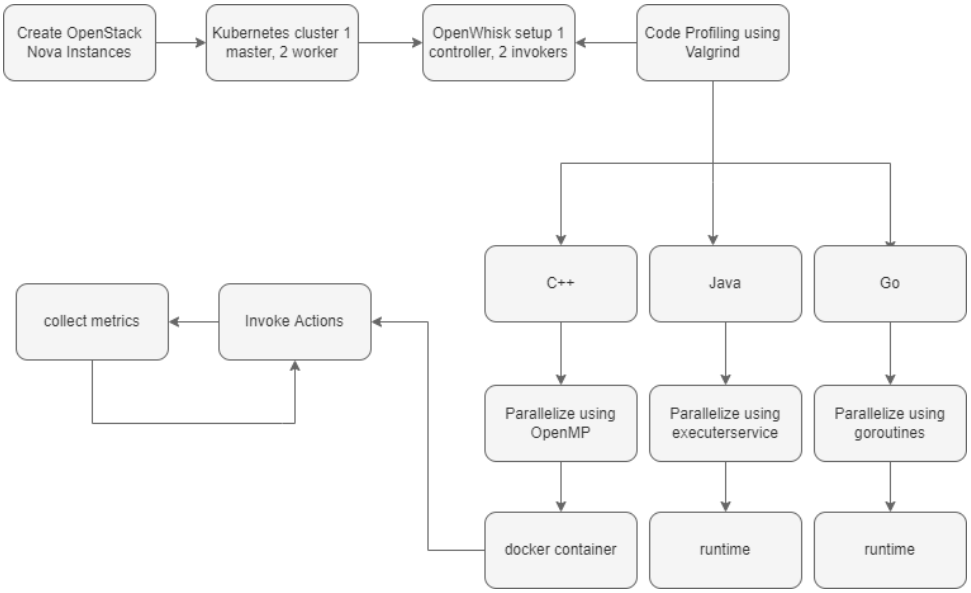

Figure 1: Process flow of the research

3.1 Proposed serverless architecture

A key contribution of this research is to check feasibility of OpenWhisk on Kubernetes on private cloud. Figure 2 shows the main components of proposed serverless architecture built on private cloud, NCI OpenStack [6]

3.1.1 OpenWhisk

OpenWhisk is a production ready open-source function as a service. It comes with many features such as fast runtime command-line tools, support for language runtime

[4]NP bench: `https://github.com/spcl/npbench`
[5]Pyperformance: `https://github.com/python/pyperformance`
[6]NCI OpenStack: `http://www.ncirl.ie`

in different languages such as JavaScript, Python, Swift, Java, PHP and Go. If a programmer needs to develop functions in a language other than the ones supported then they can bring their own Blackbox container. In this research both types of languages with runtimes and with Blackbox container are evaluated. OpenWhisk is built on top all opensource components nginx, controller, CouchDB, Kafka and Invoker 2. The entry point into the system is nginx, which transfers the HTTP calls to controller. The controller can identify what the user is trying to do. If it is a POST request it translates it into an action. The controller then checks with the CouchDB if the user is authorized to create/invoke the action. If the CouchDB authenticates the request. It is added to the CouchDB. For the action to be invoked the controller publishes a message to Kafka, if a response is received the action is invoked using Invoker and results are returned to the CouchDB 2.

