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. According to study[17], adopting serverless architectures decreases the application operating cost to 77.08%, which leads its popularity in application developer communities. 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 and then evaluate three serverless architectures: OpenLambda, IronFunctions, and Clofly. Both OpenLambda and IronFunctions use Docker containers with different state operations on containers for providing application isolation, while Clofly handles requests purely by using subprocess calls. We observe that the current isolation solution costs five times more on server resource usages comparing to the one with no isolation guarantee. Also, we discuss the importance of limiting resource allocations for user functions, scalability-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 computing to support mobile, web, or IoT back-ends have began increasing since the 21st century started, but the developers pay a high operating cost in deploying and maintaining the cloud infrastructures under current common solutions. For example, to design a weather application for smart watches, a developer acquires a cloud back-end service that pulls the latest forecast data from the database every three hours. Then, the developer has to go through a complicated deploy process which includes renting a virtual server, installing server software, dealing with dependencies, and uploading the applications. Moreover, the developer's responsibilities include maintaining the server, monitoring the system states, and taking care of security problems. In terms of expenses, the minimum server rental tier would cost the developer roughly $5 per month[3], but less than %1 of the capacity is used.

Serverless computing becomes an emerging solution for above problems regardless of the success of containers. Under a serverless architecture, an application developer deploys the applications at the level of functions, and a platform provider manages the infrastructure of the application executions. At the time Amazon AWS Lambda, a serverless platform, first released in 2014, the serverless idea had been a popular topic; however, not many architecture approaches have been studied or evaluated. This need motivates our measurements on the utilization per hardware resources as it provides insights for architects to make trade-offs between different architectures, and enables developers to estimate the operating cost and make decisions between the serverless architectures and the traditional IaaS.

The contributions of this paper are as follows:

- We evaluate system resource utilization of three serverless architectures: OpenLambda, IronFunctions, and Clofly.
- We calculate the cost of application isolation by comparing the evaluation results between the architectures with and without container mechanisms.
- We discuss the design goals of a serverless architecture and how they are approached by the three different architectures: OpenLambda, IronFunctions, and Clofly.

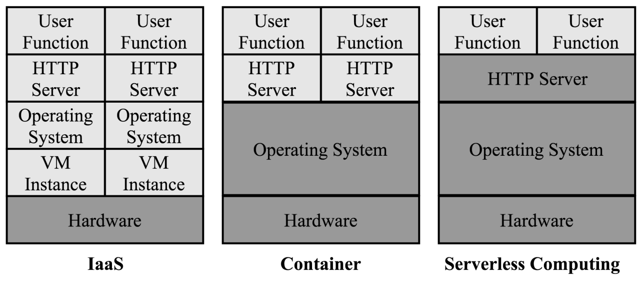

Figure 1: Evolution of cloud computing infrastructure: dark parts are managed by service providers, and light parts are rented to multiple application developers.

2. DESCRIPTION

2.1 Background

In this section, we review the evolution of cloud computing deployment infrastructures ending with the emerge of serverless computing. Figure 1 visualizes the evolution between these infrastructures. We find that the utilization evaluation on serverless servers is a missing part to understand the benefits 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 play a critical role in mobile and ubiquitous computing as they handle light requests 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 micro-service architecture should be able to satisfy this kind of requirement efficiently, and users should only be charged based on usage.

Another example of the emerging need of an easy-to-use fine-grained Internet service is Web back-end APIs. When one application developer wants to implement an authentication control where the data of authenticated users is stored on a 3rd party database, a solution with minimum operating cost and effort to achieve the customized authenticate procedure is expected. One more example is found in voice assistant services like Siri or OK Google. This type of service is triggered occasionally. Therefore, an always-waiting server is a waste of resources and an efficient way of provisioning the back-end service is needed.

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 users. Users rent virtual machines on demand, while the service providers offer system maintenance, backup and resiliency planning. 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 model, the application developer first rents a virtual machine from cloud providers like Amazon Web Services (AWS) or Google Cloud Engine. Then, software server like Nginx needs to be deployed on the newly created virtual machine, and the application code should be installed properly. Last, the application developers are required to manually setup HTTPS certificates, domain names, design scale up or down strategies, and handle other operating issues before the service launches. In addition, higher cost is 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 fails to provide a reasonable price for applications with tiny usage of resources.

