2018 IEEE 11th International Conference on Cloud Computing

Evaluation of Production Serverless Computing Environments

Hyungro Lee, Kumar Satyam and Geoffrey C. Fox
School of Informatics, Computing and Engineering
Indiana University Bloomington
{lee212, ksatyam, gcf}@indiana.edu

***Abstract*—Serverless computing provides a small runtime container to execute lines of codes without infrastructure management which is similar to Platform as a Service (PaaS) but a functional level. Amazon started the event-driven compute named Lambda functions in 2014 with a 25 concurrent limitation, but it now supports at least a thousand of concurrent invocation to process event messages generated by resources like databases, storage and system logs. Other providers, i.e., Google, Microsoft, and IBM offer a dynamic scaling manager to handle parallel requests of stateless functions in which additional containers are provisioning on new compute nodes for distribution. However, while functions are often developed for microservices and lightweight workload, they are associated with distributed data processing using the concurrent invocations. We claim that the current serverless computing environments can support dynamic applications in parallel when a partitioned task is executable on a small function instance. We present results of throughput, network bandwidth, a file I/O and compute performance regarding the concurrent invocations. We deployed a series of functions for distributed data processing to address the elasticity and then demonstrated the differences between serverless computing and virtual machines for cost efficiency and resource utilization.**

Keywords*—*FaaS, Serverless, Event-driven Computing, Amazon Lambda, Google Functions, Microsoft Azure Functions, IBM OpenWhisk

I. Introduction

Serverless computing is a commercial cloud service that enables event-driven computing for stateless functions executable on a container with small resource allocation. Containers are lightweight which means that it starts in a second and destroys quickly whereas a software environment for applications is preserved in a container image and distributed over multiple container instances. This elastic provisioning is one of the benefits that serverless computing takes along with its ease of use while traditional virtual machines on Infrastructure as a Service (IaaS) need some time to scale with system settings, i.e., an instance type, a base image, a network configuration and a storage option.

Most services in the cloud computing era, pay-as-you-go is a primary billing method in which charges are made for allocated resources rather than actual usage. Serverless computing may provide a cost-efficient service because it is billed for the execution time of containers without paying for procured resources that never used. Serverless also uses 0.1 second as a charging metric although many VM servers still use an hourly charge metric. Amazon recently applied per-second billing to EC2 services as Google Compute and Microsoft Azure already have the per-second billing, but it even costs every second whether a program runs or not.

Serverless is a miss-leading terminology because it runs on a physical server but it succeeded in emphasizing no infrastructure configuration along with the preparation of computing environments. Fox et al [1] defines serverless computing among other existing solutions, such as Function-as-a-Service (FaaS) and Event-Driven Computing, and we see production serverless computing environments offer an event-driven computing for microservices in which event is an occurrence generated by other systems and resources and microservices are described as a formal syntax written in a programming function. New record on a database, deletion of object storage, or a notification from the Internet of Things devices is an example of various events and the event typically contains messages to be processed by single or multiple event handlers. Sometimes an event is generated at a particular interval of time which is predictable, but in many cases, significant numbers of event messages need to be processed at scale instantly. Horizontal scaling for processing concurrent requests is one of the properties of cloud-native applications [2] which have practical approaches and designs to build elastic and scalable systems. Data processing serverless software (ExCamera [3], PyWren [4]) for video rendering and Python program recently show that large loads on the event handlers can be ingested on serverless computing by using concurrent invocations. We also understand that namespaces and control groups (cgroups) offered by containers power up serverless computing with resource isolation to process dynamic applications individually, but provisioning a thousand of instances within a few seconds.

A new event message is processed on a function instance isolated by others, and multiple instances are necessary when several event messages are generated at the same time. Event messages generated by mobile applications, for example, are lightweight to process but the quantity of incoming traffic is typically unpredictable so that such applications need to be deployed on a particular platform built with dynamic provisioning and efficient resource management in which serverless computing aims for [5]. We may observe performance degradation if a single instance has to deal with multiple event messages with a heavy workload in parallel. Unlike IaaS, the cost of instantiating a new instance is relatively small, and an instance for function execution is short-lived on serverless computing thus it would have demanded to process concurrent function invocations using multiple instances like one instance per request. Some applications that can be partitioned into several small tasks, such as embarrassingly parallel, may

2159-6190/18/$31.00 ©2018 IEEE
DOI 10.1109/CLOUD.2018.00062

take advantage of the concurrent invocations on serverless computing in which horizontal scaling is applied to achieve the minimal function execution time required to process the distributed tasks.

In this paper, we evaluate serverless computing environments invoking functions in parallel to demonstrate the performance and throughput of serverless computing for distributed data processing. We compare the performance of CPU, memory, and disk intensive functions running in between a sequential and a concurrent invocation which helps understanding performance bottlenecks and function behaviors on serverless computing environments. We also measure the throughput of a set of event handlers including HTTP, database, and storage which may indicate a maximum size of dequeuing event messages because functions are triggered by these common handlers supported by each serverless provider. Continuous development and integration are tested with source code changes and function configuration changes, e.g., timeout value and memory size while concurrent functions are running. The rest of the paper contains comparisons between IaaS and FaaS using experiments on big data and deep learning applications and the latest features offered by each serverless computing environment from Amazon Lambda, Microsoft Azure Functions, Google Functions and IBM OpenWhisk.