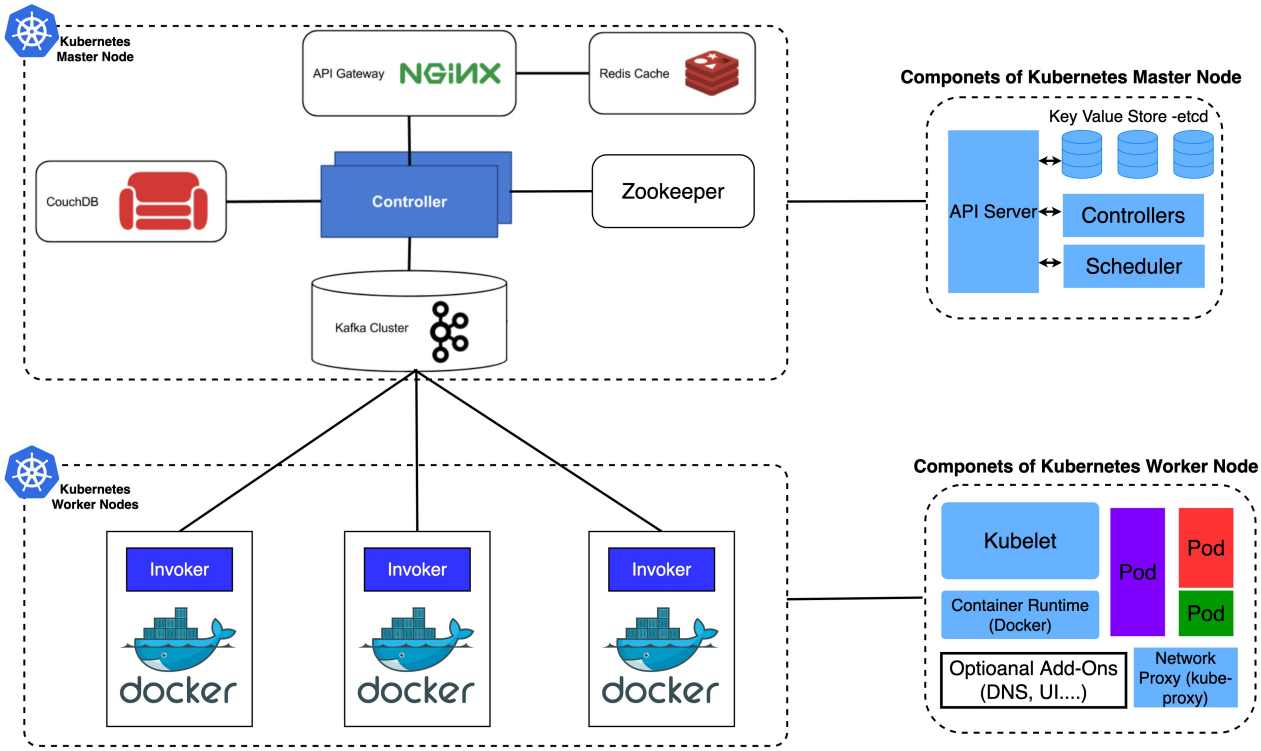

Figure 2: Proposed architecture of the framework

3.1.2 OpenWhisk on Kubernetes

In the serverless framework developed OpenWhisk is deployed on Kubernetes cluster, a Kubernetes cluster is created with its own key components to form a cluster. A master node is responsible for maintaining the cluster. It runs a kube-apiserver, a kube-scheduler responsible for tracking pod creation and selecting the right pod for function execution. kubeadm is used for bootstrapping the cluster with master and worker nodes. kubelet is responsible for starting the pods and containers and finally kubectl is the command line utility to interact with the cluster.

Once the Kubernetes cluster is ready OpenWhisk is deployed on top of this cluster. But there are certain traits that needed to be considered before selecting the right de-

ployment. The invoker is one of the main components of OpenWhisk. It is responsible for creating and managing the containers that execute the functions. For OpenWhisk to work in conjunction with Kubernetes there are 2 types of deployments supported DockerContainerFactory and KubernetesContainerFactory.

In the case of DockerContainerFactory, the invoker runs on every worker node, and it communicates directly with the docker daemon running on worker nodes to manage containers for running user functions. Though this model can be helpful in terms of latency for container management. it uses the default system and does not leverage kubernetes powerful resource management and security configurations. Another disadvantage is that it cannot be used if the underlying container engine is cri-o. Hence, in this research KubernetesContainerFactory is used wherein Kubernetes is responsible for creating, managing, and scheduling pods where functions are run, and invoker is in control of the cache of user containers. Another advantage is that it can be used with any container engine. The other 2 components used in the architecture are Prometheus and Grafana.

3.1.3 Prometheus and Grafana

To monitor the health of the cluster and to see time-series data Prometheus is used which is also an opensource tool. It will keep track of cloud metrics and report them to Grafana. Grafana is also an open-source tool which is used for visualizing the cloud metrics. It shows in its dashboard details of the functions run and its status. The detail of the cold starts and warm start and a pictorial representation of the functions can be obtained from these tools[7] [8] . These services are accessible from a different instance that is created on OpenStack with centos. To connect to these services a load balancer is created and the instance is added to the same subnet.