2.1.3 Container

Containers have become 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 service providers. Examples are 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 needed to assemble an image. Once the docker image was built, it can be deployed into a container based of customizable execution environment settings.

Multiple containers can be started from the same image, and operations on a container are run, stop, pause, un-pause, and kill.

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 was introduced by Zimki in 2006[9], but have not become successfully commercialized until AWS Lambda released in 2014. Serverless computing model benefits both application developers and service providers in many ways.

For application developers, applying serverless cloud services such as AWS Lambda allows to reduce their infrastructure costs down 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 charges around $0.20 per 1 million requests[4]. In addition, serverless computing lowers the background knowledge needed to be benefited by using cloud computation resources. For example, computing tasks on a mobile device could be migrated onto the cloud machines for a longer battery lifetime. Machine learning researchers without distributed system background knowledge can deploy their training computations onto the cloud machines as if running locally. A work flow of deploying a simple serverless function on Clofly is demonstrated in Listing 1.

```
$ 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"
```

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

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

In this paper, we 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 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 achieve. Meanwhile, there are still some other design goals that should be taken into account in order to provide a production service in a public market. These include isolation, availability, scalability, flexibility, and responsibility.

2.2.1 Efficiency

Efficiency here is defined as the number of 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 service provider's perspective, by having an efficient server, he can decrease the operating cost by paying less on the hardware resources while serve same amount of demands. On the other hand, from an application developer's perspective, less cost will be spent on provisioning the applications when they are able to rent the cloud machine resources in a fined-grained fashion. For instance, the developers are charged based on the number of milliseconds their application use, 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 other developers, and provides a secure environment that guarantees the data privacy and integrity. In order to attract users then gain market shares, a secured environment of running the untrusted code uploaded by public developer communities is a fundamental property. Therefore, a serverless server should prevent 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, because the application developers don't involve any provision decision. Scalability means that the serverless servers should be able to scale up and down dynamically by predicting the future request load or by observing the current request load. A common solution is to add a load balancer as the entry point of all the requests, and it forwards requests to scalable servers where each server has 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 anytime.

2.2.4 *Flexibility*

Flexibility becomes relatively important as the code deployed onto the servers is much smaller but with higher update frequency compared to existing architectures like IaaS or containers. It means serverless server should be able to serve a newly deployed function instantly, and to revoke or update a deployed function without too much overhead. Under above assumptions, new development flows became practically possible including online developments and cloud resource allocation in runtime. Online developments mean that the developers can implement their applications on web or by command line interfaces without running the code locally. Cloud resource allocation in runtime means that the applications can deploy new functions onto serverless services and execute them on the fly, 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. In each serverless architecture, we focus on the work flow of deploying a new application function by an application developer, the procedure of handling a request, cache policies, and 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 run the user functions, and a worker instance that handles requests from the end users. An 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 application function, then handles the incoming request by starting a Docker container.

At the time when this paper was written, each instance is 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 because the user functions are not being shared between machines. For a cycle of handling an incoming request, one container will be started per user function when receiving the request, be paused after the request is handled, and be un-paused for handling the following requests on the same function. Under this mechanism, high performance in terms of latency and throughput is achieved, but it's less flexible for an 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 another open source serverless platform integrated with a cloud-based message queueing (IronMQ) and a task processing (IronWorker) service developed by Iron.io[1]. To simplify the comparison between different architectures, our study focuses on the IronWorker in a single server mode that uses an embedded database and message queue. An application developer is excepted to develop and to test the functions locally by executing them in a Docker container. For deployment, the application developer uploads the Docker image onto DockerHub, a cloud-based Docker image repository service. When a request of that application function is observed, a IronWorker pulls the image from DockerHub then starts it in a Docker container for handling the request. Docker images are cached on the server, but no running or paused container holds resources after the request is handled.

