Experimental Evaluation of Serverless Functions

Pranjal Gupta , Shreesha Addala
University of Waterloo
{p43gupta, saddala}@uwaterloo.ca

Abstract

There has been a rapid increase in the adoption of serverless functions in the industry. Typical use cases range from building single page web applications to processing Internet of Things (IoT) based events. In this work, we evaluate the performance of serverless functions with respect to network, memory and I/O overhead. Specifically, we evaluate functions built by the OpenFaas framework. Additionally, we introduce a serverless implementation of the MapReduce framework. Our results show that the performance of a function is dependent on more than just the task being performed.

1 Introduction

Cloud services have revolutionized how organizations manage applications, but many companies still view their systems in terms of servers, even though they no longer work with physical servers. A *serverless* approach is when we take away the concept of servers, and begin to think of back-end applications as workflow, distributed logic, and externally managed data stores. *Serverless* means back-end application logic is still written by the developer, but unlike traditional architectures, it's run in stateless containers that are event-triggered, and ephemeral. Despite the name, it does not actually involve running code without servers. The name *serverless* is used because the organization that owns the system does not have to purchase, or provision servers for the back-end code to run on, thereby providing operational cost savings.

1.1 Containers and Kubernetes

Containers are an operating-system-level virtualization technique for running multiple isolated systems on a host using a single kernel. At their core, containers are a way of packaging software, such that we know exactly how they will run. There are no unexpected errors when we move it to a new machine. An application's code, libraries, and dependencies are packed together in the container as an immutable artifact. Running a container is like running a virtual machine, without the overhead of spinning up an entire operating system. For this reason, bundling your application in a container versus a virtual machine will significantly improve startup time. We use Docker [1] for creating containers in our experiments.

Kubernetes [2] on the other hand is a platform for managing containers, facilitating both configuration and automation. It abstracts deployment, monitoring and scaling of containers. Cluster groups together hosts running containers, and Kubernetes helps you easily and efficiently manage these clusters by defining and maintaining the policies for scheduling, auto-scaling and grouping containers. Kubernetes has a master node and multiple slave nodes that run the containers. The platform also introduces the concept of pods which are a logical group of containers that are tightly coupled and dependant on one other for servicing requests. In our work, we have a single container that is executed in isolation - however, we keep the term pod for the rest of the paper.

1.2 OpenFaas [3]

OpenFaas is a well-known open-source framework for building serverless functions over Docker and Kubernetes (or Docker Swarm). OpenFaas builds up using three core components: API Gateway, Function Watchdog and Prometheus.

- *API Gateway* - Exposes the API endpoints to the hosted functions, and scales the functions according to the request demand by altering the service replica count in the Kubernetes API.
- *Function Watchdog* - A lightweight Golang HTTP server acting as an entry point for HTTP requests to be forwarded to the target function via STDIN. The response is sent back to the caller by writing to STDOUT from the function.
- *Prometheus* - It is responsible for metrics collection (invocation count, requests/second), and to trigger the API gateway to auto-scale.

The OpenFaas wraps the function into Docker containers that are then used for executing on the nodes, being managed by Kubernetes.

1.3 Contributions

Our objective and contributions of our work is two-fold:

1. We investigate our setup of serverless to evaluate if there is significant overhead in network, processing and I/O

when using serverless function built using OpenFaas. For a given workload, we perform a series of experiments to answer these question that help us determine performance is affected depending on the number of nodes, number of pods in a node and cold start or warm running of a container. We also vary the workload for few workloads. Specifically, our experiments focus on the latency of response, speed of the computation, bandwidth and scalability.

2. We propose a MapReduce implementation similar in spirit to the Google's MapReduce [5] and Apache Hadoop [4]. Our implementation is inspired by an article by Amazon [12] in the sense we have a set of functions (mappers, reducers, coordinators) that we chain together to perform the required computation. We take perform primary experiments on our implementation and try to give reasoning about the costs of the operations.