3.2 Workloads

The serverless workloads use a microbenchmark from NPBench. NPBench is a 3-clause BSD licenced benchmarking Suite for High-Performance NumPy 4[31]. The microbenchmark is called Atax which computes matrix and vector multiplication and stores the result into Atax matrix after multiplying it with Atax. Another benchmark used from the NPBench is Mandelbrot which implements an escape-time algorithm to spawn Mandelbrot sets.

The third serverless workload used is an application Monte Carlo used from pyperformance 5 which is licensed under MIT. Monte carlo simulations workload estimates the digits of pie. This is done using random number that are generated within a square. It uses these values to count all the points with distance to the centre of the square < 1. Then ratio of these points is calculated which is equal to pi /4 which can then be used to retrieve the value of pi. All these workloads are implemented in python by default. These are translated into three different languages. C++, Java, go runtimes that are invoked as functions in the experiments.

[7]Grafana: `https://github.com/grafana/grafana`

[8]prometheus: `https://prometheus.io/`

3.3 Language Runtime

To evaluate performance of parallelized functions for OpenWhisk on Kubernetes 3 languages are chosen i.e., C++, Java and Golang. C++ is chosen as it is one of most widely used language in scientific high-performance computing applications. However, OpenWhisk does not support C++ by default. To run the C++ code as functions on OpenWhisk a docker Blackbox container is created with all the dependencies. Then this docker image is uploaded on DockerHub. While invoking the function the image is passed as argument.

Go is used because it is supported by default by OpenWhisk and can be run directly without having to create docker Blackbox container. Also, Go was created with concurrency, hence it makes parallelization simpler. Finally, Java is used for to its wide popularity, its design and default runtime support by OpenWhisk.

3.4 Parallelization

To utilise parallelism for workloads discussed above in different languages first the areas in the code are identified using profiling tool such as Valgrind and Callgraph[21]. Valgrind provides tools that can be used to automatically detect memory management [9]. First, the code is run through Valgrind which generates PID extensions which is then passed through Callgrind. Callgrind generates callgraphs used for analysing the code that can be parallelized. In case of above workloads, for loops are indicated as areas that could exploit parallelism. Once these regions are isolated additional libraries or features provided by languages chosen are used for parallelization. For C++ code, OpenMP is used. its a common library for parallel programming in shared memory architectures[32] to ease the development of HPC applications and programs [33]. With OpenMP developers use pragmas to split an area of code into multiple threads.

In the code implemented in this research the areas to be run in parallel are encapsulated within these pragmas. First the number of cores available in the cluster is checked. Based on the value of cores present, the tasks are split into multiple threads in a fork-join manner. A single thread aka master thread spawns the threads and once the work is finished threads join back into master thread. For Go programs goroutines are used which are lightweight threads provided by the language and for Java ExecuterService is used. OpenMp is the best option for parallel code here as it automatically splits the tasks into threads whereas for goroutines and executerservice this information is provided explicitly.

3.5 Experimental Setup and Workflow

The serverless architecture is built using 3 m1.Large Nova virtual machines and 1 m1.small VM on the NCI OpenStack which is a private cloud 6. Each m1.Large machine is configured with 4 vCPUs, 8GB RAM and 80GB of memory with Ubuntu-18.04-x86_64 image. These 3 Vms are configured as 1 master and 2 worker nodes. All the nodes have docker 20.10.17 and Kubernetes v1.24.3 is installed for container management which is installed using kubadm v1.24.3. Kubectl is installed as command line interface. The workloads are implemented in C++ 11, Java 11, Go 1.13 languages. To deploy these as serveless functions OpenWhisk 1.2.0 is which is deployed using Helm v3.9.2. OpenWhisk comes with following by default: wsk cli 1.2.0, couchDB 2.3.1, Kafka 2.3.1, Nginx 1.21.1, Redis

[9] valgrind:`https://valgrind.org/`

4.0.14, Zookeeper 3.4.14. Grafana 6.3.0 and Prometheus 2.14.0 are installed explicitly for interacting with cluster and for monitoring. Once the above setup is ready. The code