A. Trigger

A trigger, also called a front-end event handler invokes a function with event messages as input parameters thus a function is executed to process a request. Timers invoked by crons are used to accomplish a set of tasks on a regular interval. For example, an Apache HTTP web server error log is removed 3 AM daily followed by archiving a copy of the log 2 AM by scheduling routine cron tasks. Sensors at monitoring services detect events as a series of logic, and new event handlers are continuously added based on application behaviors and purposes. For example, Internet of Things (IoT) device at a smart home detects receiving a package from online retailers and generates a new event message which might be a trigger of other applications e.g. sending text messages to a package recipient. Serverless computing providers understand various use cases and support different types of events including HTTP requests, object storage like AWS S3, and a database like IBM Cloudant thus as many actions as they can handle to answer back the event messages. Event handlers also called triggers either listen to events and create a function invocation (push model) or collect changes at a regular interval to invoke a function (pull model). In this paper, We measure trigger resolutions to see how sensitive it is and understand its capacity of concurrent event messages. To measure the latency of triggers, we ran a simple function on AWS Lambda with three triggers; HTTP API gateway, DynamoDB, and S3. For IBM OpenWhisk, we ran a function using an HTTP trigger and the IBM Cloudant trigger. For Google Cloud Function, we had triggers from HTTP, Google Cloud Storage and a pub/sub messaging trigger. For Azure Functions, we had triggers from HTTP and storage.

1) HTTP Trigger: HTTP trigger provides a simple but a rich format to invoke a function with various content types such as archive files, text, and JSON and multiple methods such as PUT, POST and DELETE to deliver event messages differently. Asynchronous non-blocking call is available to deal with concurrent requests, but queuing systems and database systems are more suitable to estimate dynamic requests than the HTTP trigger.

2) Database Trigger: Database trigger invokes a function when there is an insertion/modification/deletion of a record in a table which behaves like a message queuing system. Google supports pub/sub trigger in the serverless platform, and it would be exchangeable with a database trigger since Google Functions does not have a database trigger. We see the comparison of the database type of trigger with AWS DynamoDB and IBM Cloudant as a direct trigger to their respective vendors' functions. As of now, we cannot compare Azure and Google Cloud as they do not have a direct trigger available to their respective functions. As per the graph, we see that performance of the AWS DynamoDB trigger surpasses the IBM Cloudant trigger.

3) Object Storage Trigger: Object storage is widely used to store and access data from various platforms, and we find that the object storage trigger is supported in the most serverless providers. AWS S3 trigger performs better than the Google Cloud storage trigger, but we still find that HTTP trigger is a more reliable choice for processing multiple requests. Note that we were not able to perform the object storage trigger for IBM cloud storage it does not offer a direct trigger to IBM Openwhisk as of now.

Serverless computing environments with concurrent invocations may support distributed data processing with its throughput, latency and compute performance at scale [6]. There are certain restrictions that we must be aware of before implementing a serverless function, for example, event handler types are a few; HTTP, object storage, and database in common, memory allocation is small; 512MB to 3GB memory allowance per a container, function execution time is allowed only in between 5 minutes and 10 minutes and a 500MB size of a temporary directory is given. In the following sections, we show our results towards Amazon Lambda, Google Functions, Microsoft Functions, and IBM OpenWhisk Functions with its elasticity, concurrency, cost efficiency and, functionality to depict current serverless environments in production. Big Data Benchmark from AmpLab and TensorFlow ImageNet examples are included as a comparison of costs and computation time between serverless computing and virtual machines as well.

II. Evaluation

We evaluate serverless computing environments on the throughput of concurrent invocation, CPUs, the response time for dynamic workload, runtime overhead, and I/O performance. We also compare cost-effectiveness, event trigger throughput, and features using a set of functions written by supported runtimes such as nodeJS, Python, Java, and C#. Each provider has different features, platforms, and limitations, so we tried to address the differences and find similarities among them. Some of the values may not be available because of two reasons, an early stage of the serverless environments and limited configuration settings. For example, Microsoft Azure runs Python 2.7 on Windows NT as an experimental runtime language thus some libraries and packages for data analysis are not imported, e.g., TensorFlow library with Python 3.5 or higher, and Google Functions is in a beta version which only supports NodeJS, a javascript runtime although Python is internally included in a function instance. 512MB memory limit on IBM OpenWhisk prevents running TensorFlow ImageNet example which requires at least a 600MB size of memory to perform image recognition using trained models. New recent changes are also applied in our tests such as 3008MB memory limits on Amazon Lambda, and Java runtime on Microsoft Azure Functions. All of the evaluations were performed using 1.5GB memory allocation except IBM with 512MB and 5 minutes execution timeout. We use the Boto3 library on Amazon Lambda to specify the size of a concurrent function invocation, and HTTP API for Microsoft Azure and IBM OpenWhisk. We use object storage to invoke Google Functions as well. We completed benchmarks using the set of functions written by NodeJS 6.10, Java, C#, and Python 3 and 2.7.

A. Concurrent Function Throughput

Function throughput is an indicator of concurrent processing because it tells how many function instances are supplied to deal with extensive requests. Asynchronous, non-blocking invocations are supported by various methods over the providers. Amazon SDK (Boto3) allows to invoke a function up to an account's concurrent limit, and S3 object storage or DynamoDB database is an alternative resource to invoke a function in parallel. Google Functions only allows for concurrent execution by a storage bucket and a rate of processing event messages varies on the length of the message and its size. Microsoft Azure Functions also scales out automatically by its heuristic scale controller. IBM OpenWhisk does not seem to provide scalability unless functions are manually invoked as a workaround. We had to create a thousand of functions with an identical logic but a different name and treat them like invoking a single function in parallel. Figure 1 is a throughput result per second over the four serverless providers from 500 to 10000 concurrent invocations. Amazon Lambda generates about 400 throughputs per second in average, and AWS quickly reaches its maximum throughput from a small number of concurrent invocation (1000). IBM OpenWhisk and Microsoft Azure Functions show similar behavior in reaching the best throughput at 2000 invocations and decreasing slowly over increased invocations. Google Functions shows a slow rising but steady progress of throughput over increased invocations. Throughput at ten thousands of invocations on Google Functions is about 168 per second which is better than IBM and Azure.

