Utilization-focused Evaluation on Serverless Architectures

Heron Yang
Information Networking Institute
Carnegie Mellon University
heronyang@cmu.edu

Kung-Hsien Yu
Information Networking Institute
Carnegie Mellon University
kunghsiy@andrew.cmu.edu

ABSTRACT

Serverless computing is a pay-as-you-go code execution platform in which the cloud provider fully manages the execution environment to serve requests. As it potentially decrease the application operating cost to 77.08%[17], a heating trend of adopting serverless architectures had be found since 2016. A sufficient understanding of the system utilization of a serverless architecture will benefit both service providers in designing architectures and application developers in making adoption decisions.

In this paper, we first describe design goals of a serverless architecture then evaluate three serverless architectures: OpenLambda, IronFunctions, and Clofly. Both OpenLambda and IronFunctions apply Docker containers with different strategies of using state operations on containers for providing application isolations, while Clofly handles requests purely by using subprocess calls. We observed that the current isolation solution costs five times more on server resource usages comparing to the one with no isolation guarantee. Also, we discussed the importance of limited resource allocations for user functions, scalable-concerned cache policies, and efficiency improvements by sharing runtime resources.

Categories and Subject Descriptors

C.2.1 [**Computer-communication Networks**]: Network Architecture and Design—*serverless architecture*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Performance

Keywords

serverless architecture, utilization evaluation

1. INTRODUCTION

The need of cloud computings to support mobile, web, or IoT back-ends have began increasing since the 21 century started, but the developers pay a high operating cost in deploying and maintaining the cloud infrastructures under current common solutions. For example, an smart watch weather application developer who acquires a simple cloud back-end service that pulls the latest forecast data from the database every three hours will go through a complicated deploy process which includes renting a virtual server, installing server software, dealing with dependencies, and loading the applications. Moreover, it's also the developer's responsibilities to maintain the server, monitor the system states, and take care of sercurity problems. The minimum server rental tier would cost the developer roughly $5 per month[3] where he or she only utilizes less than %1 of its capacity.

Serverless computing became an emerging solution for above problem after container approaches like Docker containers had succeed. Under a serverless architecture, an application developer will deploy the applications at the level of functions, and a platform provider will manage the infrastructure of executing the user functions. While Amazon AWS Lambda, a serverless platform, first released in 2014, the serverless idea had been a popular topic; however, not many architecture approaches had be studied or evaluated. This became critical as the decision in choosing between the new serverless architectures and the traditional IaaS solutions will be based on the operating cost. And, only by measuring the utilization per hardware resources, we can estimate the cost and make trade-offs between different architectures.

In this paper, our contributions are:

- We evaluated system resource utilizations of three serverless architectures, OpenLambda, IronFunctions, and Clofly.
- We calculated the cost of application isolation by comparing the evaluation results between the architectures with and without container mechanisms.
- We discussed the design goals of a serverless architecture and how they had be approached by the three different architectures, OpenLambda, IronFunctions, and Clofly.

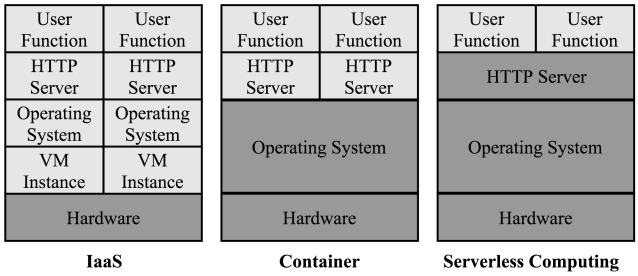

Figure 1: Evolution of cloud computing infrastructure: dark parts are managed by service providers, and light parts are rented to multiple application developers.

2. DESCRIPTION

2.1 Background

In this section, we will be looking into the evolution of cloud computing deployment infrastructures ending with the emerge of serverless computing. We found that the utilization evaluation on serverless servers is a missing part to understand the potentials of migrating applications to a serverless service.

2.1.1 Use Case - Micro-service Architecture for Mobile and Ubiquitous Computing

In a micro-service architecture, services are fine-grained and loosely coupled with a lightweight protocol for communication. According to Wikipedia[10], "the benefit of decomposing an application into different smaller services is that it improves modularity and makes the application easier to understand, develop and test. It also parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently". Micro-services plays a critical role in mobile and ubiquitous computings as they acquires light computing requests to be handled with a high degree of efficiency, availability, and cost efficiency. For example, the smart watch weather application mentioned in Section 1 would need a RPC call to pull the latest forecast from database every three hours. An Internet service architecture should be able to allocate this kind of requirement efficiently, and users will be charged only based on the usage.