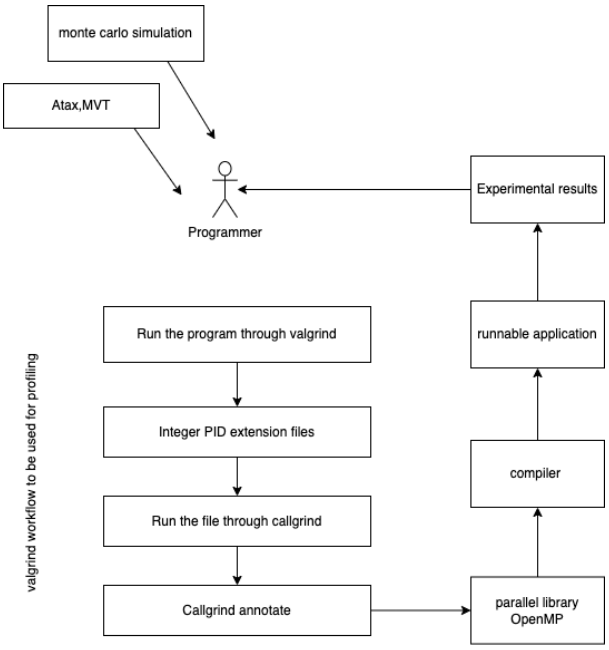

Figure 3: Steps followed for code profiling

written in C++ is packaged as docker container and deployed as actions. But before the functions are deployed all of the code is parallelized. Figure 3 shows the steps involved in parallelizing the code which is an extension of the methods suggested by structural parallel application development by [32]. The next step is running these functions as wsk actions. For the evaluation each workload is run 10 times first for serial code and 10 times for parallel code. The inputs are recorded using log from standard output and wsk list. Metrics related to success/failure, or any errors are collected from Prometheus and Grafana. The same steps are repeated for other 2 language runtimes as well with difference that Java and Go runtime is supported by OpenWhisk. Although, each runtime requires different configuration and parameters.

4 Evaluation

The key evaluation metrics in this research are the impacts of using different languages on OpenWhisk, speed up achieved by parallelizing the code, Impact of cold starts and success rate of running workloads.

4.1 Impact of language

One of the chief findings while performing experiments is for C++. As C++ runtime is not directly supported, the code is first shipped as a docker container which is then downloaded while creating action. As this Blackbox container is downloaded from the repository and run as an action, it is taking lot of time which results in the action command timing out resulting in developer error. From the logs, it is observed that the code performed the computation for all three workloads, but it is not able to exit

gracefully due to timeout. Another potential reason for this could be the use of OpenMP which is an external library included in the docker file at compilation. While running the C++ workloads it is observed that OpenWhisk/dockerskeleton supports Alpine docker which does not support libstd++.so and other libraries. It is highly possible that OpenMP is also not supported directly which results in developer error.

