
Proceedings Paper:
Predoaia, Ionut and García-López, Pedro (2025) A Cloud-Agnostic Serverless Architecture for Distributed Machine Learning. In: Proceedings - 2024 IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT 2024. 11th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, BDCAT 2024, 16-19 Dec 2024 International Symposium on Big Data Computing. IEEE, ARE, pp. 131-140.

https://doi.org/10.1109/BDCAT63179.2024.00032

A Cloud-Agnostic Serverless Architecture for Distributed Machine Learning

Ionut Predoaia
University of York
United Kingdom
ionut.predoaia@york.ac.uk

Pedro García-López
Universitat Rovira i Virgili
Spain
pedro.garcia@urv.cat

***Abstract*—Serverless computing has shown vast potential for big data analytics applications, especially involving machine learning algorithms. Nevertheless, little consideration has been given in the literature to cloud-agnostic serverless architectures that leverage existing parallel implementations of machine learning algorithms. This work bridges this gap by proposing a multi-cloud serverless architecture for distributed machine learning, that enables machine learning engineers without cloud computing expertise to effortlessly port already implemented parallel machine learning algorithms to serverless, whilst overcoming vendor lock-in. In this work, two stateful machine learning algorithms have been ported to serverless, *k*-means clustering and logistic regression. The serverless implementation of *k*-means provided superior performance and scalability compared to a serverful implementation when using a number of workers that is equal to or slightly lower than the total number of vCPUs available on the VM running the serverful implementation. Additionally, it achieved an 87-fold speedup compared to a sequential implementation. Moreover, two storage designs of the shared state will be proposed for the serverless implementations, one that requires locks for updating the shared state, and another that is lock-free. Our experimental evaluation demonstrates that the performance of the lock-free serverless implementation of *k*-means declines with the increase in the number of clusters.**

***Index Terms*—Distributed Machine Learning, Big Data, Serverless Architectures, Cloud Agnostic, Multicloud, Lithops**

I. Introduction

The rise of serverless computing has widely propagated the democratization of massive-scale data parallelism. By leveraging serverless computing, cloud users can today launch thousands of concurrent stateless functions to run big data analytics workloads, without the need of complex cluster management and the burden of managing and provisioning cloud resources. One can seamlessly launch thousands of cores on demand, to enable the parallel execution of machine learning (ML) algorithms over data sets in the order of terabytes, that could typically not be stored on a single machine.

A multitude of research efforts have been carried out in the context of serverless machine learning, related to stateful applications requiring shared state [1], [2], frameworks and prototypes for serverless ML [3]–[6], tradeoff analysis between ML on serverless and serverful [7], [8], neural networks [9], [10], distributed optimizations [11], [12], and others focused on details regarding implementation [13] and architecture [14], [15]. Nevertheless, the prior works have not focused on a simplified cloud-agnostic serverless architecture that leverages existing parallel implementations of ML algorithms. This paper aims to close this gap by proposing a serverless architecture that enables ML engineers without cloud computing expertise to effortlessly port already implemented parallel ML algorithms to serverless, whilst overcoming vendor lock-in, by minimally modifying the code of the algorithms. Particularly, we are interested in stateful ML algorithms, that require shared state when executed in a distributed manner. As such, two stateful ML algorithms will be ported to serverless in this work, *k*-means clustering and logistic regression. For instance, the *k*-means clustering algorithm is highly parallelizable, however, it requires regular communication between workers at the end of each iteration to update and retrieve the new centroids.