Another example of the emerging need of an easy-to-use Internet service in a fine-grained fashion is Web back-end APIs. When one application developer wants to implement an authentication control of a certain source where the data of authenticated users are stored on a 3rd party store, the developer would need a solution with minimum operating cost and effort to implement a API for the customized authenticate procedure. One more example could be found in voice assistant services like Siri or Ok Google. This type of service is triggered occasionally indicating an efficient way of provisioning the back-end service to support it becomes beneficial since an always-waiting server will be a waste of resources.

2.1.2 Infrastructure as a Service (IaaS)

In an Infrastructure as a Service (IaaS) model, the service provider hosts hardware machines and other infrastructure components on behalf of the users. Users will be able to rent virtual machines on demand while the service providers offer system maintenance, backup and resiliency planning as services. Virtual machines can be scaled up or down dynamically and programmatically based on the policies written by the developer. Each virtual machines can be started from an empty system or a snapshotted system image.

In order to deploy a new application under an IaaS solution, the application developer will first rent a virtual machine from cloud providers like Amazon Web Services (AWS) or Google Cloud Engine. Then, software service needs to be deployed on the newly created virtual machine, for example, Nginx, NodeJS and the application code should be installed properly. Last, the application developers will need to manually setup HTTPS certificates, domain names, design scale up or down strategies, and other operating issues before the service launches. Even higher cost will be spent on maintaining the running service like the expense on protecting the server from cyber attacks. IaaS provides an isolated and dedicated system for median or large scale applications; however, it failed to provide a reasonable price for applications with tiny usage of the resources.

2.1.3 Container

Containers have became a better solution for packing the environment required by the application since Docker was released as an open source project by dotCloud in 2013[5]. According to Docker's website[11], "A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Containers isolate software from its surroundings, for example, differences between development and staging environments can help reduce conflicts between teams running different software on the same infrastructure."

Container technology enables the developers to install, manage, or scale Dockerized applications in a cluster, such as Kubernetes clusters, managed by providers like Amazon EC2 Container Service[2], Google Container Engine[6], etc. To deploy an application onto a cluster, the developer will first write a description file called *Dockerfile* to describe all the commands a user could call to assemble an image. Then, once the docker

image was built, it can be started in a container based on execution environment settings. Multiple containers can be started using a same image, and each container can be run, stopped, paused, un-paused, and killed.

2.1.4 Serverless Computing

Serverless computing, also known as Function as a Service (FaaS), is a pay-as-you-go code execution platform in which the cloud provider fully manages the execution environment to serve requests. The basic concept is first introduced by Zimki in 2006[9], but hasn't became successfully commercialized until AWS Lambda released in 2014. Serverless computing model benefits both the application developers and the service providers in many ways.

For application developers, applying serverless cloud services such as AWS Lambda allows the companies to reduce their infrastructure costs in up to 77.08%, according to Mario et al[17]. In an IaaS model, the lowest common price for renting a minimum dedicated virtual machine per month is around $5, but a serverless service only change around $0.20 per 1 million requests[4]. In addition, serverless computing lowers the background knowledge needed to be benefited by cloud computation resources. For example, more computations on a mobile devices could be migrated onto the cloud machines for a longer battery lifetime. Machine learning researchers without distributed system background knowledge could deploy their training computations onto the cloud machines as if running locally.

For service providers, a serverless computing provides a fine-grained usage on the hardware resources leading to more applications can be ran on a fixed amount of machines. However, based on our best knowledge, no evaluations had been published on discussing how much benefit could a cloud provider earn if they switched the services from IaaS model to serverless model. Maciej Malawski[14] also raised the need of a more detailed performance evaluation on emerging serverless infrastructures would help us understand the possibilities and limitations of this approach, which motivated our researches done in this paper.

Listing 1: User function deployment under serverless architecture Clofly: f.js contains the user function obtaining the request and response objects for handling a request. It can be deployed onto Clofly service using client side toolkit cf. A deployed URL will be returned and it is able to served end users immediately.

```
$ cat f.js
global.f = function (req, res) {
    res.send(new Date());
}

$ cf on f.js
Deployed at http://clofly.com/username/f (version: 1)

$ curl http://clofly.com/heron/f
"2017-04-21T18:04:57.826Z"
```

In this paper, we will use *serverless service* to refer a serverless architectures such as OpenLambda. *Application developers* indicates developers who write *user functions* to be run on cloud services. End users stand for the users of the deployed user functions. *Requests* we discussed in this paper are all HTTP or HTTPS requests, which could be using either a GET or POST method.

2.2 Design Goals

The major benefit of applying a serverless architecture is to decrease operation costs for both the application developers and service providers by utilizing the machines in a fine-grained fashion. That is, efficiency is the major design goal that different serverless services should try to maximize. Meanwhile, there are still some other design goals should be taken into concerns in order to provide a production service in a public market, which includes isolation, availability, scalability, flexibility, and responsibility.

