Evaluation of Network File System as a Shared Data Storage in Serverless Computing

Jaeghang Choi
Dept. of Computer Science, Kookmin Univ.
Seoul, S. Korea
chl8273@kookmin.ac.kr

Kyungyong Lee
Dept. of Computer Science, Kookmin Univ.
Seoul, S. Korea
leeky@kookmin.ac.kr

Abstract

Fully-managed cloud and Function-as-a-Service (FaaS) services allow the wide adoption of serverless computing for various cloud-native applications. Despite the many advantages that serverless computing provides, no direct connection support exists between function run-times, and it is a barrier for data-intensive applications. To overcome this limitation, the leading cloud computing vendor Amazon Web Services (AWS) has started to support mounting the network file system (NFS) across different function run-times. This paper quantitatively evaluates the performance of accessing NFS storage from multiple function run-times and compares the performance with other methods of sharing data among function run-times. Despite the great qualitative benefits of the approach, the limited I/O bandwidth of NFS storage can become a bottleneck, especially when the number of concurrent access from function run-times increases.

ACM Reference Format:
Jaeghang Choi and Kyungyong Lee. 2020. Evaluation of Network File System as a Shared Data Storage in Serverless Computing. In *Workshop on Serverless Computing (WoSC'20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429880.3430096

1 Introduction

Serverless computing architecture is adopted in many cloud-native applications. Fully managed "library" services and FaaS are major components that make serverless computing applications feasible [3]. Fully managed library services provide various functionalities that are essential in applications development, such as object storage, notification, database, and HTTP web endpoint, without incurring manual resource management overhead. They allow users to focus on core application development while providing inherently highly available services with auto-scaling and load-balancing. However, most fully managed cloud services do not allow customization, and application developers should rely on services that are provided by vendors. FaaS can overcome the shortcoming by allowing users to execute custom-implemented source code on a fully managed run-time environment.

Public cloud service vendors similarly support FaaS (e.g., AWS Lambda, and Google Cloud Functions). Each service has uniqueness in the integration with the other fully managed library services, function run-time management mechanisms [1, 16], and billing policies.

Despite the wide adoption of FaaS for serverless application development, it has a few major limitations, such as no peer-to-peer (P2P) communication between function run-time support [7, 8, 12], performance variation, including the cold-start issue [2, 6, 9, 15], no specialized hardware support [3], and a lack of generally available workloads [5, 13].

Among the limitations, the lack of P2P communication support among function run-times is a great obstacle to developing a diverse set of serverless applications, especially for data-intensive applications that require various communication patterns [3]. To overcome the limitation, various object storage services have been used to store intermediate datasets temporarily. In the cloud, users can choose a different level of object storage service concerning price and latency performance [8, 12], which uses either cheap/slow disk-based object storage services (Amazon S3) or expensive/fast RAM-based caching services. The current approaches support file operations within an object level, and an entire file should be uploaded or downloaded from a FaaS run-time. To better support data-heavy applications in serverless computing, block storage services should be supported within a FaaS run-time that allows manipulation on the byte level within a file. With the necessity, AWS has started to support mounting an NFS in function run-times so that they can communicate through the file system and share data while performing file operations on the byte level [4].

The adoption of NFS in an FaaS environment can greatly improve the usability of the current FaaS applications by allowing multiple functions to share states across distinct executions. Using a mounted disk access point from a function

WoSC'20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8204-5/20/12…$15.00
https://doi.org/10.1145/3429880.3430096

run-time, developers are expected to perform disk and network I/O operations transparently without using software development kits (SDKs) to access cloud library services, especially for data-intensive applications. In the very early stages of supporting NFS in FaaS, it is important to understand the performance characteristics to envision a proper architecture when developing applications using serverless computing. To achieve the goal, we performed experiments thoroughly using AWS Lambda with the elastic file system (EFS) as an NFS solution. The experimental results reveal that appropriate NFS storage bandwidths should be provisioned to provide multiple function run-times with comparable performance with an object storage service or local storage.

2 FaaS and Storage Services

Serverless computing is a cloud service execution model in which service providers run servers and dynamically manage the servers in the direction of maximizing resource utilization while providing scalability and fault tolerance. Fully managed cloud services are major components that make serverless computing feasible. For example, AWS provides Lambda as a FaaS to allow users to write custom source codes and execute them on managed servers. In addition, AWS provides fully managed HTTP end-point service (API Gateway), distributed key-value storage (DynamoDB), object storage (S3), and many more services.