2 Related Work

Joseph *et* al. [7] investigate the first-generation of serverless compute offerings in the public cloud services. The authors believe that serverless offerings are a bad fit for large data systems workloads due to limited lifetimes, I/O bottlenecks, and data communication through slow storage. The authors point out that perhaps the biggest architectural shortcoming of functions is that they run on isolated VMs, separate from data. In addition, serverless functions are short-lived and non-addressable, so their ability to cache state internally to service repeated requests is limited. [9] present the use of serverless function for analyses of big biomedical data. The authors propose that serverless computing has tremendous potential for biomedical research in terms of ease-of-use, instantaneous scalability and cost effectiveness.

While the traditional use case for serverless function continues to focus on building micro-services and event-driven processing pipelines, Youngbin *et* al. [8] examine the possibility of analytical processing on big data. The work presents Flint, a prototype Spark execution engine that uses AWS Lambda to provide a pay-as-you-go cost model. With Flint, a developer uses PySpark without needing an actual Spark cluster. For their experimental setup, Flint provides better performance at a lower cost when compared to PySpark and Spark. Qubole [11] specializes in building big data system on serverless platforms. In 2017, they presented an implementation of Spark, and on-demand ETL pipelines using AWS Lambda.

3 Experimental Setup

All experiments were performed on a cluster provided by Data System Group at the University of Waterloo. Hardware specifications of each node and software versions can be seen in Table 1. Every node is allotted the default memory and CPU limits. Our environment for experiments has a master node and up to 6 homogeneous slave nodes. To find the overheads, the task we are performing is implemented as an OpenFaas function using python, timed and compared with the same implementation using script in the same language locally on one of the cluster nodes not involved in the environment. We ignore network costs because it is a negligible overhead (0.3 milliseconds). we repeat each experiment 100 times and report the average (and, standard deviation) at all places.

Cluster Description	
Attribute	Value
Memory	16GB
CPU	6-core Intel Xeon E5-2620 @ 2.10 Ghz
Ethernet	Mellanox MT27500 ConnectX
NFS	1TB
Docker version	18.09.2
Kubernetes version	1.14
features	
Openfaas version	0.8.6

Table 1: Data description

3.1 Ping and Compute

The experiments for Ping and Compute have a similar setup. A client function invocation is first processed by the Kubernetes master, and then assigned to one of the slave nodes (where the relevant pod is spawned) for execution.

The ping experiment is a simple network request, with no computation in the function. The response from the slave node is sent directly to the client. This test is meant to find the overhead due to the cost incurred at Kubernetes master for internal processing and redirection to one of the slaves. For ping, we measure times in two setups:

- **Cold-start**: When the pod is not spawned in one of the slave nodes. In this setup, when the request is received by the Kubernetes master, the pod is made to run and likewise the internal routing table is changed, before the request actually is begun to be catered.
- **Warm-start**: When the pod is already up and running on at least one of the slave nodes.

The compute operation, finds the largest prime number less than or equal to given input. The different values of n were: 1, 1000, 5000, 1000, 50000, 100000 and 500000. The objective here was to see if there is an increase in overhead with increase in computation time.

Figure 5 presents the setup for ping and compute experiments.

3.2 Scalability

The objective for the scalability test is to find the performance of OpenFaas when the number of requests is far greater than the available containers. The experimental setup is similar to ping and compute, with 200 requests being concurrently fired to the Kubernetes master. We performed the experiments on two variants:

- **Fixed number of pods (p) already exist in the cluster** - To test the response time of the slowest of the 200 requests and get an idea of the level of concurrency

achieved with varying p and also the number of slave nodes.

- **Auto-scaling** - To capture the response of the setup to the increasing requests w.r.t efficiently scaling for dealing with peak requests.

The experiment used a similar setups and routing pipelines as that of ping and compute.