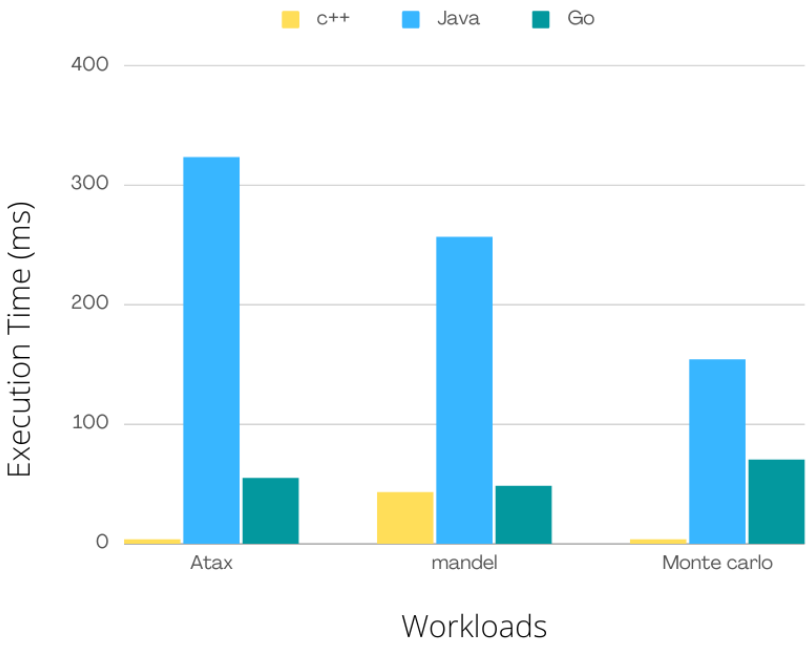

Figure 4: Execution times captured for different workloads

As the runtimes of Go and Java are supported by OpenWhisk actions run seamlessly in these cases. Implementation done in C++ and Go outperforms Java as can be seen Figure 4. The poor performance of java can be attributed to the way Java handles multithreading. As the number of application threads and communication increase performance of threads decreases. In case of Go, the performance is better than Java due to the built-in support of threading in Go language. C++ on the other hand has the lowest execution time as per logs.

4.2 Speedup achieved

One of the main goals of the research was to understand the speedup that is achieved by parallelizing the compute-intensive workloads. Figure 5 depicts the average speedup achieved by parallelizing the atax, monte carlo simulations and Mandelbrot set workloads on Openwhisk. The average speedup is calculated by taking the mean of the run time of running serial code and dividing it by mean of run time of running parallel code of each workload. As mentioned in the earlier section the setup is composed of 4 vCPUs(cores) which are used by the OpenMP, and other parallel libraries used in the code. When the code is run in parallel the workload is distributed across these 4 cores each running in parallel and sending the result back to main thread. Whereas serial code runs the code sequentially on 1 core.

Figure 5 shows the average speedup of small and medium types of workloads for Atax, mandle, montecarlo in three different languages. Small depicting matrix multiplication of 1000 rows and 1500 columns. Medium has the values 2000 and 2500 for rows and columns respectively. For Large configuration row = 20000 and col = 22000, in most of the cases a timeout is observed or out of memory error is observed. Hence, those results are not included in the graphs. From the figure, it is clear that for C++ there is an average speedup of approximately 2 times. Whereas in case of java there is not much difference observed by parallelising the code. In case of Go average speed is 1.5. The following results are captured for 4 vCPUs and a maximum memory of 256MB per container. For a lesser memory configuration of 128MB the results show almost no difference in speedup with parallel code.