B. Concurrency for CPU Intensive Workload

Multiplying two-dimensional array requires mostly compute operations without consuming other resources thus we created the matrix multiplication function written in a JavaScript to stress CPU resources on a function instance with concurrent invocations. Figure 2 shows an execution time of the function between 1 and 100 concurrent invocations. The results with 1 concurrent invocation which is non-parallel invocation are consistent whereas the results with 100 invocations show the overhead of between 28% and 4606% over the total execution time. The results imply that more than one invocation was assigned to a single instance which may have to share allocated compute resources, i.e., two CPU-intensive function invocations may take twice longer by sharing CPU time in half. For instance, the median of the function execution time on Amazon Lambda (3.72 seconds) is about twice the non-concurrent invocation (1.76 seconds).

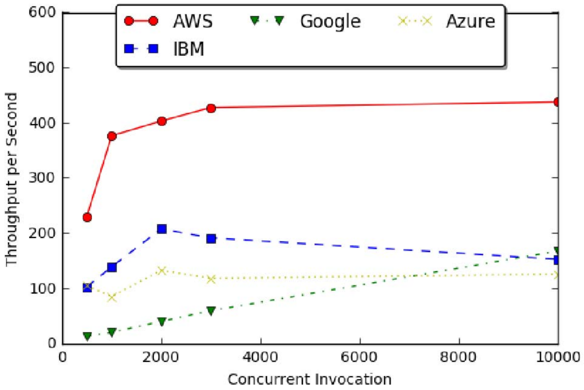

Fig. 1: Function Throughput on Concurrent Invocations

TABLE I: CPU Performance

Provider	GFLOPS per function	TFLOPS in total of 3000
AWS	19.63	66.30
Azure	2.15	7.94
Google	4.35	13.04
IBM	3.19	12.30

19.63 gigaflops are detected on AWS Lambda with the 1.5GB size of memory configuration (whereas 40 gigaflops are measured with 3GB memory), but it can reach more than a teraflop when a fleet of containers are provisioned for concurrent invocations. Serverless platform allocates computing resources based on the number of requests which shows to a peak double-precision floating point performance of 66.3 TFLOPs when an Amazon Lambda function is invoked concurrently. Table I is the result of invoking three thousands of functions on serverless functions which indicates proportional between the number of functions and the aggregated flops. 66.3 teraflops are relatively good performance. For example, Intel six-core i7-8700K generates 32 gigaflops, and the latest NVIDIA TESLA V100 GPU delivers 7.8 teraflops for a double precision floating point. In the comparison of the total of TFLOPS, AWS Lambda generates 5-7 times faster computing speed than others. Azure Functions, IBM OpenWhisk, and Google Functions are in either a beta service or an early stage of development; therefore, we expect that these results will need to be revisited in the future.

C. Concurrency for Disk Intensive Workload

A function in serverless computing has a writable temporary directory with a small size like 500MB, but it can be used for various purposes, such as storing extra libraries, tools, and intermediate data files while a function is running. We created the function which writes and reads files in the temp directory to stress a file I/O. 100MB size of a file is written by a random size of blocks with fsync to ensure all

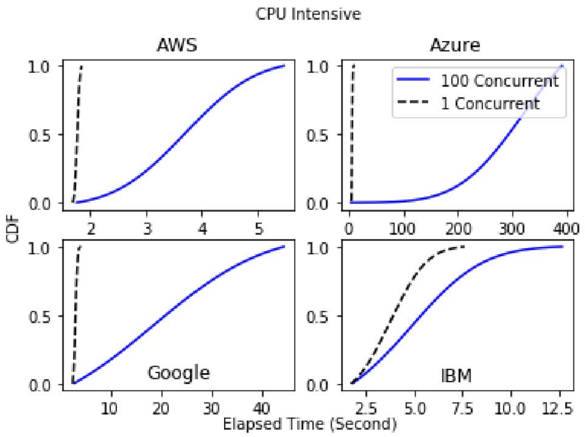

Fig. 2: Concurrency Overhead with CPU Intensive Function

TABLE II: Median Write/Read Speed (MB/s)

Provider	100 Concurrent		1 Concurrent	
	Write	Read	Write	Read
AWS	39.49	92.95	82.98	152.98
Azure	-	-	44.14	423.92
Google	3.57	54.14	9.44	55.88
IBM	0.50	33.89	7.86	68.23

buffered file objects are transferred to the disk. Reading the file is done by random offset blocks with 512 bytes read. We do not have information of the actual device hardware type of the temporary directory we tested. Google claims that the temporary directory exists on memory which consumes allocated memory size of a function but we do not find information from other providers whether they mount it with a persistent storage device like HDD or SSD. The measured I/O performance toward a temporary directory is shown in Figure 3 with concurrent invocations. The results with 100 invocations show that Amazon generates an execution time overhead of 91%, Google generates the overhead of 145% and IBM generates the overhead of 338% whereas Microsoft Functions fail to complete function invocations within the execution time limit, 5 minutes. The median speed of the file read and write is available in the Table II. Reading speed on Azure Functions is the most competitive among others although it is not measured on 100 concurrent invocations due to the storage space issue. Writing a file between 1 and 100 concurrent invocations is slightly worse compared to reading, the overhead of 110% on Amazon Lambda, 164% on Google Functions and 1472% on IBM OpenWhisk exist whereas the writing speed on Amazon Lambda is 11 to 78 times faster than Google and IBM when 100 concurrent invocations are made.