2.2.1 Efficiency

Efficiency here is defined as the number of constrained applications can be served per unit of real or virtualized hardware resource. A service with higher efficiency indicates a higher degree of utilization of the resource, which can be measured in our later evaluations. From a provider's perspective, by having an efficient server, we could decrease the operating cost by paying less on the hardware resources but still serving the same demands. On the other hand, from an application developer's perspective, less cost will be spent on provisioning the applications as they will be able to rent the cloud machine resources in a finer grain. For example, the developers will only be charged based on the number of milliseconds their application had been ran, and there's no need for them to pay for any unused resource.

2.2.2 Isolation

Isolation protects the user functions from being interfered or eavesdropped by other running functions deployed by others, and provides a secure environment that guarantees the data privacy and integrity. A serverless server will be running untrusted user applications, and each user shouldn't be worried whether any information about their code will be learned by other users or even the serverless service provider. Therefore, a serverless server should disallow an user application from reading any data in disk or memory, binding to arbitrary ports, etc. The user applications should be

running within a limited CPU and memory resource.

2.2.3 Scalability and Availability

Scaling and making the application available to end users are the responsibilities of the server provider under serverless architectures since the application developers don't involve any provision decisions. Scalability means that the serverless servers should be able to scale up and down dynamically by recognizing the current request load. A common solution is to add a load balancer as the entry point of all the requests, and then forward requests to a scalable group of servers where each server should have the same ability of handling the requests. On the other hand, availability means that the serverless servers should be failure tolerated, and be accessible to the end users at any time.

2.2.4 Flexibility

Flexibility becomes relatively important as the code being deployed onto the servers is much smaller but with higher use frequency compared to existing architectures like IaaS or containers. A flexibility serverless server should be able to serve a newly deployed function instantly, and also be able to revoke or update a deployed function without too much overhead. Under above assumptions, a new development flow can be practically possible including online developments and runtime cloud resource allocation. Online developments mean that the developers can fully rely on developing codes on web or command line interfaces without running codes locally. Runtime cloud resource allocation means that the applications can deploy new functions onto serverless services and then execute them, which can be a useful mechanism for computation-heavy tasks like training a machine learning model.

2.3 Architecture

Architectures of three serverless platforms will be discussed in this section based on our best knowledge at the time this paper was written. In each serverless architecture, we will be mainly studying on the workflow of submitting a new application function by an application developer, the procedure of handling a request on a server, the cache policies, and the isolation solutions.

2.3.1 OpenLambda

OpenLambda[12] is an open-source project hosted by researchers from University of Wisconsin with a hope of providing a complete Lambda infrastructure that enables researchers to evaluate novel designs on a serverless platform. There are three main components in OpenLambda, an Nginx instance acting as a load balancer, a registry instance that holds the docker images which runs the user functions, and a worker instance that handles requests from the end users. A application developer will write their own Python application functions and then submit it to the registry. When a new request for this application function is observed by a worker instance, it will pull the docker image file loaded with the function, and then handles the incoming request by starting a Docker container.

At the time this paper was written, each instance is being prototyped as a docker container running on the same machine, and user functions are stored in a directory on disk locally; therefore, scalability is limited as the user functions are not being shared between machines. While handling an incoming request, one container will be started per user function, paused after the request is handled, and un-paused for handling the following requests on the same function. Under this workflow, high performance in terms of latency and throughput could be achieved, but it's less flexible for a application developer to dispatch a running function. Isolation is guaranteed at the same level as a regular docker container approach provides.

2.3.2 IronFunctions

IronFunctions[13] is an open source serverless platform integrated with a cloud-based message queueing (IronMQ) and a task processing (IronWorker) service developed by Iron.io[1]; however, our study is focused on the IronWorker in a single server mode using an embedded database and message queue. An application developer is excepted to develop and test the functions locally by executing them in a Docker container. For deployment, the application developer will upload the Docker image onto DockerHub, a cloud-based Docker image repository service, per application function. When a request of that application function was observed, the IronWorker will pull the image from DockerHub and then start it in a Docker container to handle the request. Docker images will be cached on the server, but no running or paused container will be holding resources after the request was handled.

As every request is handled by starting a new Docker container, it consumes a high overhead in CPU time leading to an inefficient processing time comparing to other architectures we found. In addition, DockerHub could potentially decrease the flexibility and increases the operating cost of the system since it's an extra service containing states that should be maintained by both the application developers and the IronFunctions service provider. However, since DockerHub is a native and easy-to-access platform for storing Docker images, IronFunctions could be served without storing any state, which brings high scalability and availability.