Our serverless architecture has been instrumented through Lithops [16], a multi-cloud framework that enables untrained users to run single-machine Python code at scale in the cloud. Lithops provides an Application Programming Interface (API) that mimics the Python *concurrent.futures* [17] API. Thus, engineers can take existing Python-based parallel ML algorithms that rely on *concurrent.futures* and port them to serverless via Lithops through minimal code modifications. Our serverless implementation of *k*-means provided superior performance and scalability compared to a serverful implementation when using a number of workers that is equal to or slightly lower than the total number of virtual CPUs (vCPUs) available on the virtual machine (VM) running the serverful implementation. Furthermore, it achieved an 87-fold speedup compared to a sequential implementation. Moreover, two storage designs of the shared state will be explored, by using a serverless implementation that requires locks, and another that does not require locks, for updating the shared state.

Section II presents the necessary background. Section III introduces Lithops, a distributed computing framework. Section IV presents our serverless architecture and implementation. Section V evaluates and validates our work. Section VI presents related work. Section VII concludes the paper and provides future work directions.

II. Background

Serverless computing is an emerging paradigm for the deployment of software applications in the cloud. When using the serverless paradigm, servers do exist, however, their provisioning and management is handled by the cloud provider.

```
1  import lithops
2
3  def my_function(x):
4    return x + 7
5
6  if __name__ == "__main__":
7    data = [3, 6, 9]
8    fexec = lithops.FunctionExecutor()
9    fexec.map(my_function, data)
```

Listing 1. Example of a Lithops-Based Program

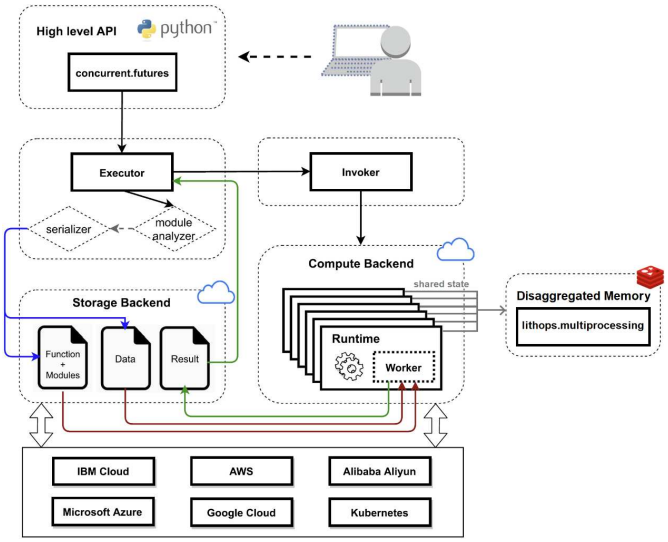

Fig. 1. High-Level Architecture of Lithops

With serverless computing, customers simply upload their code to a cloud platform, and then, the platform executes the code on the customer's behalf as needed at any scale [18]. Serverless computing has brought the shift toward a disaggregated data center, where each resource type (e.g., compute, memory, storage) is built as a standalone resource blade, and a network fabric interconnects the resource blades [19]. Hardware resource disaggregation is a technique for decomposing general-purpose monolithic servers into segregated, network-attached resource pools, each of which can be built, managed, and scaled independently [20].

Function as a Service (FaaS) is a serverless computing model in which the unit of computation is a function that is executed in response to triggers, e.g., events and HTTP requests [21]. Serverless functions have the ability to scale to zero and have almost infinite on-demand scalability. The functions are run in stateless compute containers, that are created and destroyed by the FaaS provider depending on the runtime need. The execution duration of a function is limited to a short amount of time, and after the timeout, the function is terminated, e.g., the maximum timeout for AWS Lambda is 15 minutes. By leveraging disaggregated compute resource services (e.g., AWS Lambda), one can benefit from hundreds of compute units, i.e., vCPUs, to parallelize algorithms, where one vCPU typically equates to one serverless function. Note that serverless functions are not directly network-addressable, hence concurrent serverless functions cannot communicate with each other to share state.

Backend as a Service (BaaS) is a serverless computing model in which engineers outsource the behind-the-scenes aspects of software applications, such as database management and cloud storage. An example of cloud storage BaaS is Amazon S3, which is a disaggregated object storage service that provides scalability, data availability, security, and performance. Data is stored in Amazon S3 as objects within buckets, where an object is a file along with any metadata that describes the file, and a bucket is a container for objects [22].