As every request is handled by starting a new Docker container, it consumes a high overhead in CPU time leading to a longer processing time compared to other architectures. In addition, DockerHub could potentially decrease the flexibility and increase the operating cost of the system, because it needs an extra service to contain states that should be maintained by both the application developers and IronFunctions service provider. However, since DockerHub is a native and easy-to-access platform for storing Docker images, IronFunctions is able to serve without storing any state on the server, 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 deployed under an uWSGI architecture. The minimum uWSGI setup requires 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 be executed through a Python subprocess call. The user application uses 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 loads application functions from a remote database, DynamoDB, into memory. Function executions are done in memory without reading downloaded data from disk. Clofly is also designed for scale as there's no state saved on the server. Instead, application functions are saved in plain text on DynamoDB, and libraries needed by an application function will be pre-installed on the servers. However, since Clofly is still in the early prototyping stage, there's no isolation mechanisms or cache mechanisms in its current design. The server is planned to dynamically manage globally installed libraries in the runtime with LRU or Adaptive Replacement Cache (ARC) algorithms as a revoke policy, but these features have not be implemented yet.

3. EVALUATION

In this section, we evaluate performances of OpenLambda, IronFunctions and Clofly 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 Ubuntu 14.04 LTS is installed. For each serverless server, we upload 150 predefined and ready-to-serve functions, and each function returns time information to clients like "2017-04-22T00:41:51.458Z". The reason why we choose such simple function is to know the maximum number of functions that can be supported in one machine. Therefore, the performance difference between architectures can be easily observed. For fairness concern, we control each server to run in a single process mode excluding the container processes.

To generate proper workloads, we create an additional virtual machine to simulate the client requests by *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 is 150. We start the test by only one client, then in each iteration, we increase the number of our clients by 2 until the end of the test. In settings for 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 for 30 times. To automate the test, we write another program called JmeterAuto.py. In each iteration, JmeterAuto.py generates Jmeter configure file based on above descriptions, executes Jmeter program and collects data. At the same time, system resource statistics such as CPU usage and memory usage are logged on the service servers by using *dstat*.

3.2 Client Observed Metrics

3.2.1 Throughput

Throughput is measured by either the amount of received requests shown in Figure 2, or bytes per second before timeout shown in Figure 3. Because 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 2, OpenLambda handles more than 20 requests per second, when there's only one client requesting one function, and its maximum capability is to handle near 50 requests per second. However, it hits the limit and starts 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, and the server stop functioning 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 removed after a request is finished. 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 from database or building the Docker container.

3.2.2 Drop Rate and Latency

As shown in Figure 4, IronFunctions and Clofly are able to function normally only when the number of concurrent clients is less than 3, and nearly 100% of the requests will fail when there are more than 7 concurrent clients. In addition to the speed of handling a request, the nearly 100% drop rate indicates the servers don't manage the request queue very well. That is, instead of dropping some requests and then handling new requests such that they wouldn't be timed out, the servers continue to serve timeout requests. Hence, it increases the drop rate.

For latency shown in Figure 5, both IronFunctions and Clofly are able to handle requests within 400 ms,

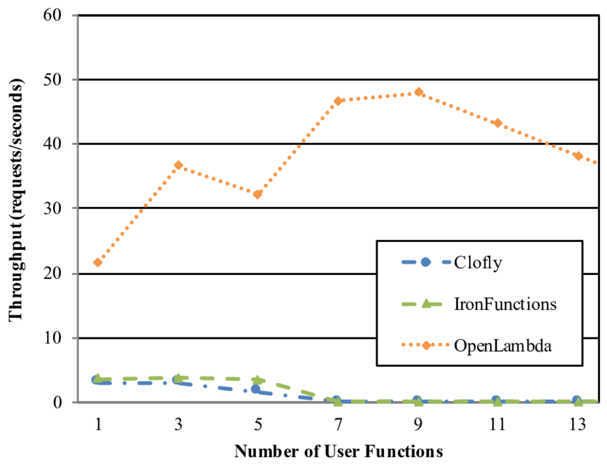

Figure 2: Throughput (requests/seconds)