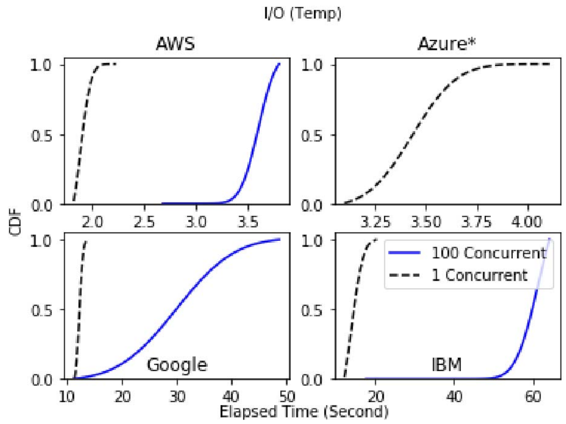

Fig. 3: Concurrency Overhead with File I/O Intensive Function

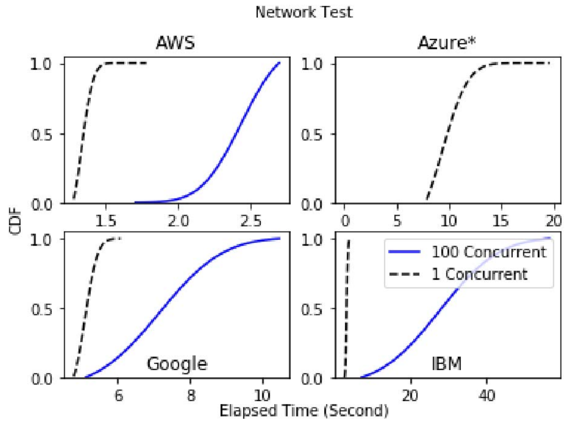

Fig. 4: Concurrency Overhead with Downloading Objects Function

D. Concurrency for Network Intensive Workload

Processing dataset from dynamic applications such as big data and machine learning often incur significant performance degradation in congested networks due to large transactions of file uploading and downloading. If such activities are distributed at multiple locations, network delays can be easily mitigated. Containers for serverless functions tend to be deployed at different nodes to ensure resource isolation and efficient resource utilization and this model may help resolve this issue especially when functions are invoked in parallel. We created a function which requests data size of 100 megabytes from object storage on each service provider thus a hundred concurrent invocations create network traffic in a total of 10 gigabytes. Figure 4 shows delays in the function execution time between 1 and 100 concurrent invocations. We find that Google Functions has a minimal overhead between 1 and the 100 concurrency level whereas Amazon Lambda is four times faster in loading data from Amazon object storage (S3) than Google object storage. Microsoft Azure Functions fails to get access of data from its blob storage at 100 concurrency level, and we suspect it is caused by the experimental runtime, i.e. Python 2.7 or a single instance for multiple invocations. Default runtime such as C# and F# may support scalability better than the other runtime under development on Microsoft Azure Functions.

E. Elasticity

A dynamic application performing latency-sensitive workloads needs elastic provisioning of function instances otherwise overhead and failure would be observed during the processing of workloads. We assume that serverless computing scales out dynamically to provide additional compute resources when the number of concurrent invocations is increased rapidly. We created the simple function that takes less than 100 milliseconds, and the function was invoked with different concurrent sizes ranging from 10 to 90 over time resulting in about 10 thousands of the total invocations within a minute. With this setup, we expected to observe two values; delays of instantiating new instances (which also called cold start) and a total number of instances created during this test. We believe that it explains whether elasticity mechanisms on serverless computing is efficient or not with regarding provisioning and resource utilization. Delays in processing time would be observed when existing function instances are overloaded and new instances are slowly added which may cause performance degradation in the entire invoked functions. Figure 5 shows different results among the serverless providers with the same dynamic workloads over time. The line with a gray color indicates the number of concurrent invocations per 200ms time window which completes ten thousand function executions within a minute and the changes of $\pm 3\ to \pm 30$ concurrent sizes was made in each invocation to measure horizontal scaling in/out. We observed that new function instances were added when a workload jumps to higher than the point that existing instances can handle and the increased number of function instances stay for a while to process future requests. We find that Amazon and Google support elasticity well in which the 99th percentile of the function execution time is below 100 and 200 milliseconds whereas both IBM and Azure show significant overhead at least two times bigger than others if we compare the 99th percentile of the execution time. The number of instances created for this workload was 54, 10, 101 and 100 in the order of Amazon, Azure, Google, and IBM. If there is a new request coming in while a function is in processing current input data, Amazon provides an additional instance for the new request whereas others decide to increase the number of instances based on other factors, such as CPU load, a queue length, an age of a queued message, which may take some time to determine. The function we tested uses the NodeJS runtime and scalable trigger types, but we would consider other runtimes such as C# and Java and other triggers like databases to see if it performs better in dealing with the dynamic workload. Each serverless provider uses different operating systems and platforms and it seems certain runtimes and triggers have better support in handling elasticity than others. For example, Azure Queues has the 32 maximum batch size to process in parallel and Azure Event Hubs doubles the limit. Table III contains actual numbers we measured during this test and the function execution time which is represented by blue dots in the figure would expect to take less than 0.1 second in a single invocation, but there are overhead when the workload is increased in which the standard deviations and 99th percentile indicate in the table. It explains that the increased number of instances should be available instantly with additional amounts of computing resources to provide enough capacity for the future demands.