A disaggregated memory resource service (i.e., Redis) has been used in this work for managing shared state. Redis is an in-memory data structure store that can be used as a distributed in-memory key-value database, cache and message broker.

III. Lithops : A Distributed Computing Framework

Lithops is a multi-cloud distributed computing framework that enables engineers to run unmodified single-machine Python code at scale over the main cloud platforms, such as AWS, Microsoft Azure and Google Cloud. Therefore, the framework avoids vendor lock-in by providing portability across multiple cloud providers via its cloud-agnostic architecture. Code is transparently deployed into serverless functions, without requiring knowledge of how the deployment is carried out. This has the benefit of lowering the entry barrier for novice cloud users, who may often be puzzled by the wide array of possible choices (e.g., VM instance type and cluster size) that can be made regarding the execution of a simple application in parallel [16].

A. Architecture

Figure 1 captures the high-level architecture of Lithops. In the following, the main components of Lithops [16] are presented by using Listing 1 as an example.

Lithops makes use of a *compute backend* to run parallel jobs and a *storage backend* to store the input and output data of the jobs. One can run Lithops in a serverless context, in which the compute backend is a FaaS platform (e.g., AWS Lambda) and the storage backend is a BaaS service (e.g., Amazon S3), such that the compute and storage resources can scale independently of each other. Alternatively, one can run Lithops on a local machine (i.e., in *localhost* mode), in which the compute backend is represented by local parallel processes and the storage backend is represented by the local file system.

The *Executor*, upon a Lithops API call (e.g., the call to the `map` method from line 9), serializes and uploads the single-machine code (e.g., `my_function` from lines 3–4) and the input data (e.g., the `data` variable defined at line 7) from the local machine to the storage backend. Note that the single-machine code is a user-defined function (UDF). At this point, the *Invoker* has the duty of invoking a serverless function for each partition of the input data against the compute backend. Considering the example from Listing 1, the *Invoker* performs three invocations of serverless functions against the compute

backend, i.e., one for each item from the `data` list defined at line 7. It is worth noting that Lithops provides a built-in data partitioner that splits a data set into several partitions, where each partition is passed as input to a serverless function. Next, the *Worker* runs inside a serverless function on the compute backend, and it fetches the (UDF) code and input data from the storage backend, then it executes the code, and finally, the output is persisted to the storage backend. The *Executor* monitors the storage backend for the output data, and when it becomes available, it transfers the data to the local machine.

Lithops provides an API that mimics the *concurrent.futures* [17] API from Python. The Lithops API contains three methods for running code in serverless functions: `call_async()`, `map()` and `map_reduce()`. Furthermore, it contains one method that keeps track of a serverless function's execution, i.e., `wait()`, and another method that downloads the final results of a serverless function from the storage backend, i.e., `get_result()`.

B. Shared State Abstractions

Lithops contains a Multiprocessing module that provides abstractions to enable concurrent serverless functions to share state with each other. The Lithops Multiprocessing API mimics the Python Multiprocessing API. The shared state abstractions from the Python Multiprocessing API [23] store their state values in local memory, whereas the shared state abstractions from the Lithops Multiprocessing API store their state values through disaggregated memory, in a Redis memory data store. The shared state abstractions of Lithops are categorized into three types: message-passing primitives (e.g., *Pipe*, *Queue*), shared memory primitives (e.g., *Value*, *Array*, *Manager*) and synchronization primitives (e.g., *Lock*, *Barrier*). The previously enumerated Lithops primitives behave similarly to the ones from the Python Multiprocessing API, with the difference that their values are stored in a Redis node by executing Redis commands (e.g., BLPOP, LPUSH).