2.3.3 Clofly

Clofly is a serverless service developed by students at Carnegie Mellon University (CMU) as a side project.

The Clofly server implements Web Server Gateway Interface, WSGI, and is able to be deployed under an uWSGI architecture. The minimum uWSGI setup will be two running processes, one master and one worker. When a new request is found, the user functions will be loaded into an application template process and then be executed through Python subprocess call. The user application will be using globally installed library modules on the server instead of having its own built environment.

Unlike OpenLambda and IronFunctions, Clofly doesn't apply Docker containers and it's able to load the application function from a remote database, DynamoDB, into memory then execute the function without writing to or reading from the disk. Clofly is also designed for scale, there's no state saved on the server, application functions are saved in plain text on DynamoDB, and libraries needed by an application function will be preinstalled on the servers. However, since Clofly is still in its early prototyping stage, there's no isolation mechanisms or cache mechanisms in its design. The server will able to dynamically manage the globally installed libraries in the runtime using LRU or Adaptive Replacement Cache (ARC) algorithms as a revoke policy, but this feature hasn't be implemented yet.

3. EVALUATION

In this section, we launched evaluation tests on OpenLambda, IronFunctions and Clofly deployed on Google Cloud Platform. We describe the environment setup first, and then present the results.

3.1 Environment

Each of serverless service runs on one virtual machine on Google Cloud Platform. The virtual machine is equipped with 1 vCPU and 3.75GB memory, and installs Ubuntu 14.04 LTS. For each serverless server, we had uploaded 150 predefined and ready-to-serve functions where each of them will return time information to clients like "2017-04-22T00:41:51.458Z" The reason of choosing such simple function is to test how many functions can be supported by one machine without considering application processing time. Therefore, the performance difference between different architectures could be ralatively easy to observe. In each server, we only start one process to handle and response the requests for fairness concern.

To generate proper workloads, we created an additional virtual machine for each serverless service, and then simulated the client requests with Jmeter v3.2. These virtual machines are located within the same subnet, so network overhead could be minimized. In our test case, we assume that each client only requests one function from the server, so the maximum number of concurrent clients that can be tested is 150. We started

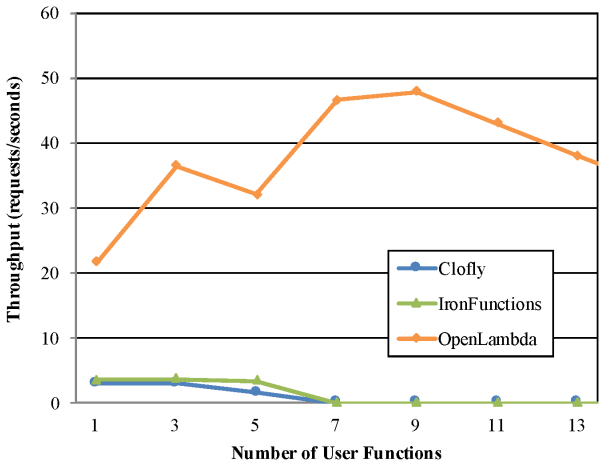

Figure 2: Throughput (requests/seconds)

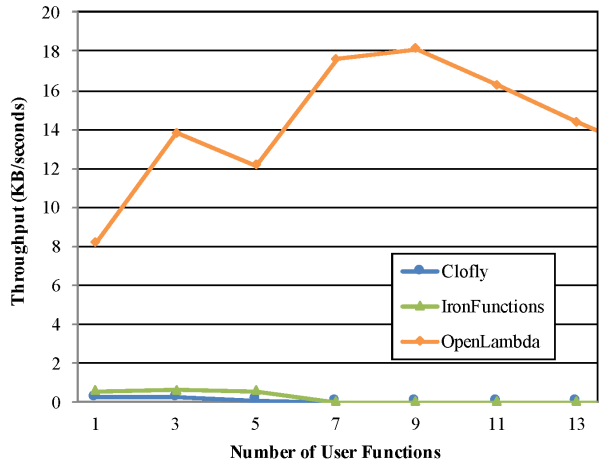

Figure 3: Throughput (KB/seconds)

our test by only one client, and then in each iteration, it will increase the number of our clients by 2 until the end. In Jmeter, we set the response timeout to 1500 ms, and connection timeout to 1500 ms. The LoopCount is 30, which indicates each client will request the server 30 times. To automate the test, we write another program called JmeterAuto.py. In each iteration, JmeterAuto will generate Jmeter configure file based on above descriptions, execute Jmeter program and collect data. In the same time, system resource statistics such as CPU usage and memory usage are logged on the service servers using dstat.