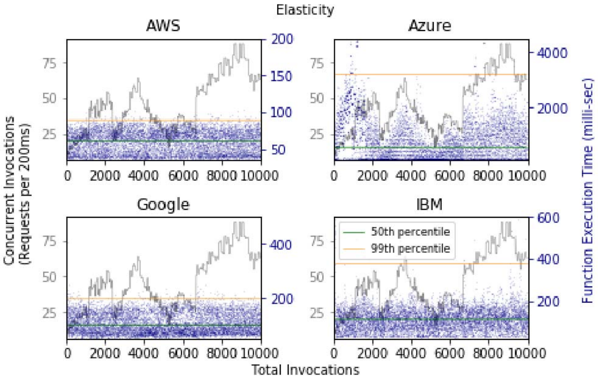

Fig. 5: Response Time for Dynamic Workload

TABLE III: Execution Time (milli-sec) on Elasticity

Provider	MED	SD	1st PCTL	90th PCTL	99th PCTL
AWS	61.08	14.94	35.08	78.99	89.41
Azure	574.0	747.33	118.0	1808.30	3202.0
Google	101.0	38.75	57.0	162.0	198.0
IBM	112.0	142.23	31.0	177.0	378.79

MED = Median, SD = Standard Deviation, PCTL = Percentile

F. Continuous Deployment and Integration

*Dev*elopment and *Op*eration*s* (DevOps) paradigm is applicable to serverless functions in enabling continuous delivery and integration while functions are in action. Functions ought to be often changed for bug fixes and updates, and a new deployment of functions should not affect existing workloads. In this section, we measure failures and delays of function executions when code blocks are updated and function configurations are changed where the transition to the next version of a function is explained in the context of concurrent invocations. We started with 10 concurrent invocations with a simple NodeJS function which takes a few seconds to complete and made a total of 500 function executions within 10 seconds. Two events were made during this experiment. First, a change of source code was made before the first 200 invocations completed and the second event with new function configurations such as updates on timeout value and the size of memory allocation was made in the next 200 invocations. We also prepared the function with a warmed-up instance by executing the function multiple times before this experiment to ensure that function instances are initialized and ready to process a certain load. Figure 6 shows different behaviors in dealing with those events. A gray dot indicates an existing function instance and green plus marker indicates a new function instance whereas a red 'x' marker indicates a failure of a function invocation which we avoid in any level of processing production workloads. It seems that Amazon re-deploys a function instance whenever there is a change in a code or a configuration but it keeps an existing deployment in a certain period to handle incoming requests during the transition. If the function is invoked right after those events, it is likely that the previous version of the function will be processing the new invocation. We also do not find a same number of instances are prepared. For example, we saw that 10 instances were waiting for new invocations, but new deployment started with a single instance along with purging previous instances. If a new function instance requires initializing processes such as downloading necessary files to a temp directory, it creates delays in processing new

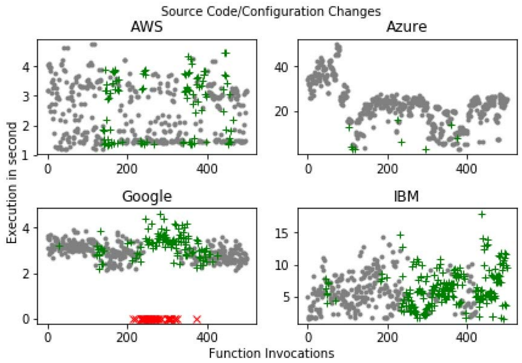

Fig. 6: Function Behavior over CD/CI

(gray dot: existing instances, green +: new instances, red x: failed instances)

invocations. In the subplot of Microsoft Azure, we observe about a small number of new instances launched during the entire invocations where it either applies any changes through existing deployments or has only a few deployments to swap. It is not visible in the figure due to a small number of instances, and we may need additional tests to determine the behavior of functions towards those events. In the subplot of Google, we observed failures and delays of function executions for those events. We need to revisit Google when the beta release of the serverless platform is ended. Unlike AWS Lambda, IBM re-deploys a new version of source code and starts to manage incoming messages on new instances that may cause excessive delays on a client-side program if new deployment takes some time to initialize.

We suggest that new deployment of a function need to be prepared with the equivalent size of function instances compared to the current loads which will prevent delaying response time in the context of concurrent invocations. Serverless framework with DevOps may enhance software development and continuous delivery through an agile function deployment and configuration as we will use multiple functions together with frequent changes.