IV. Serverless Architecture and Implementation

Lithops has been used for porting *k*-means and logistic regression to serverless. Already existing Python-based parallel implementations of *k*-means and logistic regression that relied on the *concurrent.futures* API [17] for enabling task parallelism and the Multiprocessing API [23] for sharing state, were modified to use Lithops by changing a few lines of code. The serverless architecture followed in our implementations is the one illustrated in Figure 1, whose workflow has been described in Section III. The implementation artifacts are open-source: https://github.com/CLOUDLAB-URV/lithops-distributed-ml

The serverless implementations have been built by following the Bulk Synchronous Parallel (BSP) synchronization protocol, which relies on synchronization barriers that wait for all workers at the end of an iteration before continuing to the next iteration. BSP ensures the correctness of the results, unlike other synchronization protocols such as Barrierless Asynchronous Parallel (BAP) which can develop incorrect solutions. However, the BSP protocol has the limitation that it brings synchronization overheads that slow down execution.

A client machine (e.g., laptop, VM) executes client code that launches several serverless functions (i.e., workers) that share mutable state through disaggregated memory. Each worker operates on a partition of the data set and at each iteration exchanges partial results with the other workers via the shared state through a Redis node. The workers perform parallel computations regarding shared data (e.g., centroids) and then the partial results obtained by all workers are aggregated to attain global results. At each iteration, the partial results computed by the workers are stored in the shared state, as one of the workers fetches the partial results of all other workers, and then aggregates them. Note that only one worker carries out this task, as there is no need in having multiple workers performing the aggregation. Furthermore, the global results are stored in the shared state, for the reason that they must be accessed by all workers at each iteration. In the case of the *k*-means algorithm, the partial results are represented by the cluster totals and cluster counters, whereas the global results are the centroids that are computed by dividing the cluster totals by the cluster counters. Moreover, in the case of the logistic regression algorithm, the partial results are represented by the partial gradients, whereas the global results are represented by the weights and the global gradients.

When using this architecture, the Redis node must be accessed as little as possible to minimize communication overheads. Therefore, the workers should operate over data stored in local memory as much as viable, rather than shared data, and then access and update the shared state only when necessary. Furthermore, bulk operations for accessing and updating the shared state should always be preferred, as they minimize the number of communication requests to the Redis node. At each iteration, the workers fetch the global shared state from the Redis node, then run a computation phase that yields partial results that are stored in the shared state before continuing to the next iteration. At the end of every iteration, one of the workers fetches the partial results from the global shared state, and then performs an aggregation of the partial results of all workers, and finally stores the aggregated results in the shared state. By convention, the worker who performs the described aggregation is always the first worker.

Two versions have been implemented for both algorithms, where each version uses a different storage design of the shared state. In the first version of the implemented algorithms, there is one single copy for each data object being stored globally in the shared state. Therefore, locks are used to prevent concurrent serverless functions from simultaneously accessing the same data object. Locks have the limitation of carrying overheads and slowing down the execution of the algorithms, therefore the second version aims to mitigate overheads by proposing a lock-free design. In contrast to the first version, the second version stores in the shared state one copy of each data object for every worker. Therefore, whenever a worker must update the shared state with his partial results, the worker will access the data objects that are associated with

Worker ID	Data Point ID	Data Point Value	Cluster ID
1	1	[0.5 ; 0.7]	2
	2	[5.4 ; 7.1]	3
2	3	[1.1 ; 3.9]	2
	4	[4.2 ; 2]	2

clusters_counters		
0	3	1

clusters_totals		
0	[5.8 ; 6.6]	[5.4 ; 7.1]

clusters_centers		
0	[1.93 ; 2.2]	[5.4 ; 7.1]

Fig. 2. Data Set (left) and Shared State of *Lock-Based* Version (right)

Worker_1		Worker_2	
clusters_counters	clusters_totals	clusters_counters	clusters_totals
[0 ; 0 ; 0]	[0 ; 0 ; 0]	[0 ; 0 ; 0]	[0 ; 0 ; 0]

Fig. 3. Initial Shared State of *Lock-Free* Version (2 Workers and 3 Clusters)

him. By following this approach, it can be said that every worker has his own memory space in the global shared state, therefore access conflicts are avoided when the workers update the shared state with their partial results.