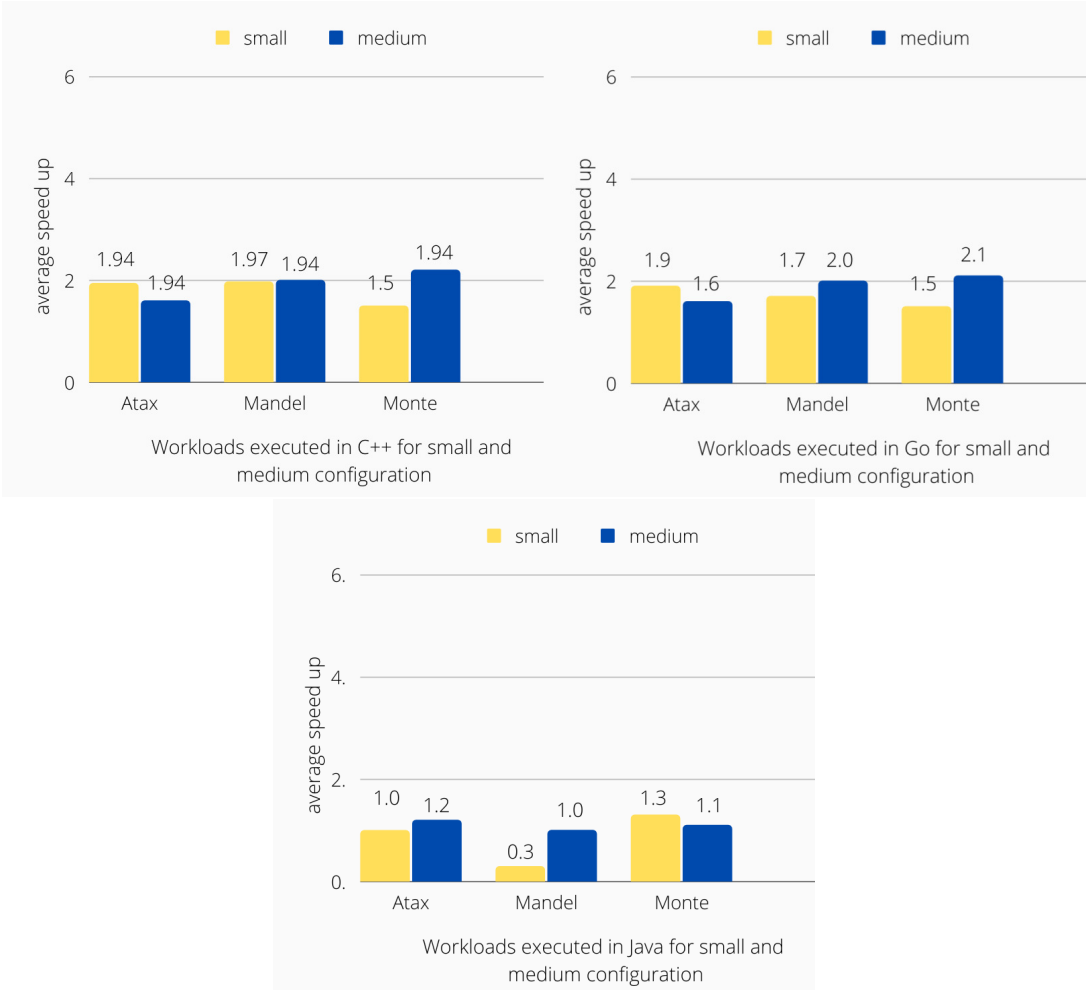

Figure 5: Average speedup achieved by parallel code

4.3 Impact of cold starts

The evaluations done also show the impact of cold start on parallel code. From the experiments, it is observed that cold start time is long for both parallel as well as serial code. This could affect the total time of running a parallel application impacting its performance. The cold start of Java code is the highest. The reason behind this could be

the time required to bring up Java virtual machine. Also, for C++ the cold start time is high attributing to the Blackbox container that is used in this case also to install all the dependencies that are installed after downloading container. Once the container is warm the execution time is observed to be reduced by almost 80% compared to cold start.