G. Serverless versus Virtual Machine

Serverless computing does not offer either high-end computing performance or an inexpensive pricing model compared to virtual machines like Amazon EC2. Virtual machines on cloud computing have offered multiple options to scale compute resources with machine types, network bandwidth and storage performance to meet the expectation of performance requirements of a given workload which requires optimal capacity planning and system management. Serverless computing, however, aims to provide dynamic compute resources for lightweight functions without these administrations and offer cost-efficient solutions in which users pay for the execution time of functions rather than paying for the leased time of machines. Amazon, for example, has an EC2 machine choice optimized for intensive tasks with up to 128 vCPUs and a 3.8TiB size of memory with a limited number of allocations per account. AWS Lambda allows invoking a function at least a thousand of concurrency per region with a small memory allocation up to 2.8GiB (3008MB) which result in a total size of 2.8TiB. We ran an experiment in this section to demonstrate possible use cases of serverless with the understanding of the differences between these two computing resources. Serverless computing is powered by container technologies which have near zero start-up delay and deleting latency during a function life-cycle. For example, we ran a test of a NodeJS function using Apache OpenWhisk with Kubernetes, and a small Docker container (as a Kubernetes Pod) is deployed and terminated within a few milliseconds for the function invocation. The container instance was running (*warm state*) for a specified period to receive a future event message which merely consumes resources and was changed to a pause state which indicates a terminated process but reserving function data like source code and a temp directory in storage. The paused instance saves time to re-instantiate a function for the future requests without wasting compute resources. Some delays might be observed at first which is called *cold start* but a configuration can be changed to extend the idle time of the running container, or a regular wake-up invocation can be implemented as a workaround if necessary. On the contrary, virtual machines take at least a few seconds to be ready, and a high-end server type with multiple virtual CPUs and large size of memory and storage with a custom machine image may take 10 to 20 minutes to initialize. Another issue of using virtual machines is that a resource utilization needs to be handled by users to maximize values of leasing machines. If a VM is idle, the utilization rate is decreased, and if more VMs are necessary to support a tremendous amount of traffic immediately, existing VMs are overloaded which may cause performance degradation. Regarding the charge interval of leased VMs, many cloud providers have applied a per-second basis like serverless computing with some exceptions. Therefore, the particular workload would be deployed on VMs if it requires high performance compute resources for a short amount of time.

We made a cost comparison between serverless computing and traditional virtual machines because we think it would explain cost-effectiveness for specific workload deployed on these two platforms. The charging unit is different, serverless computing is based on 100 milliseconds per invocation, and a virtual machine uses either an hour or a second basis charge per instance. When we break down the cost in a second, serverless is almost ten times expensive compared to a virtual machine regarding the allocated compute resources. We ran two scripts written by a python and a JavaScript building binary trees. Table IV shows the execution time of the creating binary trees and the total cost with the rank ordered by cost-effectiveness. The result indicates that a sequential function on serverless computing would not be a good choice regarding cost-savings although it is still a simple way of deploying a function as a service. However, dynamic concurrent invocations on serverless computing will save cost against overloaded virtual machines when many event messages are requested.

H. Trigger Comparison

In this section, we measure a trigger throughput to indicate the maximum number of processing event messages in parallel. Certain triggers such as Timer and GitHub Commit are excluded because they generate a single event message as a series of procedure and these are not suitable for concurrent

TABLE IV: Building Binary Tree with Cost-Awareness

Platform	RAM	Cost/Sec	Elapsed Second	Total Cost (Rank)
AWS Lambda	3008MB	$4.897e-5	20.3	$9.9409e-4 (6)
AWS EC2 (t2.micro)	1GiB	$3.2e-6	29.5	$9.439e-05 (3)
Azure Functions	192MB	$3e-6	71.5	$2.145e-4 (4)
Azure VM	1GiB	$3.05e-6	88.9	$2.71145e-4 (5)
Google Functions	2GB	$2.9e-5	34.5	$0.001 (7)
Google Compute (f1-micro)	600MB	$2.1e-6	19.2	$4.0319e-05 (1)
IBM OpenWhisk	128MB	$2.2125e-6	34.2	$7.5667e-05 (2)

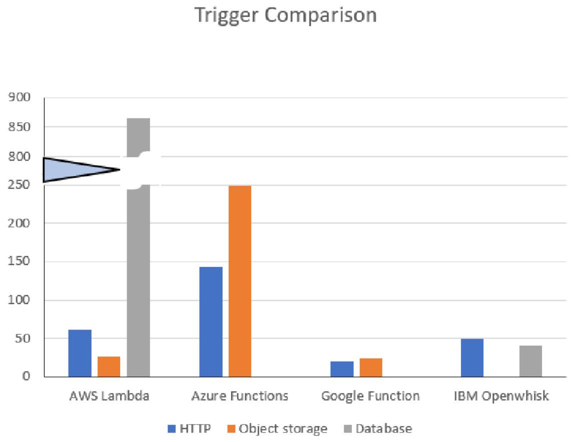

Fig. 7: Trigger Throughput

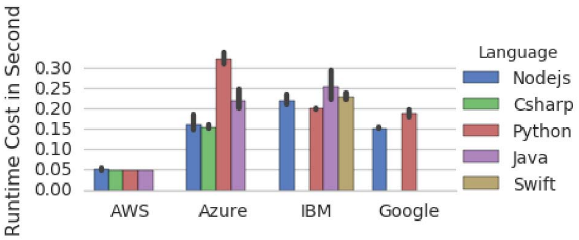

Fig. 8: Runtime Overhead (Missing bars mean a language is not supported)

function invocations. Three types of triggers are selected; HTTP, database and object storage triggers to measure the trigger throughput. In the Figure 7, Triggers in AWS Lambda show that the median throughput of the HTTP trigger is 55.7 functions per second and the object storage has the 25.16 functions per second median throughput. The database trigger in AWS Lambda has the throughput of 864.60 functions per second which is about 32 times of object storage throughput and 15 times of HTTP trigger throughput. Note that the instance of database trigger at Amazon is adjustable to deal with more event messages when it is necessary. The scale of y-axis in the figure is the number of functions processed per second. We did not compare Azure and Google Cloud database trigger as they do not have a direct trigger available to their respective functions. In the HTTP trigger, the asynchronous calls may not be supported by serverless providers, and limits of concurrent trigger processing vary as well. Details about quota and limits need to be confirmed by the providers and user accounts. As per Figure 7 we see that Microsoft Azure has the highest number of 142 invocations per second whereas Google Functions shows the least throughput as they invoke very less number of functions per second. Also, it is worth to address that all serverless providers show a linear pattern of function invocation when the number of requests is increased. We do not see any degradation of performance in handling massive requests up to 3000 concurrent invocations. We can conclude that the increase in invocation does not affect the performance.