A. Serverless K-Means Clustering

In this section, the two serverless implementation versions of the standard *k*-means clustering algorithm (Lloyd's algorithm) are detailed. Both versions follow the same logic, however, the first version requires locks for updating the shared state, whereas the second version proposes a lock-free design.

Figure 2 presents the principal shared state data objects of the lock-based implementation, where the table from the left is an example data set, and each table from the right represents a shared state variable. The bi-dimensional data set has three clusters, and is split into two partitions, one for each worker, as this example uses two workers. To compute the centroids, one would need the sum of all data points assigned to the cluster (`clusters_totals`), and the number of data points belonging to the cluster (`clusters_counters`). The shared data objects (`clusters_totals`, `clusters_counters` and `clusters_centers`) are stored as arrays, in which the index *i* is associated with the cluster *i*.

The workers iterate over their data set partition, and for each data point, they update the values of the arrays from the shared state, depending on which cluster the data point is assigned. To exemplify, the first worker assigns the first data point to the second cluster, therefore it increments by 1 the value of the second index of the `clusters_counters` array, and additionally, the first data point is summed to the value of the second index of the `clusters_totals` array. Three data points from the data set are assigned to the second cluster, therefore the value at index 2 of `clusters_counters` will become 3 by the end of the iteration. Furthermore, the value at index 2 of `clusters_totals` will be the sum of all data points assigned to the second cluster by the end of the iteration. More specifically, the data point [5.8; 6.6] represents the sum of all data points assigned to the second cluster, i.e., [0.5; 0.7], [1.1; 3.9], and [4.2; 2]. When both workers have finished assigning the data points to clusters, and have finished updating the shared state variables called `clusters_counters` and `clusters_totals`, one of the workers (i.e., the first worker) will compute the values of the centroids. The first worker will fetch the final values of the iteration of `clusters_counters` and `clusters_totals`, and then compute the centroids by dividing the totals by the counters. Finally, the first worker will update the shared state variable `clusters_centers` by the computed centroids. At the beginning of the next iteration, all workers will retrieve the current centroids by fetching from the shared state the value of `clusters_centers`.

In the lock-based implementation, locks are used for synchronization purposes, when the workers update the values of the arrays containing the cluster totals and cluster counters. The downside is that the lock objects bring additional overheads that slow down the execution of the algorithm. Considering the example from Figure 2, the two workers should not be able to write simultaneously at the same index of the arrays `clusters_totals` and `clusters_counters`. Therefore, locks have been used to synchronize access to a specific index of the two arrays. One lock has been used for every index (i.e., for every cluster), instead of using only one lock for the entire array, as an entire array should not be locked at a certain point by a worker, but rather only the access to the value of a specific cluster index should be locked by a worker.

In the lock-free implementation of the algorithm, the difference in the shared state is that the variables `clusters_totals` and `clusters_counters` are defined as a list of arrays, where each item of the list represents the memory space of one worker, as illustrated in Figure 3. For instance, `clusters_totals[0]` represents the `clusters_totals` stored in the memory space associated with the first worker. As every worker has his own memory space in the global shared state, no access conflicts can occur when the workers update the shared state with their partial results. Therefore, locks are not required considering that there is no need for synchronization. At the end of every iteration, one of the workers (i.e., the first worker) fetches all data objects of all workers from the shared state, and aggregates them, and then stores the aggregated results in the shared state before continuing to the next iteration. The limitation of the lock-free serverless implementation is that the aggregation phase lasts longer, considering that more data objects from the shared state must be aggregated, as there is one copy of each data object for every worker.