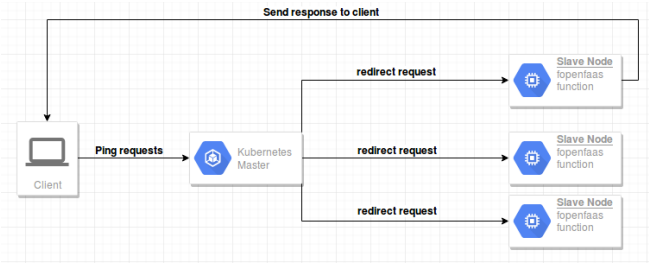

Figure 1: Experimental Setup for Ping, Compute and Scale tests

3.3 I/O

The setup for performing the I/O is similar to ping and test, with an addition of NFS storage, which is where the the input files are stored. The request from the client is received by the Kubernetes Master, which is then assigned to one of the slave nodes. The OpenFaas function has details about the directories of interest, how it can instantiate a SFTP connection with the NFS host. We implemented a function to find the number of words in a text files of size 10MB, 25MB, 50MB and 100MB. Figure. 2 visualizes the setup for the IO operation.

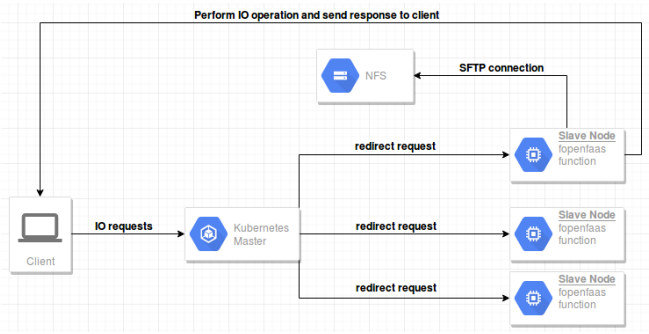

Figure 2: Experimental Setup for IO

3.4 Serverless MapReduce Framework

The MapReduce is a framework that enables parallel and distributed processing on large datasets. The core components of MapReduce are the master, set of mappers and set of reducers. The master keeps track of all mappers, reducers, and their state of operation and state of execution. A mapper task is the first phase of processing that processes each input record (from the NFS) and generates an intermediate key-value pair. Traditionally, the key-value pair is stored on local disk, but in our implementation, the intermediate output is stored in the NFS storage or returned to the master node. The reducer takes the output of multiple mappers, to generate the output. A typical MapReduce implementation will partition function (sort, hash, etc) mapper's output before the reducer processes the output.

For our implementation of a serverless MapReduce framework, we have a function imitating the MapReduce master, called Coordinator. This node is responsible for keeping track of the execution states of concurrent mappers and reducers. If any mapper or reducer fails, their task will be re-assigned by invoking the required function with same set of parameters. Progress of overall execution and the input file that has to be re-processed is tracked by the Coordinator - hence, stateful. This behaviour and long execution time of the Coordinator is very *unserverless-like* because serverless functions typically hold no state and are short-lived (maximum of 15 minutes for AWS Lambda). The Coordinator is responsible for informing the mappers about their input files. The mappers are concurrently executed on multiple pods on preferably different slaves. Once a mapper finishes processing the input file, it's result is returned to the master or stored the in the NFS storage. When all the mappers finish their execution, the master will spawn the reducers, passing the data or pointers to the data. Once all reducers finish execution, the process is complete. Figure. 3 presents our Serverless MapReduce model along with the sequence of various micro operations.

We experimented with 2 simple variants of the MapReduce, that are much reduced in capabilities than MapReduce in its original form. However, the aim of this paper is to perform experiments rather than building a complete end-to-end system. Our implementation is more about playing with the possiblities. The two variants are:

- **OnlyMappers** In this version, we only have a dedicated mapper function that operates on the data. The final consolidation of the result is done at the Coordinator only.
- **MappersAndSimpleReducers** In this version, we have a dedicated mapper functions that operates on the data and also a set of reducers that operate on the data in levels. However, our mappers do not store the intermediary files to any storage, but return it to the coordinator to be passed to the reducers iteratively.

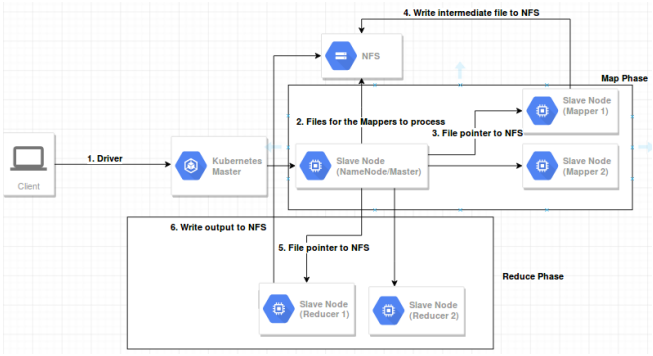

Figure 3: Serverless MapReduce Framework

4 Results and Inferences

We describe the results for the experiments followed by a discussion about the insights we have obtained.