3.2 Client Observed Metrics

3.2.1 Throughput

Throughput is measured by either the amount of received requests or bytes per second before timeout. Be-

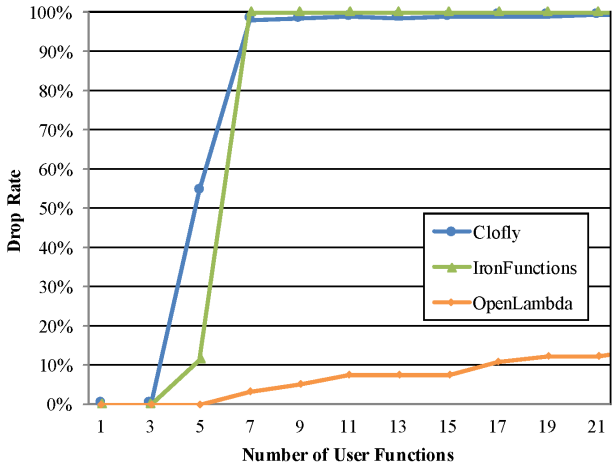

Figure 4: Drop Rate

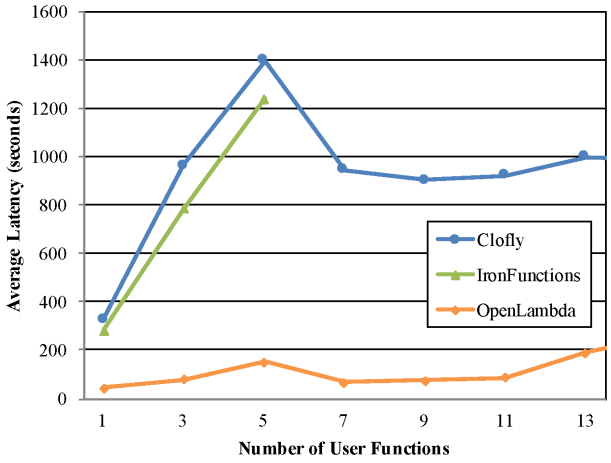

Figure 5: Average Latency

cause of different contents embedded in the HTTP header, the amount of bytes received per request are different between architectures where OpenLambda returns 378 bytes, IronFuction returns 166 bytes, and Clofly returns 71 bytes.

In Figure 3.2.1, OpenLambda is about to handle more than 20 requests per second when there's only one client requesting one of the functions, and its maximum capability is to handle near 50 requests per second. However, it hits the limit and started to decrease the throughput after 9 concurrent clients requesting different functions. In contrast, IronFunctions and Clofly are only able to handle around 3 requests per second at most, then the server stop to function when there are more than 7 concurrent clients.

OpenLambda outperforms IronFunctions and Clofly as a per-function dedicated handler is allocated in a Docker container and it is not being removed after a request was done. When a request is observed, the only overhead before working like a normal Dockerized server is to un-pause the container; on the other hand, IronFunction and Clofly pay a high overhead on downloading the user function or building the environment.

3.2.2 Drop Rate and Latency

As shown in Figure 3.2.2, IronFunctions and Clofly are able to function normally only under the number of concurrent clients is less than 3, and nearly 100% or the requests will fail when there are more than 7 concurrent clients. Besides the speed of handling a request, the nearly 100% drop rate could also indicate the servers didn't manage the request queue well. That is, instead of dropping some requests and then handling new requests such that they wouldn't be timed out, the servers stopped to serve any request successfully.

For latency shown in Figure 3.2.2, both IronFunctions and Clofly are able to handler requests within 400 ms, a real-time acceptable timeout period. However, Clofly increased to nearly 1 second and IronFunctions increased to 800 ms when there are 3 concurrent clients. Although the requests are handled eventually falls in our 1500 ms timeout limitation, the performance won't be acceptable in an real-time application where the users don't have patient to wait. IronFunction ran out of system resources after 5 concurrent clients, therefore, there's no information for us to calculate the latency as there's no successfully request's response. Even worse was that we had to force shutdown the machine running IronFunction after the test. Low latency performance found on OpenLambda benchmark leads to a high throughput it could achieve.

3.3 Server Measured Metrics

In order to analysis hardware resource usages while the server is still functioning, we picked one relatively light workload and one relatively heavy workload to measure the CPU and memory usages on these servers. Under a light workload, 3 concurrent clients is requesting the server on different user functions, where a heavy workload will have 7 concurrent clients. The servers were being monitored for one minute under these two workloads, and the requests were started to be sent between the third second to the fifth second.