Among many fully managed services, FaaS expedites the adoption of serverless computing for many cloud-native applications because it provides great flexibility during the programming phase. In AWS Lambda, an user first registers a custom-implemented source code written in *Java, Python, JavaScript, Go, Ruby,* or *C#*. A registered function [1] works in an event-driven manner, and users can register various fully managed event sources. For instance, if a function must be invoked upon the user request, the HTTP endpoint (API Gateway) can be registered.

During the registration, the user must determine the maximum memory size between *128 MB* and *3008 MB* that a function run-time can use, and the maximum time a registered source code executes. The maximum memory size is a key factor that affects the function execution time and billing. The amount of the CPU clock is allocated proportionally to the configured memory size, and the completion time of CPU-intensive tasks is primarily dependent on the configured memory size [9, 15]. Kim et al. [6, 7] discovered that, unlike CPU resources, disk and network I/O resources are not allocated proportionally to the memory size. However, the proportionally allocated CPU clock affects I/O performance and results in a performance difference. Such characteristics are common to most FaaS vendors (AWS Lambda and Google Cloud Function) with slight configuration differences (Azure Functions). For local data storage, AWS Lambda provides *500 MB* of disk space that a function run-time can use exclusively. An additional *250 MB* of layer storage is provided to allow function run-times to share static and read-only files, such as shared libraries.

The adoption of serverless computing architecture is accelerating, but the majority of practical applications are limited to embarrassingly parallel stateless tasks, function compositions, or cloud service orchestrations [3]. From academia, various data-intensive serverless applications have been proposed in the literature. They include machine learning and deep learning modeling and inference, distributed linear algebra, video processing, query processing, graph mining, and many others. Due to the nature of data-heavy applications, diverse types of inter-communication among function run-times are necessary. However, the current public FaaS systems do not support direct connection-oriented communication between run-times, and users must rely on external data storage services to share datasets.

To store and share intermediate datasets among function run-times, users can rely on cloud-native services, which are the slow and cheap disk-based object storage service and the fast and expensive memory-based caching service. In AWS, S3 is the cheapest option to store and share a file object. It is a fully managed object storage service that AWS provides with scalable and fault-tolerance resource management. It supports object auditing via version tracking and enhances security by supporting server-side encryption. It integrates with Lambda smoothly in an event-driven manner. In contrast, fully managed elastic caching services can provide similar functionality of S3 with higher cost and lower latency. An advantage of using cloud-native data storage services is the smooth integration with FaaS because they provide well-established user interfaces through web and SDKs, which developers can easily use in the FaaS source code.

The cloud-native disk-based object storage and RAM-based key-value storage services have distinct characteristics regarding price and access latency. To overcome the challenge, hybrid architecture intermediate storage services have been proposed [12, 14]. Locus [12] proposed using slow (S3) and fast (Redis) cloud storage together, considering the characteristics of shuffle tasks among FaaS executions. The fast storage service is used to process small chunks with high IOPS, whereas the slow storage service is used to store large chunks. Apache Crail [14] aims to act as a very fast ephemeral data storage and sharing service. The proposed architecture is designed from the ground up for modern high-performance storage (NVMe SSD) and networking (RDMA) hardware, and it can be accessed directly from FaaS run-times. The proposed systems demonstrate superb performance compared to cloud-native storage services, but users must manually install the services to access them from serverless applications. Furthermore, the systems support only object-level access that can limit usage scenarios.

[1]We note the source code and function interchangeably.

Storage Type	Native	Speed	Price	Access	Share	Interface	Boundary	Persistence	Event
Object (S3)	Yes	Slow	$	Object	Yes	SDK	Global	Permanent	Yes
Caching (Redis)	Yes	Fast	$$$	Object	Yes	SDK	Region	Permanent	No
High Performance (Crail)	No	Very Fast	$$$$	Object	Yes	SDK	Global	Permanent	No
Local disk	Yes	Fast	0	Block	No	POSIX	Local	Temporary	No
NFS (EFS)	Yes	Fast	$$	Block	Yes	POSIX	Region	Permanent	No