4.1 Ping

	ping-test on open-faas		ping-test on local	2RTT over TCP
	Cold Start	Hot Start		
Average	1824.238	79.327	21.965	0.277
Std. Dev.	42.238	1.706	1.530	0.032

Figure 4: Results for Ping experiment

Figure 4 shows the results for the two variants of experiments we performed on the ping along with the stats for executing the identical procedure locally. We can infer the fact that there is an overhead of about 55 ms associated with executing over the OpenFaas environment. We suspect this cost to be incurred at the Kubernetes master for book-keeping and routing over to one of the slaves.

The cost for the case when the pod is not spawned is even greater as expected (possibly due to the reasons stated above, among others) and comes out to be around 1850 ms. This cost is aligned (or a bit on a higher end) to the ones we found reported on the web. We also see that the standard deviation for cold start is greater than that of Warm start, but it is not too significantly high (given approx. 23x increase in average) to be investigated.

4.2 Compute

Figure 5 shows the plot of the increasing overhead with the increase in the computation that is done inside the function hosted in our environment. The x-axis of the graph gives the value of n, the size of our computation, while the left x-axis gives the overhead time in ms. The reference for calculating the overhead is identical function executed locally. Also in the graph is a reference plot of n-squared with axis on the right side.

Analyzing the graph, we make out that the overhead grows slower than the n-squared but a little faster than the linear. This is indeed in align to the amortized complexity of our algorithm for calculating the primes one at a time till n. To infer about the overhead, it seems that there some cost of computing inside the container than directly in the default userspace of the node. A possible reason for this could be the resource sharing scheme at play on the slave node because of which the container gets a reduced unequal share of the system resources. This reason is further confirmed by a similar set of overheads we get in our I/O test as well.

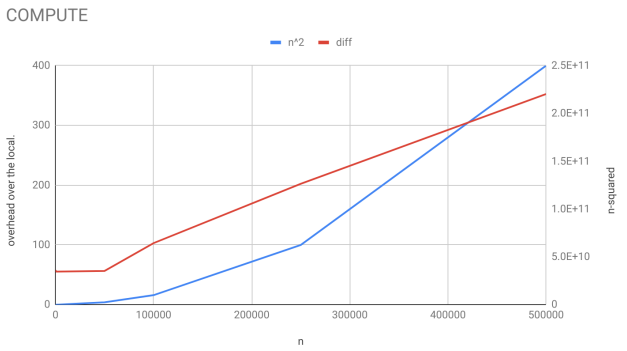

Figure 5: Overhead increases linearly w.r.t input

4.3 Scalability

We summarize the numbers we get from our test for scalability on our setup. We noted down the number of requests rejected and the worst response time for catering 200 simultaneous requests while varying the number of running pods and the number of slaves in our setup. For testing the auto-scaling feature, we didn't bound the number of pods and let it to scale organically while noting the time span in which our cluster expanded to the maximum capacity required. We define the complete concurrency as the mean time taken by the requests to be served when there is a single pod on a slave node, serving to just one request.

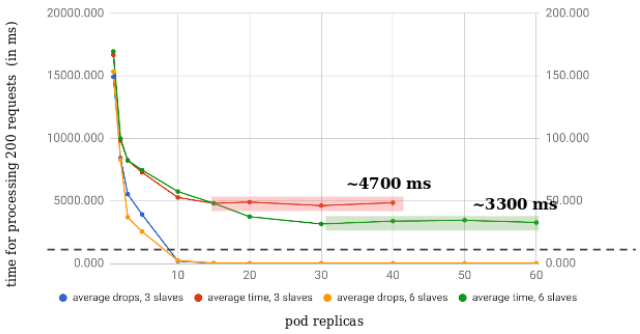

Figure 6: Time taken to server 200 requests for fixed pods

Figure 6 shows as we increase the number of pods, we decrease the number of requests that were getting rejected in our setup. However, increasing the pods doesn't actually help us achieve the complete concurrency since after a point, pods on the slave node starts fighting for the system resources and hence, thrashing becomes the reason for the overhead in serving requests. The problem can be mediated by having more slaves in the setup, which leads to the decrease in the value of pods-per-slave and hence decrease the overhead, thereby improving the worst response time.