I. Feature Comparison

The feature comparison would be helpful to the new users of serverless computing and assist the readers of this paper to understand the underlying system level information of the serverless platform. As per the Table V, AWS Lambda offers a wide range of trigger endpoints compared to the other cloud providers. We also see that the cost of usage of the serverless function is based on two metrics. First, the number of invocation of serverless functions. Second, the time is taken by a serverless function to execute and complete paired with an amount of memory in size of gigabytes allocated. Invocation to the serverless functions is cost-effective in all serverless providers if an application is executable with certain restrictions that serverless computing has. All providers have similar pricing tables, but IBM openWhisk does not charge the number of invocations whereas the other providers do charge. Google upscales regarding memory as it provides a maximum of 2 GB of memory to run a serverless function. Google also outperforms regarding the maximum execution timeout of 9 minutes which would be effective for long running jobs. IBM OpenWhisk provides the best clock speed of 2100 *4 MHz of CPU.

J. Language Support

Each serverless provider supports different programming languages thus developers can write functions with a language preference. As an interpreted language, we find Node.js, JavaScript runtime environment from all of the providers, and Python is mostly supported. Compiled languages such as Java and C# are also supported although a web-based inline editor is excluded. We assume that serverless provider intends to extend language support in the future, for example, Amazon recently added Golang and Microsoft Azure added Java as their new runtime. Table VI shows a list of supported runtimes. One other observation across different language runtimes is an overhead to complete a function. We created a second wait function in different languages and measured excessive execution time on average than a second over 100 times. Figure 8 indicates that the runtime overhead in AWS is negligible and similar whereas C# in Azure creates the least overhead among other runtimes and Python in IBM OpenWhisk shows the least standard deviation. The Node.js environment is a better choice in Google Functions, but we measured Python runtime overhead indirectly through `child_process.spawn()`. For some functions, these overheads might be sensitive to choose a language runtime since there are timeouts in executing a function (See Table V).

Item	AWS Lambda	Azure Functions	Google Functions	IBM OpenWhisk
Runtime language	Node.js, Python, Java, C#, Golang(Preview)	C#, F#, Node.js, PHP, TypeScript, Batch, Bash, PowerShell, Java(Preview)	Node.js	Node.js, Python, Java, C#, Swift, PHP, Docker
Trigger	18 triggers (i.e. S3, DynamoDB)	6 triggers (i.e. Blob, Cosmos DB)	3 triggers (i.e. HTTP, Pub/Sub)	3 triggers(i.e. HTTP,Cloudant)
Price per Memory	$0.0000166/GB-s	$0.000016/GB-s	$0.00000165/GB-s	$0.000017/GB-s
Price per Execution	$0.2 per 1M	$0.2 per 1M	$0.4 per 1M	n/a
Free Tier	First 1 M Exec	First 1 M Exec	First 2 M Exec	Free Exec / 40,000GB-s
Maximum Memory	3008MB	1536MB	2048MB	512MB
Container OS	Linux	Windows NT	Debian GNU/Linux 8 (jessie)	Alpine Linux
Container CPU Info	2900.05 MHz,1 core	1.4GHZ	2200 MHz, 2 Processor	4 CPU cores,2100.070 MHz
Temp Directory (Path)	512 MB (/tmp)	500 MB (D:\Local\Temp)	(/tmp)	(/tmp)
Execution Timeout	5 minutes	10 minutes	9 minutes	5 minutes
Code Size Limit	50 / 250 MB (compressed/uncompressed)	n/a	100MB (compressed) for sources. 500MB (uncompressed) for sources plus modules.	48 MB

TABLE V: Feature Comparison

Language	AWS	Azure	Google	IBM
Python	2.7, 3.6	2.7	2.7*	2.7, 3.6
Java	8	8	-	8
NodeJS	4.3, 6.10	6.11, 8.4	6.11.5	6, 8
C#	1, 2	1, 2	-	-
Others	Go 1.x	F# 4.6	-	Docker

TABLE VI: Supported Language Runtime

* Internal Support

III. Use Cases

There are several areas where serverless can play an important role in research applications as well as in a commercial cloud. Big Data map-reduce application can be executed similarly but a cost-effective way of deployment as we discuss implementations of the big data applications in a series of serverless functions with cloud object storage and databases [7], [8]. We ran some Big Data Benchmark tests by scanning 100 text files with different queries and aggregating 1000 text files with a group by and sum queries which show that certain applications are executable on serverless framework relatively easily and fast compared to running on server-based systems. Image processing for CDN is applicable by the serverless framework to process thumbnails of the images concurrently. Internet Of Things (IoT) is also one of the use cases for serverless framework because IoT devices typically have a small computing power to process all the information and need to use remote resources by sending event messages which are a good fit for serverless computing. IoT devices may invoke a function using a policy. For example, in case of a datacenter, a cooling facility is essential for proper functioning of servers. When cooling is not working properly, a thermostat invokes a function to calculate live migration of allocated workloads to other data centers and determine shutdown of servers on particular areas. We hope to see several use cases of serverless computing as the main type of cloud-native application development soon.