4.4 Success rate

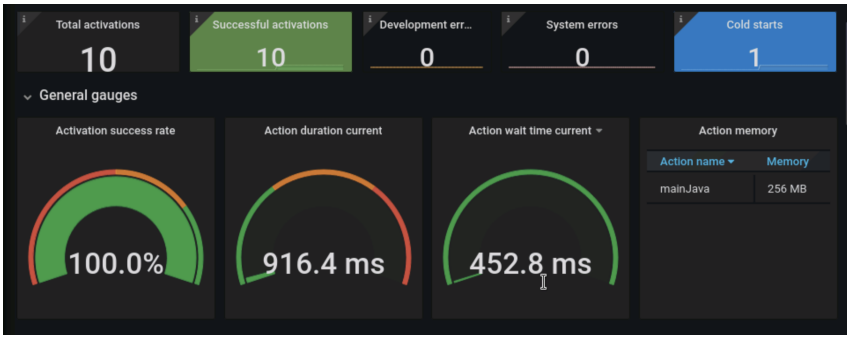

Figure 6: Graphana dashboard showing successful activation

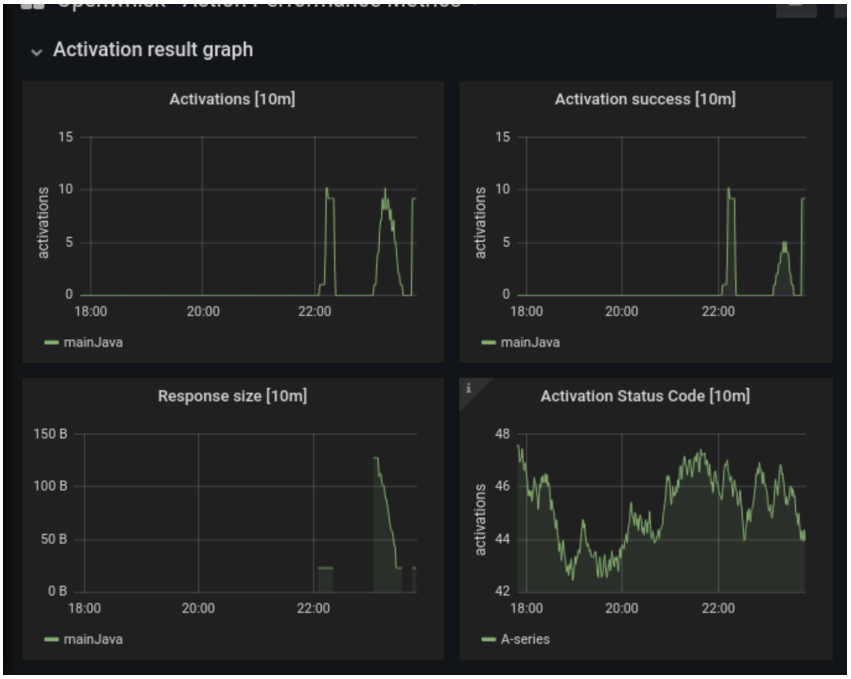

Figure 7: Graphana dashboard depicting graphs of response rate

Figure 6 screenshot is capture of graphana dashboard showing the success rate of running the workloads overall. Except C++ code where the Openwhisk shows a developer error in cold start and Internal error in warm start the application runs seamlessly for Java and Go without any errors. Figure 7 also shows the number of successful functions application has executed and response rate.

4.5 Discussion

The serverless framework developed shows the feasibility of OpenWhisk on Kubernetes in a private cloud setup. Experiments performed showed that OpenWhisk works well with Kubernetes on NCI OpenStack. The main experiments done were to understand the impact of parallel workloads for different language runtimes. It is observed that for a configuration with 4vCPUs i.e., 4 cores and 256MB used by the parallel code an average speed up of 2 overall is achieved compared to serial code using one core. For a larger setup with 16vCPUs more speed up can be achieved resulting in increase in performance of compute-intensive workload.

Another observation is that OpenWhisk does not perform well for the language runtimes which are not directly supported resulting in developer errors and internal errors. For other languages, the code works well for compute-intensive workloads for lower configurations. For example, in the case of Atax workload if the configuration is 20000 * 22000 the actions fail 90% of the time. A reason behind this could be the lower memory available per container. For a higher memory of 2048MB per container, the results can be improved. An observation of the results is that even though there are 4vCPUs available to use for parallel code, its performance also depends on the maximum memory allocated per container to execute the functions. This works well for simple serverless functions but for a high-performance application more memory per container is needed to exploit parallelism.

5 Conclusion and Future Work

This research studies the suitability of open source serverless offering OpenWhisk on a private cloud and gains insight into its underlying system. The experiments show the effect of parallel code for compute-intensive workloads that run within a container instance and how it can help in the improvement of performance for such workloads. It is observed that compute-intensive workload performs better with parallel code depending on the type of workload and language runtime used. Performance degradation for some language runtime is attributed to the underlying implementation of the threads. Additionally, It is also observed that the cold start latency is almost equal for both parallel and serial code. The limitation of this research is in terms of the cluster size. A larger cluster size with higher memory and CPU could provide a better insight into the larger size of workloads and provide a better comprehensive understanding of speedup. Another area for future research is to study other language runtimes provided by OpenWhisk for comparing their performance with the runtimes used in this research.

References

[1] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ml workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 13–24, 2019. CORE2021 Rank: National: India.

[2] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 281–295, 2020. CORE2021 Rank: National: India.

[3] P. Grzesik and D. Mrozek, "Serverless nanopore basecalling with aws lambda," in *International Conference on Computational Science*, pp. 578–586, Springer, 2021. CORE2021 Rank: A.

[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99Symposium on Cloud Computing, SoCC '17, (New York, NY, USA), p. 445–451, Association for Computing Machinery, 2017. CORE2021 Rank: National:India.

[5] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow:{Low-Latency} video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 363–376, 2017. CORE2021 Rank: A*.

[6] V. Giménez-Alventosa, G. Moltó, and M. Caballer, "A framework and a performance assessment for serverless mapreduce on aws lambda," *Future Generation Computer Systems*, vol. 97, pp. 259–274, 2019. JCR Impact Factor 2021: 7.187.

[7] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, pp. 41–54, 2019. CORE2021 Rank: A.

[8] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "Numpywren: Serverless linear algebra," *arXiv preprint arXiv:1810.09679*, 2018. CORE2021 Rank: National:India.

[9] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *arXiv preprint arXiv:1812.03651*, 2018. CORE2021 Rank: A.