Table 1. File storage service accessible from FaaS run-times

To widen the adoption of serverless computing for various applications, supporting P2P communication among function run-times and block-level shared storage service is necessary [3]. Mounting an NFS from multiple function run-times can solve the limitation that the current FaaS systems impose. In the context, AWS Lambda has started to support mounting its managed NFS service, EFS [4]. The user can easily create an EFS endpoint through the AWS web console, CLI, or SDK. The user can mount an EFS on servers in the on-premise data center or Amazon EC2, a computation service. Additionally, EFS provides two throughput modes: bursting and provisioned. In the bursting mode, the throughput of an EFS endpoint scales with the consumed storage size and can reach 100 MB/S per *1 TB* of consumed storage. In the provisioned mode, the user can purchase additional throughput that an EFS endpoint can provide. For pricing, the consumed storage size determines the cost in the bursting mode, and the throughput assigned by the user determines the cost in the provisioned mode.

To use EFS within Lambda function run-times, the user must create an EFS disk and mount the disk in the Lambda menu. During the setup, the user can decide the mount point path in a Lambda run-time. Within the Lambda source code, the user can access the shared data storage using *POSIX* file system APIs and perform file operations on the byte level, which is similar to accessing the local storage disk. For other object storage services, the developer must use specialized SDKs to interact with external storage services with proper access control permission. A mounted EFS disk is accessible from multiple function run-times simultaneously sharing available bandwidths of an EFS disk.

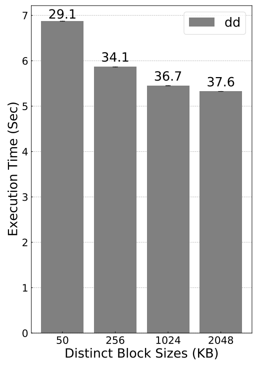
Table 1 summarizes the characteristics of various file storage services available to function run-times whose values are shown in the *Storage Type* column. The term *Native* indicates whether the storage service is supported by the cloud service vendor in a fully managed manner. For services whose value is *No* for the column, users must install and manage a cluster of machines, such as the Crail cluster. The *Speed* and *Price* columns indicate the latency and cost of using the storage service, respectively. The *Access* column indicates the level of access that the storage service offers. The value is either *Object* or *Block*, where object means that the file is the smallest unit the user can handle, whereas block indicates that the user can perform byte-level operations in the file. The *Share* column indicates whether the storage service allows simultaneous file operations from multiple function run-times. The *interface* column shows the method through which the function run-time accesses the storage service. For *SDK*, the user can use the *boto3* library in the Python program. For *POSIX*, users can use the POSIX file system APIs directly from the function source code. The *Boundary* column indicates the location of function run-times and storage services that can communicate without a specialized setup. The *Global* value means that the storage service is globally accessible, and the *Local* indicates that the storage service is accessible only from the function run-time. The *Region* value means that the function run-time can access the storage service if located in the same region, which is a term to group geographically separated locations used in most cloud vendors. In most cases, users can create a specialized setup, such as virtual private cloud (VPC) peering, to allow inter-region communications, but it incurs complex resource management and additional cost. The *Persistence* column indicates whether the dataset stored in the corresponding storage service is permanent or not. The *Event* column indicates whether the storage service can generate an event to the function run-time whenever a file-level update occurs. We considered managed events generated from the storage service and excluded user-implemented event notifications.

3 Evaluations

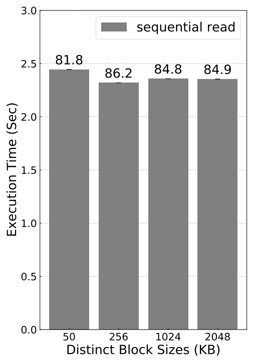
Compared to other cloud storage services presented in Table 1, supporting NFS from the function run-time is a relatively new feature, and it is important to understand the quantitative characteristics of using NFS from the function run-time. To achieve the goal, we evaluated the performance of accessing an AWS EFS endpoint after mounting it in an AWS Lambda run-time. The evaluation focuses on the file size effect, various application scenarios, and scalability.

3.1 Experiment Environments

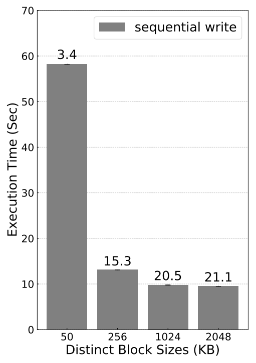
To evaluate the performance of NFS with FaaS, we set a test environment using AWS EFS and Lambda. We placed all necessary resources in the *us-west-2* region including the AWS S3 master location. We created EFS storage in *provisioned throughput* mode, setting the bandwidth to 100 MB/S, and we created a Lambda function mounting the EFS endpoint.