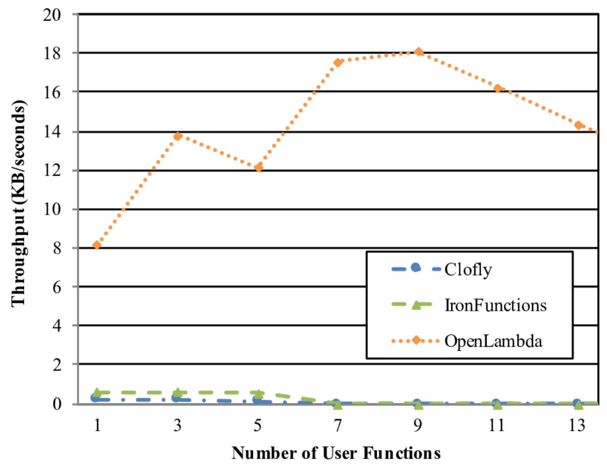

Figure 3: Throughput (KB/seconds)

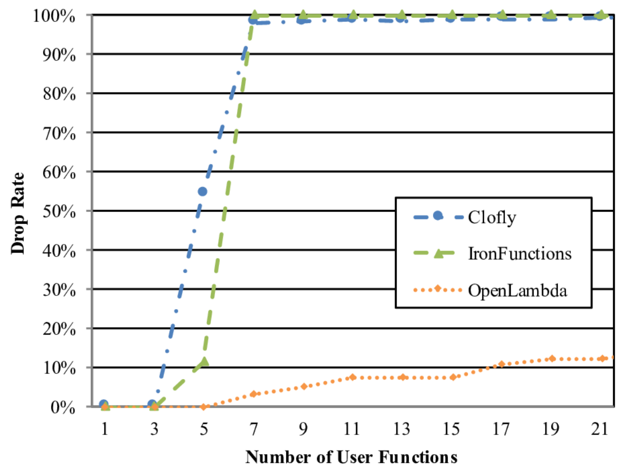

Figure 4: Drop Rate

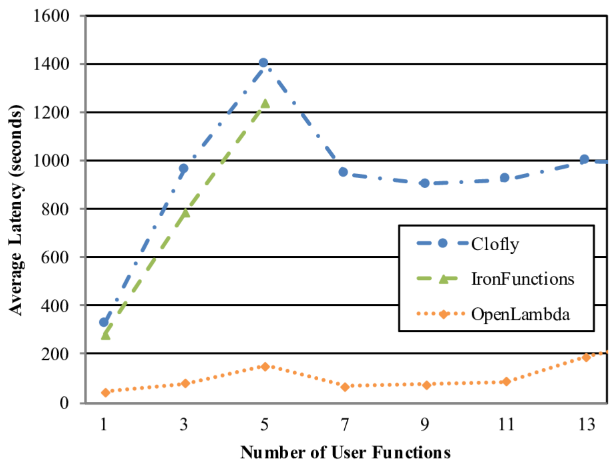

Figure 5: Average Latency

an acceptable timeout period for real-time applications. However, latency of Clofly increases to nearly 1 second, and of IronFunctions increases to 800 ms when there are 3 concurrent clients. Although the latency of requests are within our 1500 ms timeout limitation, the performance won't be acceptable in a real time application because the users don't have patience to wait. IronFunction runs out of system resources after 5 concurrent clients. Therefore, there's no information for us to calculate the latency as there's no successful response. Even worse, we are forced to shutdown the not-responding machine running IronFunction after the test. Low latency performance found on OpenLambda leads to a high throughput it achieves.

3.3 Server Measured Metrics

In order to analyze hardware resource usages, we choose one light workload and one heavy workload to measure the CPU and memory usages on these servers. Under a light workload, 3 concurrent clients are requesting the server with different user functions. On the other hand, a heavy workload is generated by 7 concurrent clients. The servers are monitored for one minute under these two workloads.

3.3.1 CPU and Memory Usage under Light Load

As shown in Figure 6, OpenLambda consumes a large amount of CPU resources when the user function is requested for the first time, because the server is starting a handler that produces a Docker container from a Docker image, which is computationally expensive. Afterward, it is able to handle requests less than 1% CPU usage as the containers was started already. However, in Figure 7, OpenLambda consumes a largest amount of memory after it is started because the containers still hold memory resources for future requests.