Figure 7 is the evaluation of the auto-scaling policy of our setup. The policy is highly customizable to define the minimum replicas (min_replicas), maximum replicas (max_replicas) and the frequency (freq) of scaling in a setup. Openfaas records the requests-per-second metric for

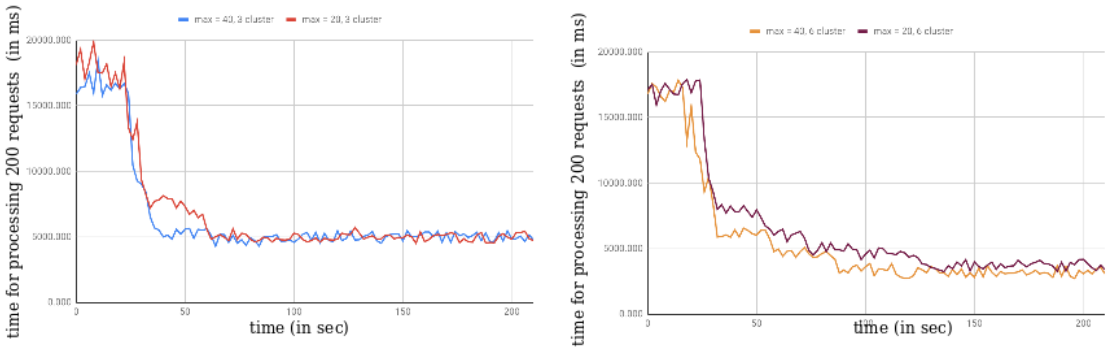

Figure 7: Auto-scaling with 3 slave and 6 slave nodes

each hosted function which when exceeds the predefined threshold leads to spawning more pods till the number meets 'max_replica'. The scaling is based on trigger which can be controlled by freq. On each trigger, the number of new replicas spawned is 20% of max_replica. We run 4 set of experiments by varying max_replica (20 and 40) and number of slave nodes.

From our results of auto-scaling, we figure out that the characteristic curve (stepped hyperbola) of figure 7 can be tuned by controlling 3 variable in the system.

- The worst response time of the setup is the inverse proportional function of the number of nodes we have since this will help us scale the number of pods without actually running into thrashing.
- The smaller the freq in our setup the greater is the vertical slope of the characteristic curve. This means that smaller is the time to reach stability in light of an unanticipated load.
- The value of the max_replica determines the number of scaling steps the setup takes to scale to a stable size.

Our finding is depicted in Figure. 8.

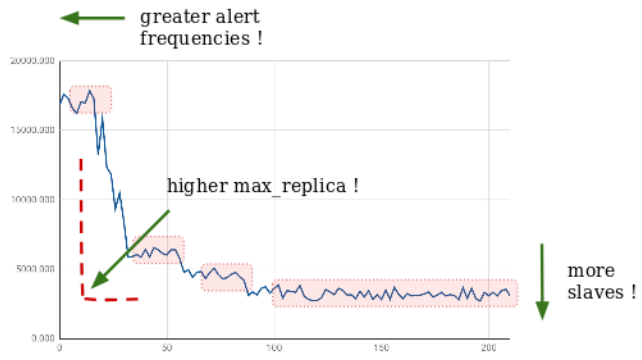

Figure 8: Increasing the maximum replicas, number of slaves and frequency of alerts will provide better performance