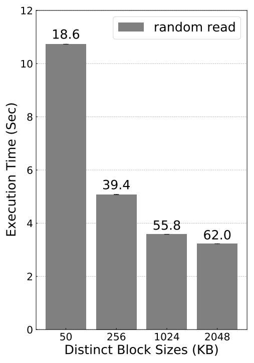
(a) dd



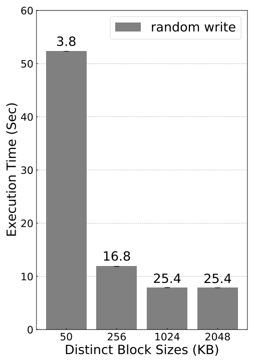
(b) Sequential Read



(c) Sequential Write



(d) Random Read



(e) Random Write

Figure 1. AWS Lambda with EFS performance differences for various benchmarks with distinct file sizes

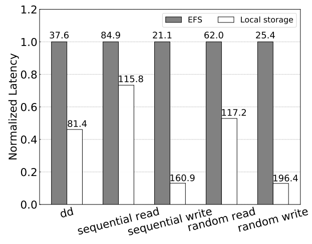
To make realistic workload scenarios, we used FunctionBench [5]. Among the benchmarks in the shared repository, we used I/O intensive workloads, *dd*, *sequential read*, *sequential write*, *random read*, and *random write*. All workloads have a file size and block size configuration. The file size (S_f) indicates the total size on which a workload operates, and the block size (S_b) indicates the unit size of each workload where $S_f \geq S_b$ always stands. For example, if S_f is 200 MB and S_b is 2 MB, a *random write* workload generates 50 byte-wise starting points spaced at 2 MB each. The workload randomly selects a starting point among 50 values without replacement 50 times so that every byte can be written in a file of size S_f. Unlike *random* workloads, *sequential* workloads process each block in sequential order. The *dd* workload reads input data from storage and writes to a new file in the same storage service. In the experiments, we set the target file size as 200 MB because it is the maximally available size that meets the restrictions of components involved in the experiments (the *layer* size).

3.2 Impact of Block Sizes

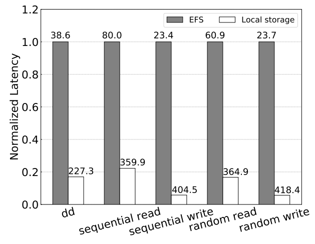
Figure 1 shows the latency of completing various workloads (*dd* - 1a, *sequential read* - 1b, *sequential write* - 1c, *random read* 1d, and *random write* - 1e) in the vertical axis with differently configured block sizes shown in the horizontal axis. The throughput value (MB/S) is written on each bar. *Random* and *sequential write* operations have drastic performance differences regarding different block sizes. Although the maximum available bandwidth of an EFS disk is 100 MB/S, a single function run-time has difficulty using the full bandwidth.

3.3 Comparing NFS with a Local Storage

To understand how NFS storage performs with FaaS, we compared the latency of completing the aforementioned workloads for EFS and local storage of Lambda function runtimes in Figure 2. The experiment shows the results of 2 MB



(a) Lambda RAM 512MB



(b) Lambda RAM 2GB

Figure 2. Comparing performance of function local storage under different Lambda RAM configuration with 2MB block

block sizes configured with different Lambda memory: 512 MB Lambda RAM size in Figure 2a and 2 GB Lambda RAM size in Figure 2b. Of five workloads in 3.2, we present the results that have stark differences across different experiment settings. To compare the performance, we normalized the latency of the local storage to EFS, and the throughput values (MB/S) are shown on the top of each bar.

Comparing the two figures, EFS does not gain a performance benefit by increasing the configured RAM size of Lambda, whereas the benefit of local storage is noticeable. We believe that network performance is the major bottleneck that determines the overall performance. This finding supports the findings that Park et al. discovered through I/O performance evaluation [10], which showed that increasing the Lambda RAM size results in a performance gain for the local storage I/O but did not make much difference for network-heavy tasks. The read throughput of the local disk is exceptionally high in some cases, and we believe that it is due to reading from the page cache. We did our best to avoid reading from a page cache, but it was not possible because the current function run-time does not provide root permission to an underlying operating system.