IronFunction consumes 90% to 100% of CPU time

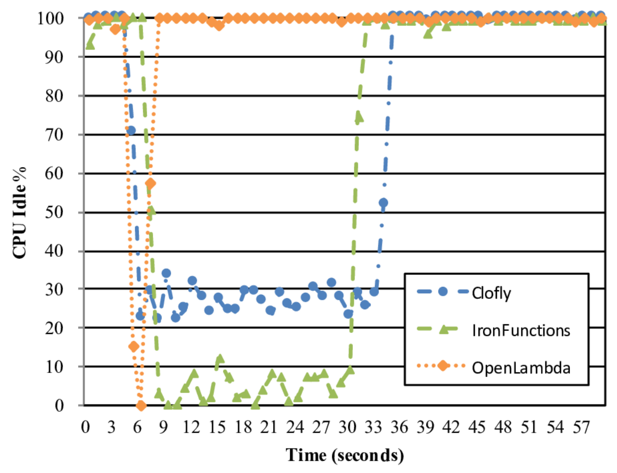

Figure 6: Server CPU Idle % under Light Load

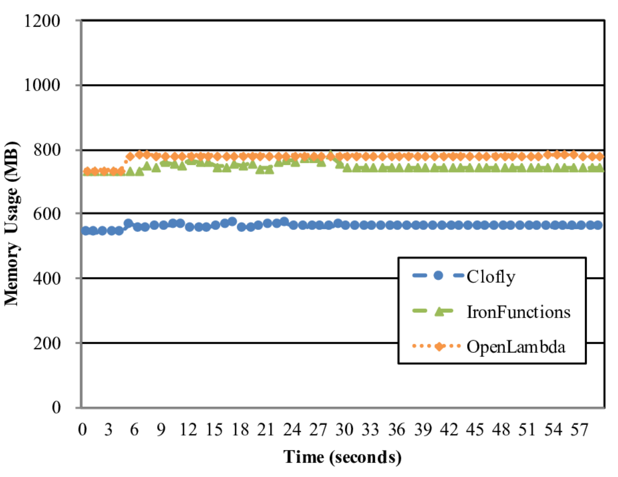

Figure 7: Server Memory Usage under Light Load

while handling the requests. This is a dangerous situation because 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 bounds the maximum CPU usage and minimizes 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*

Compared to the light workload test, heavy workload requires the servers to hold CPU resource for a much longer time as shown in Figure 8, where the Clofly server holds 45 seconds, and IronFunctions holds longer than one minute. OpenLambda is able to serve requests with 99% to 100% CPU idle time after the containers

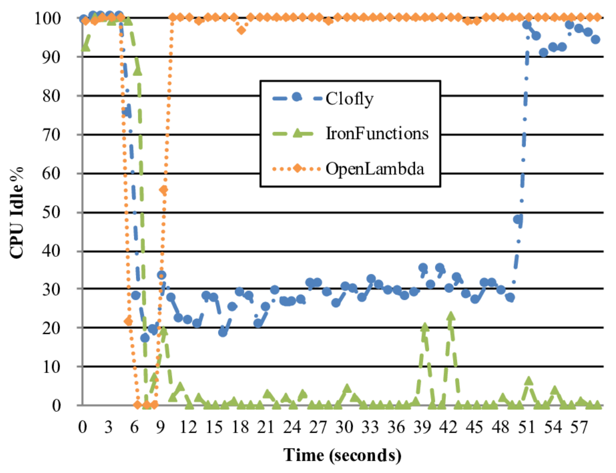

Figure 8: Server CPU Idle % under Heavy Load

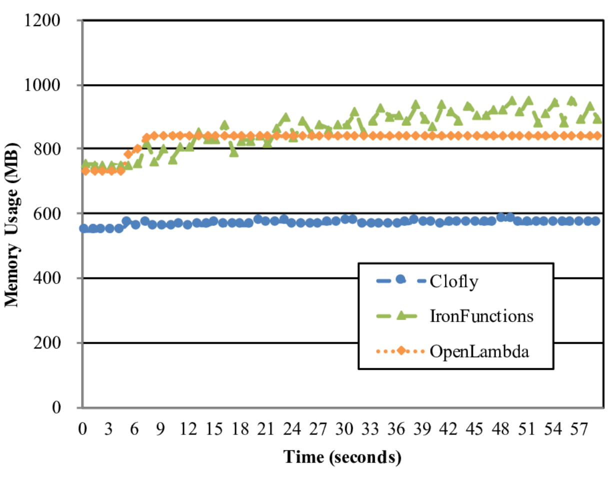

Figure 9: Server Memory Usage under Heavy load

is started, Clofly bounds the CPU usage at maximum 70% to 80%, but IronFunctions consumes nearly 100% CPU resource.