4.4 I/O

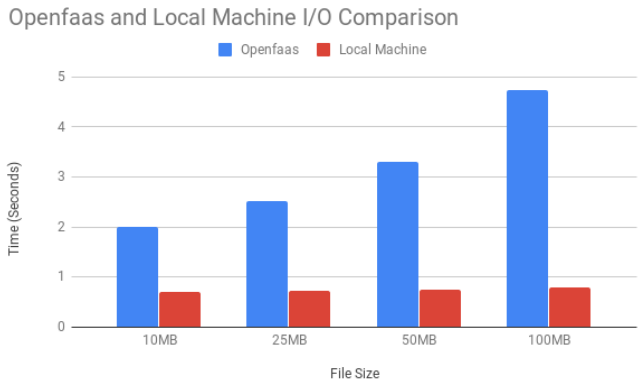

Figure 9: I/O Results. Significant increase in overhead w.r.t file size

The results for the I/O experiment are presented in Figure. 9. There is significant overhead when compared to running the same task on a local machine, but it is surprising to see the increase in overhead as the file size increases. We contemplate that the reason for this observation is because the containers are isolated processes running in the kernel space, competing not just for CPU and memory, but also kernel resources. Similar observations have been reported on Github and technical blogs [10][6]. These posts have specifically mentioned kernel commands such as *__d_lookup_loop* and *posix_fadvise()* causing these issues. To come up with concrete conclusions for our observations, we have to profile kernel calls with a tool such as *perf*.

4.5 Serverless MapReduce

The workload we chose for the case of only Mappers was to count the total number of words in a set of 200 news documents. While for the case 2, we count the number of words that begin with each of the 26 characters on the same load.

Figure 10 shows the results for the first case, wherein the mapper execution time shows a linearly increasing trend with

the number of file given to it for processing. Since the mappers are called concurrently from the coordinators, the time taken by it is more than mappers, probably having much overhead due to few failed mapper operations. After the mapper returns the count of each set of files assigned to it, the coordinator just sums and returns the result. Hence, the coordinator has constant overhead except for calling mappers, which is evident from the figure.

Figure 11 shows the case 2 in which we have reducers to process the results that are returned from the mappers in a multi-level arrangement. The number of reducers that are called for a given job is the function of the number of mappers assigned for the given job - the reducer can take results of only 2 mappers to process at a time. The trends in graph are similar to what we anticipated in which execution time of the coordinator is now sub-linear to the number of reducers it calls (in comparision to being constant in case 1).

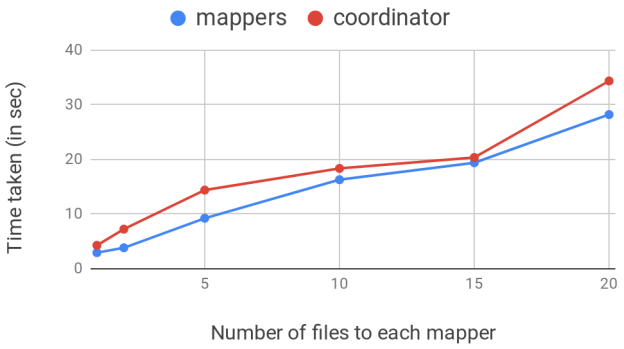

Figure 10: Time taken for a mapper to process input file vs. time taken by coordinator to complete entire MapReduce process

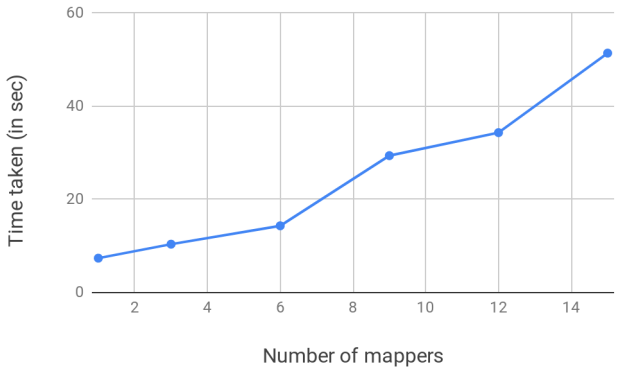

Figure 11: Time taken for a mapper to process input file vs. time taken by reducer to process intermediate file vs. time taken by coordinator to complete entire MapReduce process