IV. Discussion

Serverless computing would not be an option for those need high-end computing power with intensive I/O performance and memory bandwidth because of its resource limitation, for example, AWS Lambda only provides 3GB of memory and 2 virtual cores generating 40 flops with 5 minutes execution timeouts. These limitations will be adjusted once serverless environments are mature and there are more users but bulk synchronous parallel (BSP) and communication-free workloads can be applied to serverless computing with its concurrent invocations. Additional containers for concurrent function invocations reduce a total execution time with a linear speed up, for example, a function invocation divided into two containers decreases an execution time in half. There are also overlaps and similarities between serverless and the other existing services, for example, Azure Batch is a job scheduling service with an automated deployment for a computing environment. AWS Beanstalk [9] is deploying a web service with automated resource provisioning.

V. Related Work

We have noticed that there were several efforts to utilize existing serverless computing for parallel data processing using concurrent invocations. PyWren [4] is introduced in achieving about 40 TFLOPs using 2800 Amazon Lambda invocations. The programming language runtime on serverless computing has been discussed in the recent work [10]. Deploying scientific computing applications has been conducted with experiments to argue the possible use cases of serverless computing for adopting existing workload [11] with its tool [12]. McGrath et al [13] shows serverless comparison results for function latency among production serverless computing environments including Microsoft Azure Functions but it was not a comprehensive review, such as testing CPU, network and a file I/O, and several improvements have been made later such as an increment of memory allocation such as 3GB on Amazon Lambda and additional runtime support such as Java on Azure Functions and Golang on Amazon. OpenLambda [14] discusses running a web application on serverless computing and OpenWhisk is introduced for mobile applications [5]. Since then several offerings on the serverless framework with new features have been made. Kubeless [15] is a Kubernetes-powered open-source serverless framework, like Fission [16]. Zappa [17] is a python based serverless powered on Amazon Lambda with additional features like keeping the warm state of a function by poking at a regular interval. OpenFaaS is serverless for Docker and Kubernetes with language support for Node.js, Golang, C#, and binaries like ImageMagicK. Oracle also started to support open source serverless platform, Fn project [18]. In this work, we have investigated four serverless computing environments in production regarding the CPU performance, network bandwidth, and a file I/O throughput and we believe it is the first evaluation across Amazon Lambda, Azure Functions, Google Functions and IBM OpenWhisk.

VI. Conclusion

Functions on serverless computing can process distributed data applications by quickly provisioning additional compute resources on multiple containers. In this paper, we evaluated concurrent invocations on serverless computing including Amazon Lambda, Microsoft Azure Functions, Google Cloud Functions and IBM Cloud Functions (Apache OpenWhisk). Our results show that the elasticity of Amazon Lambda exceeds others regarding CPU performance, network bandwidth, and a file I/O throughput when concurrent function invocations are made for dynamic workloads. Overall, serverless computing scales relatively well to perform distributed data processing if a divided task is small enough to execute on a function instance with 1.5GB to 3GB memory limit and 5 to 10 minute execution time limit. It also indicates that serverless computing would be more cost-effective than processing on traditional virtual machines because of the almost zero delay on boot up new instances for additional function invocations and a charging model only for the execution time of functions instead of paying for an idle time of machines. We recommend researchers who have such applications but running on traditional virtual machines to consider migrating on functions because serverless computing offers ease of deployment and configuration with elastic provisioning on concurrent function invocations. Serverless computing currently utilizes containers with small computing resources for ephemeral workloads, but we believe that more options on computing resources will be available in the future with fewer restrictions on configurations to deal with complex workloads.

Acknowledgment

This work was partially supported by NSF CIF21 DIBBS 1443054 and the Indiana University Precision Health initiative. We thank Intel for their support of the Juliet system, and extend our gratitude to the FutureSystems team for their support with the infrastructure. We thank the WOSC workshop series for focusing attention on this area.

References

[1] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service (faas) in industry and research," *arXiv preprint arXiv:1708.08028*, 2017.

[2] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017.

[3] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads." in *NSDI*, 2017, pp. 363–376.

[4] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," *arXiv preprint arXiv:1702.04024*, 2017.

[5] I. Baldini, P. Castro, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and P. Suter, "Cloud-native, event-based programming for mobile applications," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 2016, pp. 287–288.

[6] H. Lee, "Building software defined systems on hpc and clouds," 2017.

[7] G. C. Fox and S. Jha, "A tale of two convergences: Applications and computing platforms."

[8] S. Kamburugamuve and G. Fox, "Designing twister2: Efficient programming environment toolkit for big data."

[9] A. Amazon, "Elastic beanstalk," 2013.

[10] M. Maas, K. Asanović, and J. Kubiatowicz, "Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 2017, pp. 138–143.

[11] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc," in *Latin American High Performance Computing Conference*. Springer, 2017, pp. 154–168.

[12] J. Spillner, "Snafu: Function-as-a-service (faas) runtime design and implementation," *arXiv preprint arXiv:1703.07562*, 2017.

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

[14] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," *Elastic*, vol. 60, p. 80, 2016.

[15] Kubeless, "A kubernetes native serverless framework," 2017.

[16] Fission-Platform9, "Serverless with kubernetes," 2017.

[17] Zappa, "Serverless python web services," 2017.

[18] Fn-Project, "a container native apache 2.0 licensed serverless platform," 2017.