There is one variable in the shared state which represents the state of convergence of the algorithm. This variable was used to have only one of the workers (i.e., the first worker) checking for convergence, rather than all workers, to avoid overheads. The first worker will verify convergence by comparing the centroids of the last iteration with the centroids of the current iteration, and if they are equal, then the first worker will set the value of the convergence variable to 1. The advantage of this approach is that the other workers will not verify the convergence criteria themselves, but rather they will only verify the value of the convergence variable from the shared

Algorithm 1: Serverless *K*-Means

Input: k, X, $worker\_id$, max_iter
Output: $global\_clusters\_centers$, $labels$

```
1  if worker_id == 0 then
2  |  global_clusters_centers ← getRandomCentroids(X, k)
3  global_barrier.wait()
4  for iter ← 0; iter < max_iter; iter ← iter + 1 do
5  |  local_clusters_centers ← global_clusters_centers
6  |  local_clusters_counters, local_clusters_totals, labels ← computeClusters(X, local_clusters_centers)
7  |  global_clusters_counters ← global_clusters_counters + local_clusters_counters
8  |  global_clusters_totals ← global_clusters_totals + local_clusters_totals
9  |  global_barrier.wait()
10 |  if worker_id == 0 then
11 |  |  global_clusters_centers ← global_clusters_totals / global_clusters_counters
12 |  |  global_clusters_counters, global_clusters_totals ←0
13 |  |  if global_clusters_centers == local_clusters_centers then
14 |  |  |  global_converged ←1
15 |  global_barrier.wait()
16 |  if global_converged == 1 then
17 |  |  break
18 end
```

state. If all workers were to verify the convergence criteria themselves, they would have had to fetch once more all centroids from the shared state, instead of fetching a singular boolean value representing the state of convergence.

The output of the *k*-means algorithm is the centroids and the labels. The centroids are stored in the shared state, i.e., remotely in the Redis node, therefore the serverless functions do not need to output the centroids, as they can be accessed by the client machine directly via the shared state. However, the labels (i.e., the cluster index to which each data point is assigned) represent data that is local to the serverless functions, and it must be returned to the client machine that launched the serverless functions, to avoid losing it. Therefore, the output of each serverless function is the labels of its assigned partition of the data set. Each serverless function outputs a set of partial labels, which must be concatenated later on the client machine to obtain the complete list of labels for the entire data set. A limitation of the serverless implementations is that fetching the final global centroids, retrieving the partial labels outputted by the serverless functions and aggregating the partial labels, yield additional overheads that slow down execution.

Algorithm 1 contains the pseudocode of a generalized serverless implementation of the *k*-means algorithm. The algorithm takes as input the number of clusters k, X which represents the data set partition assigned to the worker, $worker\_id$ which represents the identifier of the worker, and max_iter which represents the maximum number of iterations. The output of the algorithm is the global centroids and the labels. The shared state is represented by all variables prefixed with "`global`", whereas the variables prefixed by "`local`" represent the local version of the variables from the shared state. Initially, the values of the cluster counters and totals from the shared state are initialized to 0. At lines 1–2, the centroids are randomly initialized by the first worker, whose $worker\_id$ is by convention 0, whereas the other workers wait for the centroids to be initialized at line 3. The iteration phase begins at line 4, and at line 5, the global centroids are fetched from the shared state and stored in a local variable. At line 6, the labels and cluster counters and totals are computed. At lines 7–8, the global shared state variables for counters and totals are aggregated with the partial values computed by each worker. At line 9, all workers synchronize, and by this point, the final values for the current iteration of the counters and totals have been obtained. At lines 10–11, the first worker computes the new centroids by dividing the final totals and counters from the shared state, and then updates the global centroids with the new centroids. At line 12, the first worker resets the values of the cluster counters and totals from the shared state to 0 for the next iteration. At lines 13–14, the first worker sets the value of the `converged` variable to 1 if convergence has been met. The workers must once again synchronize at line 15, to ensure that they are operating with the new computed centroids. Finally, at the end of every iteration, at lines 16–17, all workers check whether the algorithm has converged and break the execution of the algorithm if necessary.