5 Conclusion

With serverless architectures, developers do not need to worry about purchasing, provisioning, and managing backend servers, and at the same time offer greater scalability and quicker time to release, all at a reduced cost. However, serverless computing is not a silver bullet for all applications. There are many considerations to be mindful of, before adopting a serverless architecture. A function booting from a cold-state may degrade performance. Too many containers running on the same host will cause containers to compete for resources. For data data processing applications, having a single function do the heavy lifting breaks common assumptions about serverless architecture, such as statelessness and short execution time. Our results show that there is a noticeable overhead during the cold-start of a container, and when multiple containers are running on the same node. During a large spike in network requests, a significant number of requests fail, The time taken for the system to scale and reach a state where it can server all request is significant, and will definitely affect user experience. Building a end-to-end Serverless MapReduce model is possible, but it breaks few serverless concepts like statelessness and short lived functions. A simple workaround is to delegate the role of the master node in the MapReduce to a virtual machine.

For public offerings of Function as a Service (AWS Lambda, Google Cloud functions) to be able to perform as well do, we are positive that there are efficient algorithms that manage scaling and distribution of containers across multiple servers. Deploying a serverless architecture depends on the use case, workload and traffic generated by the application. It makes more sense, both from a cost perspective and from a system architecture perspective, to use dedicated servers for applications with a fairly constant, predictable workload.

References

[1] Docker containerization. [Online; accessed 11-April-2019].

[2] Kubernetes. [Online; accessed 11-April-2019].

[3] Openfaas. [Online; accessed 11-April-2019].

[4] Apache. Apache hadoop. [Online; accessed 11-April-2019].

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[6] Gianluca Borello. Container isolation gone wrong, 2017. [Online; accessed 11-April-2019].

[7] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[8] Youngbin Kim and Jimmy Lin. Serverless data analytics with flint. *arXiv preprint arXiv:1803.06354*, 2018.

[9] Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. Serverless computing provides on-demand high performance computing for biomedical research. *arXiv preprint arXiv:1807.11659*, 2018.

[10] Maxim Leonovich. Slow performance of docker containers, 2017. [Online; accessed 11-April-2019].

[11] Qubole Technical Articles. Qubole serverless big data analytics, 2017. [Online; accessed 11-April-2019].

[12] Sunil Mallya. Ad hoc big data processing made simple with serverless mapreduce, 2016. [Online; accessed 11-April-2019].

6 Appendix 1

- Our work involved lot of tweaking into the configuration files of Docker, Kubernetes and OpenFaas, specifically to customize the environment for supporting our functions and experimentation.
- All functions were written in Python preferably because of the ease of writting. The Client scripts for invoking requests to function endpoints were written in C++ 11 and used ¡chrono¿ for keeping time of requests. Other than the standard libraries, we used pysftp, json, requests and nltk.
- The overhead seen for I/O experiments were unexpected, but this behavior has been reported by others. Further investigation is needed to come up with conclusions.
- Some of the operations in Kubernetes were quite deviated from the normal stated on their website. For instance, Even after the pod has been spawned, the requests are continued to be rejected for a span of next 5-10 sec. This is a huge overhead while beginning from Clod start. However, the OpenFaas has the default policy of having atleast one replica ready, until otherwise stated in config.
- We faced difficulties accessing data in the user space and NFS storage through a container. There does not seem to ba an intuitive way of doing so, given the fact that the serverless functions do not write and maintain it's state. There exists tweaks to solve this around. we did good research for this problem but ended up going with the wrong choice (SFTP to the underlying NFS storage, which proved to be quite limiting with the increasing load).
- We did not have any issues working with the OpenFaas. There are well written blogs and tutorials by the creator. However, we do feel OpenFaas is limited in the flexibility of adding volumes, persistent storages, changing the base container images and container configurations. We had to tweak the Docker files while wrapping the functions in containers (one thing which was not openly stated on the website). The other limitation of OpenFaas is in its support for the triggers that can initiate execution. It natively exposes endpoints while supports other triggers (like, from Kafka, Cloud services like, AWS, Google Cloud and Azure) through 3rd-party codes.

7 Appendix 2

- All the containers that are used in the above experiments are built and hosted at https://hub.docker.com/u/g31pranjal
- The code for replicating the containers, client calls, Dockerfiles are avaialable publically at https://github.com/g31pranjal/serverless-tests