3.3.1 CPU and Memory Usage under Light Load

OpenLambda consumes large amount of CPU resources when first time the user function is being requested because the server is starting up a handler that makes a Docker container from a Docker image, which is computational expensive. Afterward, it is able to handle requests less than 1% CPU usage after the containers had be started before. However, OpenLambda consume the largest amount of memory after it is started as the

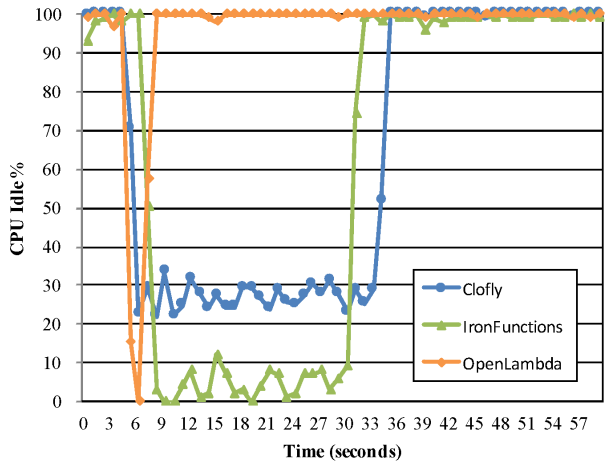

Figure 6: Server CPU Idle % under Light Load

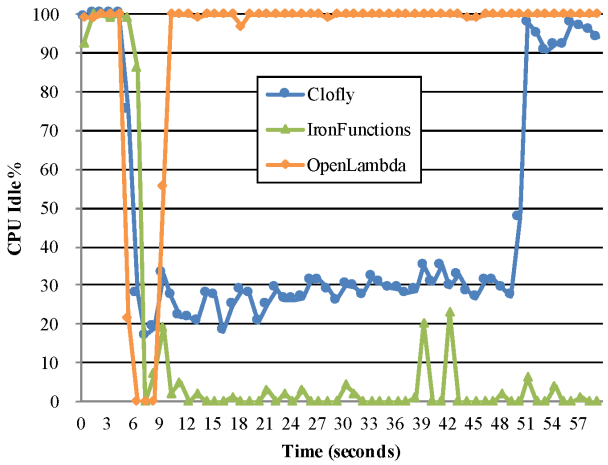

Figure 8: Server CPU Idle % under Light Load

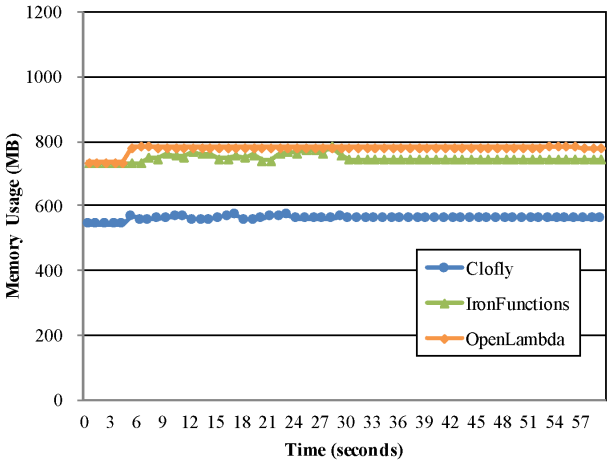

Figure 7: Service Memory Usage under Light Load

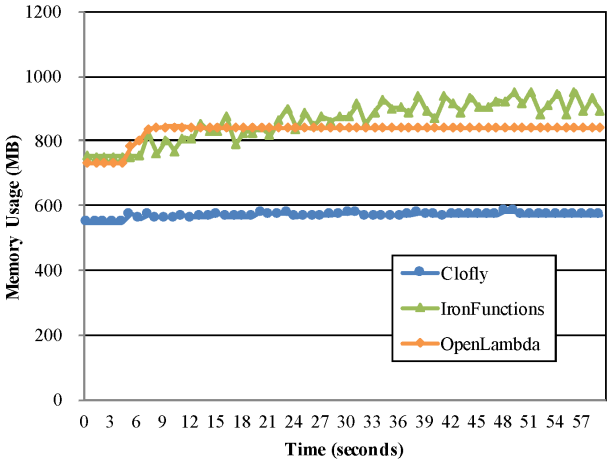

Figure 9: Service Memory Usage under Light load

containers are still handling memory resources to prepare future requests.

IronFunction consumes 90% to 100% of CPU time while handling the requests, this is a dangerous mechanism as it doesn't bound the resource allocated for running user functions, and there is no resource reserved for system programs. IronFunction consumes a high volume of memory because of the use of Docker containers. Clofly is able to bound the maximum CPU usage and minimize the usage of memory where it saves roughly 27% of memory comparing to OpenLambda by not using Docker containers.