In term of memory usage presented in Figure 9, OpenLambda consumes a large amount of memory as the same usage pattern under light workload, IronFunctions consumes also large amount of memory because running containers are accumulated. By contrast, Clofly can serve with a lowest memory usage compared to other architectures.

3.4 Utilization Metrics

By combining the metrics collected from the clients and the servers, we could evaluate the utilization of serverless architectures, and answer the main question of this paper, "how well can these serverless architectures perform per unit of hardware resource?"

	Light Workload Max Memory Usage (MB)	Heavy Workload Max Memory Usage (MB)	Marginal Memory Usage per User Function (MB)
OpenLambda	784.886	843.121	14.558
IronFunctions	732.914	953.367	55.113
Clofly	572.402	583.449	2.7617

Table 1: Memory Usage: Maximum memory usage we observed while servers handling the requests (under a light workload there are 3 concurrent clients requesting different functions, and 7 concurrent clients in 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 record the duration of 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 don't release CPU resources during the whole monitored period in our heavy load test.

Table 1 shows the maximum amount of memory usages of three serverless servers under both light and heavy workloads, and marginal memory usage per user function is calculated by dividing the memory change between two workloads by 4, the difference of the number of concurrent clients. Clofly consumes only 2.76 MB of memory usage per user function because it handles a request by 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 takes the longest duration holding CPU resources, and this is caused by the non-blocking mechanism between the service server and the application server. In the other words, after the service server initializes the application process to handle the request, it keeps sending a heartbeat to the application process every hundreds of milliseconds until it finds 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 analyzing evaluation results above, following issues are found.

3.5.1 *Isolation is Expensive*

While isolation is an expected feature for application developers to protect their data and works, it costs at a high price. In Table 1, Clofly, a serverless architecture currently without 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 such as duplicating runtime resources for each request handler, and decision of isolation solutions in serverless applications should be taken more carefully. For example, an architecture should look into different approaches like native chroot and namespaces solutions before applying Docker containers.

3.5.2 *Resource Protection is Mandatory*

In our evaluation, IronFunctions server crashed and refused incoming 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 be executed if the required resources can't be allocated. After inspections of the IronFunctions source code, we found the IronWorker actually check if the free memory size of the system before running a handler, but it only works when the mechanism is integrated with its IronMQ service, which 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 practice even if they perform well on our utilization evaluations. OpenLambda loads user applications from a local registry instead of loading them from a remote database, which is not scalable as there's no way for the user function to be used in different servers. In addition, OpenLambda un-pauses the function containers after the requests, but this mechanism eventually consumes all of the memory as time passes. A further study on managing states of Docker containers is critical.

3.5.4 Cache Policies Matter

Cache increases performance but at the same time decreases the flexibility of updating or revoking a deployed function, IronFunctions and Clofly apply little cache mechanisms so they perform poorly comparing to OpenLambda from a client's view. Ideally, the server should cache as many processes or data it obtains 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 up.

4. RELATED WORK

AWS Lambda opens the page of serverless computing in the public eye since 2014. It allows developers to pipe different AWS services using a Lambda function. For example, a Lambda function can be triggered by a database operation callback, and it then runs data transformation code and stores results into data warehouse. However, 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 with its released framework on Github[16], and the idea of pay-as-you-go application execution platform was introduced and became popular in application developer communities.

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, but the downside is that only JavaScript is supported. 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 of execution time is applied, and applications written in different languages are supported including C#, JavaScript, etc.

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

5. SUMMARY AND CONCLUSIONS

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

In our evaluation, OpenLambda is able to serve about 50 requests per second with less than 200 ms processing time, but it consumes high memory usage by holding Docker containers in memory, and no revoke mechanism based is found on our best knowledge. IronFunctions provides high availability and scalability by storing Dockerized user applications on DockerHub, but the server stops serving after more than 7 concurrent clients due to the high resource usage per request. 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 when running applications.

We find 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, and share 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.