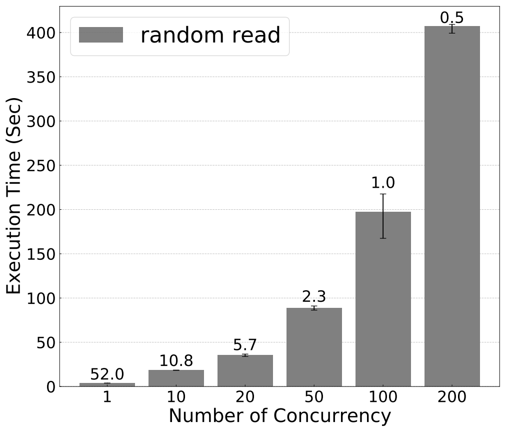


Figure 3. Evaluation of scalability of EFS with Lambda

3.4 Scalability

It is important to understand how Lambda with EFS scales as the degree of concurrent access increases. The vertical axis of Figure 3 illustrates the latency of accessing EFS storage from a different number of concurrent function run-times, which is expressed in the horizontal axis. We present the results from a *random read* workload only because other workloads show a similar pattern. A numeric value on each bar represents the average bandwidth of the function run-time, and the total aggregated bandwidth can be calculated by multiplying the per function bandwidths and the total number of concurrent executions on the x-axis. Increasing the degree of concurrent executions results in higher latency to complete a given workload. In the experiment, we used the *provisioned throughput* of the EFS set to 100 MB/S. When a single function accesses EFS storage, it cannot fully use the bandwidth. As the number of concurrent function executions increases, the total aggregated bandwidth is capped at the preconfigured bandwidths. This experiment result implies that users should be cautious regarding the available bandwidths of EFS storage, especially when multiple functions are accessed. In using the bursting mode, users can calculate the available bandwidth and bursting time from the storage size in use. The provisioned throughput mode provides more stable disk throughput with extra cost proportional to the provisioned disk throughput.

To evaluate the performance effect of the limited bandwidth of EFS storage in a practical workload scenario, we performed image augmentation tasks with different image input and output locations. Image augmentation is almost a de-facto standard process for image classification deep learning algorithms because it improves a model's generality by expanding the training dataset through the transformation of input images with various effects [11]. In the experiment, we created a FaaS-based image augmentation environment. Input images to be trained for arbitrary image classification tasks are assumed to exist in shared file storage, AWS S3 as an object storage service, or EFS. A Lambda function fetches an input image from storage and performs five image transformation tasks (*flip left-right, flip top-bottom, grayscale, rotate 90 degrees, rotate 180 degrees*) in parallel using Python's *Pillow* library. After the image augmentation tasks, five different output images are uploaded to the shared storage service (either S3 or EFS) so that they can be used to train a model.

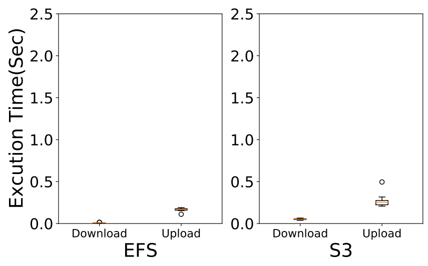
Figure 4 depicts the latency of completing the download and upload steps. To compare the scalability of S3 and EFS, we performed experiments with a different number of input images, 10, 50, and 100, whose results are presented in Figures 4a, 4b, and 4c, respectively. A single function run-time processes an image, and several function run-times corresponding to the number of images are executed in parallel. In each function, due to the nature of the augmentation tasks, a download operation happens once, and an upload option happens five times (the number of augmentation effects). To compare the performance under different loads, we fixed the vertical axis maximum value at 2.5 seconds.

When the number of images to process is relatively small (10 images), the latency to complete download and upload tasks is similar when using S3 and EFS as a storage service. However, as the degree of parallel execution increases, the performance degradation of EFS becomes noticeably significant. When 100 images are processed in parallel, EFS took an average of 1.37 seconds and S3 took 0.22 seconds to complete the upload task. To quantitatively understand the performance variation, we calculated the coefficient of variation (CoV). The CoV is defined as the ratio of the standard deviation to the average. The metric presents the degree of dispersion around the mean value. A higher value means great variability. The CoVs of the upload task for EFS are 0.13, 0.29, and 0.41 and those of S3 are 0.31, 0.14, and 0.22 when the number of input images is 10, 50, and 100, respectively. In addition to the poor performance of EFS with a higher load, the performance of EFS is not stable with multiple function run-time access.