3.3.2 CPU and Memory Usage under Heavy Load

Comparing to the light workload test, heavy workload requires the servers to hold CPU resource for a much longer time, where the Clofly server held 45 seconds, and IronFunctions held longer than one minute. OpenLambda was able to serve requests with 99% to 100% CPU idle after the containers had be started, Clofly was bounding the CPU usage at maximum 70% to 80%, but IronFunctions consumes nearly 100% CPU resources.

In term of memory usage, OpenLambda consumes large amount of memory with the same usage pattern under light workload, IronFunctions consumes increasingly large amount of memory because of running containers are accumulated, and Clofly was still serving with a lowest memory usage comparing with others.

3.4 Utilization Metrics

By combining the metrics collected from the clients and the servers, we could evaluate the utilization of the subject serverless architectures, which answers the main questions of this paper, "how well could these serverless

architectures perform per unit of hardware resource?"

Table 1 shows the maximum amount of memory usages by three serverless servers under both workloads, and marginal memory usage per user function can be calculated by dividing the change memory change between two workloads by 4, the change of the number of concurrent clients. Clofly consumes only 2.76 MB of memory usage per user function as it handles a request barely using a subprocess call on the application server script. In contrast, the other two architectures pays 14 to 55 MB memory cost for running the application code in an isolated environment.

Table 2 shows the CPU idle statistics under light workload, Clofly server took the longest duration holding the CPU resource, and this should be caused by the non-blocking mechanism between the service server and the application server communication. In the other word, after the service server initialized the application process to handle the request, it waits by keep sending a heartbeat to the application process every hundreds of milliseconds until it found the application process is alive. OpenLambda consumes a large amount of CPU resources within a relatively short duration, and IronFunctions consumes large amount of CPU resources for a long duration.

3.5 Discussion

By reading the evaluation results above, we could discuss following issues.

3.5.1 *Isolation is Expensive*

While isolation is a desired feature for application developers to protect their data and works, it costs at a high price. In Table 1, Clofly, a serverless architecture currently with no isolation mechanism, uses 2.76 MB memory per user function, while OpenLambda uses 14.56 MB memory per user function for isolation provided by containers. Five-fold memory is used for handling a single function under an isolated environment.

Docker containers provide much more than we need in serverless architectures, and decision made for the isolation of serverless applications should be taken more carefully. For example, an architecture should look into different approaches like native chroot, namespaces solutions before applying Docker containers.

3.5.2 *Resource Protection is Mandatory*

In our evaluation, IronFunctions server crashed and refused connections while we tried to logged onto the machine. To avoid this problem, user applications should always be run under bounded hardware resources, and never run an application process if the required resources can't be allocated. IronFunctions workers actually check free memory size before running a handler, but as the mechanism is integrated with their IronMQ service, it is out of the coverage of this paper.

3.5.3 *Unmeasured Metrics Affect the Results*

There are several unmeasured metrics that could make the service hard to use in practical even they perform well on our utilization evaluations. OpenLambda loads the user applications from local registry instead of loading it from a remote database, it is unscalable. In addition, OpenLambda un-puases the function containers after the requests, but this mechanism will eventually consume the memory as the deployed time pass. A further study on the managing the Docker container states is critical.

3.5.4 *Cache Policies Matter*

Cache increases performance but decreases the scalability, IronFunctions and Clofly applied very little cache mechanisms for scalability so they perform poorly comparing to OpenLambda from a client's view. Ideally, the server should load and cache as many processes or data it obtained as long as they are not stale and the hardware resource is sufficient.

3.5.5 *No Perfect Serverless Architecture*

None of the architectures we evaluate fulfill all the design goals described in Section 2.2 especially when some of the design goals are hard to achieve at the same time. For example, isolation is expensive and inefficient, and a server with cache mechanisms has performance improvement but is hard to scale.

4. RELATED WORK

AWS Lambda opens the page of serverless computings to public's eyes since 2014 as it allows developers to pipe different AWS services by writing a Lambda function. For example, a Lambda function can be triggered by an database operation callback, and it then runs data transformation code and stores results into data warehouse. As AWS Lambda locks the developers in the AWS eco-system by providing tools only convenient for integrating with other AWS services, Serverless Framework (serverless.com)[8] is a free and open-source for building applications exclusively on AWS Lambda. It started to attract people's attention in 2015 under the name JAWS by releasing the framework on Github[16], and the idea of pay-as-you-go application execution platform was introduced and became popular in developers.