[10] S. K. Mohanty, G. Premsankar, M. Di Francesco, *et al.*, "An evaluation of open source serverless computing frameworks.," in *CloudCom*, pp. 115–120, 2018. CORE2021 Rank: C.

[11] 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)*, pp. 1416–1421, 2019. CORE2021 Rank: NA.

[12] T. Rausch, A. Rashed, and S. Dustdar, "Optimized container scheduling for data-intensive serverless edge computing," *Future Generation Computer Systems*, vol. 114, pp. 259–271, 2021. JCR Impact Factor 2021: 7.187.

[13] V. Medel, R. Tolosana-Calasanz, J. Ángel Bañares, U. Arronategui, and O. F. Rana, "Characterising resource management performance in kubernetes," *Computers Electrical Engineering*, vol. 68, pp. 286–297, 2018. JCR Impact Factor 2021: 3.818.

[14] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–7, 2018. CORE2021 Rank: B.

[15] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*, pp. 1–20. Singapore: Springer Singapore, 2017.

[16] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, 2017. CORE2021 Rank: NA.

[17] 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)*, (Boston, MA), pp. 923–935, USENIX Association, July 2018. CORE2021 Rank: A.

[18] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 154–160, 2018.

[19] D. Bardsley, L. Ryan, and J. Howard, "Serverless performance and optimization strategies," in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 19–26, 2018. CORE2021 Rank: NA.

[20] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 442–450, 2018. CORE2021 Rank: B.

[21] A. M. Nestorov, J. Polo, C. Misale, D. Carrera, and A. S. Youssef, "Performance evaluation of data-centric workloads in serverless environments," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 491–496, 2021. CORE2021 Rank: B.

[22] D. Barcelona-Pons, P. García-López, A. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, "Faas orchestration of parallel workloads," in *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, (New York, NY, USA), p. 25–30, Association for Computing Machinery, 2019. CORE2021 Rank: A.

[23] D. Barcelona-Pons and P. García-López, "Benchmarking parallelism in faas platforms," *Future Generation Computer Systems*, vol. 124, pp. 268–284, 2021. JCR Impact Factor 2021: 7.187.

[24] M. Kiener, M. Chadha, and M. Gerndt, "Towards demystifying intra-function parallelism in serverless computing," in *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, WoSC '21, (New York, NY, USA), p. 42–49, Association for Computing Machinery, 2021. CORE2021 Rank: A.

[25] A. Pereira Ferreira and R. Sinnott, "A performance evaluation of containers running on managed kubernetes services," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 199–208, 2019. CORE2021 Rank: C.

[26] D. Pinto, J. P. Dias, and H. Sereno Ferreira, "Dynamic allocation of serverless functions in iot environments," in *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 1–8, 2018. CORE2021 Rank: C.

[27] L. Baresi and D. Filgueira Mendonça, "Towards a serverless platform for edge computing," in *2019 IEEE International Conference on Fog Computing (ICFC)*, pp. 1–10, 2019. CORE2021 Rank: NA.

[28] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642-939X, pp. 206–211, 2019. CORE2021 Rank: B.

[29] H. Govind and H. González–Vélez, "Benchmarking serverless workloads on kubernetes," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 704–712, 2021. CORE2021 Rank: A.

[30] A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, "A distributed analysis and benchmarking framework for apache openwhisk serverless platform," in *Proceedings of the 19th International Middleware Conference (Posters)*, Middleware '18, (New York, NY, USA), p. 3–4, Association for Computing Machinery, 2018. CORE2021 Rank: A.

[31] A. N. Ziogas, T. Ben-Nun, T. Schneider, and T. Hoefler, "Npbench: A benchmarking suite for high-performance numpy," in *Proceedings of the ACM International Conference on Supercomputing*, pp. 63–74, 2021. CORE2021 Rank: A.

[32] M. Danelutto, G. Mencagli, M. Torquati, H. González-Vélez, and P. Kilpatrick, "Algorithmic skeletons and parallel design patterns in mainstream parallel programming," *International Journal of Parallel Programming*, vol. 49, no. 2, pp. 177–198, 2021. JCR Impact Factor 2021: 1.382.

[33] J. Diaz, C. Muñoz-Caro, and A. Niño, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1369–1386, 2012. JCR Impact Factor 2021: 2.687.