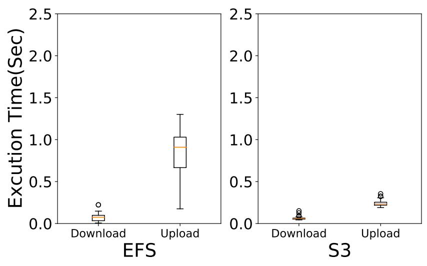
The poor performance of EFS stems from the limited total aggregated bandwidth that was set as 100 MB/S in the experiments. In using EFS, users can increase the throughput by purchasing additional capacity. Contrary to EFS, the object storage service S3 provides consistent bandwidths without incurring additional costs of which users should be aware.

4 Conclusion and Future Work

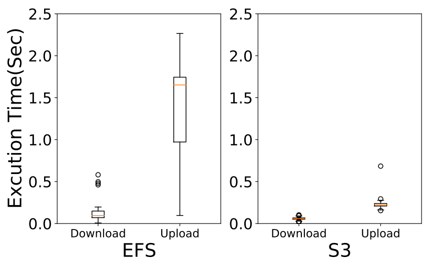
We quantitatively evaluated the performance of using NFS storage from multiple function run-times as a method to share data. Despite the addressed qualitative advantages, the available bandwidth of NFS storage can become a major performance bottleneck when multiple functions access simultaneously, and users should plan the bandwidth allocation accordingly, considering the application requirements.



(a) 10 input images



(b) 50 input images



(c) 100 input images

Figure 4. Image augmentation task in Lambda with different input/output sources with a different number of input images

Mounting NFS storage support from multiple function runtimes is a very recent feature, and we believe that the support will expand. In using NFS storage with FaaS, users can purchase additional I/O bandwidth, and further research is necessary to better understand the cost and performance trade-off in the domain. The evaluations can include further scenarios from the perspective of service vendors, workloads, and shared storage services.

Acknowledgement

This work is supported by the National Research Foundation of Korea Grant funded by the Korean Government (NRF-2020R1A2C1102544, NRF-2015R1A5A7037615), and the ICT R&D program of IITP (2017-0-00396 and 2016-0-00021).

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[2] Robert Cordingly, Hanfei Yu, David Perez Varik Hoang, David Foster, Zohreh Sadeghi, Rashad Hatchett, and Wes J Lloyd. 2020. Implications of Programming Language Selection for Serverless Data Processing Pipelines. (2020).

[3] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[4] AWS What is new. last accessed August. 2020. AWS Lambda support for Amazon Elastic File System now generally available. https://aws.amazon.com/about-aws/whats-new/2020/06/aws-lambda-support-for-amazon-elastic-file-system-now-generally-/

[5] J. Kim and K. Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504. https://doi.org/10.1109/CLOUD.2019.00091

[6] J. Kim and K. Lee. 2020. I/O Resource Isolation of Public Cloud Serverless Function Runtimes for Data-Intensive Applications. *Cluster Computing* (2020). https://doi.org/10.1007/s10586-020-03103-4

[7] Jeongchul Kim, Jungae Park, and Kyungyong Lee. 2019. Network Resource Isolation in Serverless Cloud Function Service. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*.

[8] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[9] H. Lee, K. Satyam, and G. Fox. 2018. Evaluation of Production Serverless Computing Environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, Vol. 00. 442–450. https://doi.org/10.1109/CLOUD.2018.00062

[10] J. Park, , H. Kim, and K. Lee. 2020. Evaluating Concurrent Executions of Multiple Function-as-a-Service Runtimes with MicroVM. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*.

[11] Luis Perez and Jason Wang. 2017. The Effectiveness of Data Augmentation in Image Classification using Deep Learning. arXiv:cs.CV/1712.04621

[12] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[13] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[14] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of Temporary Storage in the NodeKernel Architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 767–782. https://www.usenix.org/conference/atc19/presentation/stuedi

[15] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang

[16] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotcloud19/presentation/young