Google Cloud Functions, a serverless environment to build and connect cloud services, was beta released in 2017 March. Comparing to AWS Lambda, there's no maximum execution time for the user functions, only JavaScript is supported, and it is designed for Google Cloud Platform event and HTTP triggers. Another player is Microsoft Azure Functions released in 2016 November. Like AWS Lambda, maximum 300 seconds

	Light Workload Max Memory Usage (MB)	Heavy Workload Max Memory Usage (MB)	Marginal Memory Usage per User Function (MB)
OpenLambda	784.8867188	843.1210938	14.55859375
IronFunctions	732.9140625	953.3671875	55.11328125
Clofly	572.4023438	583.4492188	2.76171875

Table 1: Maximum Memory Usage during responding users' requests. Under a light workload there are 3 concurrent clients requesting different functions and 7 concurrent clients for a heavy workload case.

	Duration (seconds)	Minimum CPU Idle (%)	Average CPU Idle (%)
OpenLambda	4	0	24
IronFunctions	26	0	12
Clofly	30	22	29

Table 2: CPU Idle Statistics under Light Workload: We observed the duration that different servers acquire for CPU idle percentage back to a normal state, the minimum CPU idle percentage during the test, and the average CPU idle percentage over the duration. Only light workload is analyzed here because IronFunctions and Clofly servers didn't release CPU resources during the whole monitored period in our heavy load test.

of execution time is applied, applications written in different languages could be supported including C#, JavaScript, etc.

In open source community, OpenWhisk[15] is an open serverless event-based programming service developed by IBM. In the OpenWhisk design, Nginx is used as an HTTP and reverse proxy server, and then the request will be forwarded to a controller, and eventually be handled by applications running Docker containers. IronFunctions released Alpha 2 version[7] in 2017 January, a new feature like running containers longer to increase the performance could highly improve the system resource utilizations. OpenLambda was first introduced to public on a USENIX workshop in 2016 where the researchers explored new research problems in the serverless computing space.

5. SUMMARY AND CONCLUSIONS

This paper evaluated server resource utilizations of three serverless architectures, OpenLambda, IronFunctions, and Clofly, by measuring the throughput, drop rate, latency, CPU and memory usages under different amounts of concurrent serving functions. Serverless architectures are designed for serving applications in a fine-grained fashion by sharing runtime servers; however, none of the above architectures could successfully deliver high system utilization when considering other fundamental design goals including runtime isolation, system scalability and flexibility.

In our evaluation, OpenLambda could serve requests with near 50 requests per second under less than 200 ms processing time, but it consumes high memory usage by holding unreleased Docker containers, and no revoke mechanism is found based on our best knowledge. IronFunctions provides high availability and scalability by storing Dockerized user applications on DockerHub, but the server stopped serving after more than 7 concurrent clients due to the high resource usage per request handling. Clofly, a serverless architecture prototype developed by CMU students could serve one application function with only 2 MB memory usage but it doesn't provide an isolation environment for the application runtimes.

We found the current isolation solution costs five times more server resource usages comparing to the one with no isolation guarantee. To provide an efficient and practical serverless architecture, the service providers should review and tune the application startup procedures precisely by sharing more runtime resources between applications while bounding the application resource usages efficiently.

6. REFERENCES

[1] About iron.io. https://www.iron.io/company/about/.

[2] Amazon ec2 container service docker management aws. https://aws.amazon.com/ecs/.

[3] Amazon ec2 pricing. https://aws.amazon.com/ec2/pricing/.

[4] Aws lambda — pricing. https://aws.amazon.com/lambda/pricing/.

[5] Docker persistent storage startup beamed up to the mother ship. http://searchitoperations.techtarget.com/news/450404231/Docker-persistent-storage-startup-beamed-up-to-the-mother-ship.

[6] Google container engine (gke) for docker

containers — google cloud platform. https://cloud.google.com/container-engine/.
[7] Ironfunctions alpha 2 — iron.io. https://www.iron.io/ironfunctions-alpha-2/.
[8] The serverless application framework powered by aws lambda and api gateway. http://serverless.com/.
[9] Serverless computing: An emerging trend of cloud. https://blog.intuz.com/serverless-computing-an-emerging-trend-of-cloud/.
[10] Microservices. https://en.wikipedia.org/wiki/Microservices, Apr 2017.
[11] What is a container. https://www.docker.com/what-container, Mar 2017.
[12] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.
[13] iron io. iron-io/functions. https://github.com/iron-io/functions, Apr 2017.
[14] M. Malawski. Towards serverless execution of scientific workflows–hyperflow case study. In *11th Workshop on Workflows in Support of Large-Scale Science (WORKS@ SC), volume CEUR-WS 1800 of CEUR Workshop Proceedings*, pages 25–33.
[15] Openwhisk. openwhisk/openwhisk. https://github.com/openwhisk/openwhisk, Apr 2017.
[16] Serverless. serverless/serverless. https://github.com/serverless/serverless, Apr 2017.
[17] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, et al. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pages 179–182. IEEE, 2016.