B. Serverless Logistic Regression

The logistic regression algorithm based on (batch) gradient descent has been implemented with a serverless architecture in two versions. The rationale followed in the logistic regression serverless implementations is similar to the one described in the previous section for *k*-means, thus relevant details will be omitted for brevity. Note that the training phase (i.e., the fitting) of the algorithm has been implemented, and not the classifier's prediction procedure of a new given observation. The first version of the serverless implementation requires locks for updating the shared state, whereas the second version follows a lock-free design, similar to Figure 3.

The shared state is primarily represented by the gradients and the weights. In the lock-based implementation, both of these data objects are represented as an array, where the length of the array is dictated by the number of features of the training set. In the lock-free implementation of the algorithm, the difference in the shared state is that the gradients have been defined as a list of arrays, where each item of the list represents the memory space of one worker, similar to Figure 3. The synchronization of the workers is realized at each iteration via a barrier object. At every iteration, each worker increments the global gradients from the shared state with the partial gradients (i.e., sub-gradients) computed by the worker. Before the end of every iteration, one of the workers (i.e., the first worker) fetches the final global gradients aggregated from all workers and computes the new values of the weights by using the gradients and a learning rate. The weights represent the output of the logistic regression algorithm, however, the weights do not need to be returned by the serverless functions to the client machine that launched the functions. The weights are stored in the shared state, i.e., remotely in the Redis node, therefore they can be fetched by the client machine directly.

V. Evaluation and Validation

The Lithops-based serverless implementations were ported successfully to AWS Lambda and Google Cloud Functions, in order to validate that vendor lock-in is indeed avoided. Analogous, one could port the Lithops-based serverless implementations to other platforms of choice (e.g., Microsoft Azure Functions), ensuring the avoidance of vendor lock-in. Furthermore, the correctness of the serverless implementations has been validated successfully through validation scripts that compare the results obtained using the serverless implementations with the results obtained using standard implementations of *k*-means and logistic regression based on the *scikit-learn* ML library. For instance, the validation script of *k*-means validates that the centroids and labels obtained by the serverless implementation are identical to the centroids and labels obtained by the *k*-means implementation from the *scikit-learn* library. The validation of the *k*-means serverless implementation has been carried out successfully, by using data sets of various sizes, with various numbers of clusters, and various numbers of workers. Specifically, the validation has been carried out using data sets with sizes of [10, 50, 100, 500] MB, with a number of [3, 5, 7, 10] clusters, and with a number of [1, 2, 10, 20, 50] workers. Analogous, the same rationale has been followed to validate the correctness of the serverless implementation of logistic regression.

Several experiments have been carried out to evaluate the performance of the serverless implementations. Throughout the experiments, the serverless functions were executed over AWS Lambda, and the data sets were stored in Amazon S3. The experiments were performed over *k*-means, using randomly generated bi-dimensional data sets. All resources have been placed in the *eu-west2 (Europe–London)* region, including the EC2 VMs, the AWS Lambda functions having a timeout of 15 minutes, and the Amazon S3 buckets storing the data sets. For minimizing network latency, the AWS Lambda functions and the EC2 instances, i.e., the Redis node and the client machine, have been configured to run inside the same virtual private cloud (VPC).

A memory-optimized *r5.large* (2 vCPU, 16GB RAM) EC2 instance was used for hosting Redis. A general-purpose *t2.2xlarge* (8 vCPU, 32GB RAM) EC2 instance has been used as the client machine that launches the serverless functions. A client machine with 32GB of memory was used for the reason that the experiments involved the *k*-means algorithm, which outputs one label for every observation of the data set, therefore in the case of large data sets, a considerable amount of memory is required for being able to store in memory the labels outputted by the serverless functions. The experiments have initialized different executions of the algorithm with the same centroids to secure a fair evaluation. The execution times of the experiments have been measured in the context of warm starts of serverless functions, rather than cold starts.

A. Serverless Synchronization Overheads

An experiment has been carried out to measure the synchronization overheads incurred by barriers. The *k*-means algorithm has been executed with a data set of 2GB over serverless functions with 1769MB, which is equivalent to 1 full vCPU [24]. The average time spent waiting on a barrier has been measured with an increasing number of workers.

The chart from Figure 4 outlines the results of the experiment. The horizontal axis of the chart represents the number of workers used for executing the *k*-means algorithm, whereas the vertical axis represents the average time spent waiting on a barrier. Thus, when running 300 workers, a synchronization overhead of 2.72 seconds is incurred by each barrier. Considering that two barriers are used at each iteration for synchronizing the execution of the *k*-means algorithm, substantial overheads can result when executing the algorithm over a large number of iterations.

B. Serverless Peformance Compared to Sequential

This experiment aims to compare the performance of the serverless (lock-based) implementation of the *k*-means algorithm with its sequential (i.e., serial) implementation. The sequential implementation of *k*-means has been executed on a compute-optimized *c5.4xlarge* (16 vCPU, 32GB RAM) EC2 instance. Both implementations have used the same 8GB data set. Furthermore, for running the serverless implementation, each serverless function has been allocated with 1 full vCPU.

Figure 5 presents the speedup obtained by using a serverless implementation compared to a single-machine sequential implementation. The speedup is measured as the ratio of the execution time of the sequential implementation, to the execution time of the serverless implementation. Note that a speedup factor of n is called an n-fold speedup. The horizontal axis of the chart represents the number of workers used for executing the *k*-means algorithm, whereas the vertical axis represents the obtained speedup. The serverless implementation was launched with a number of concurrent serverless functions

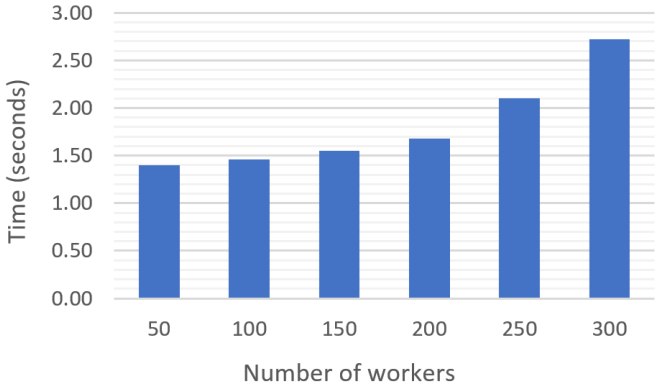

Fig. 4. Synchronization Overheads — Average Time Waiting on a Barrier

Fig. 6. Execution Time — Serverless Compared to Serverful

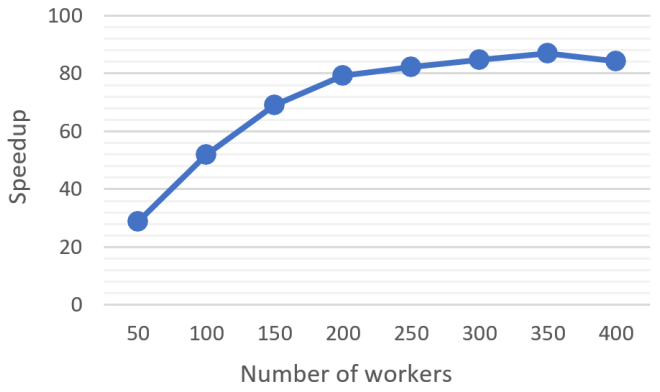

Fig. 5. Speedup — Serverless Compared to Sequential

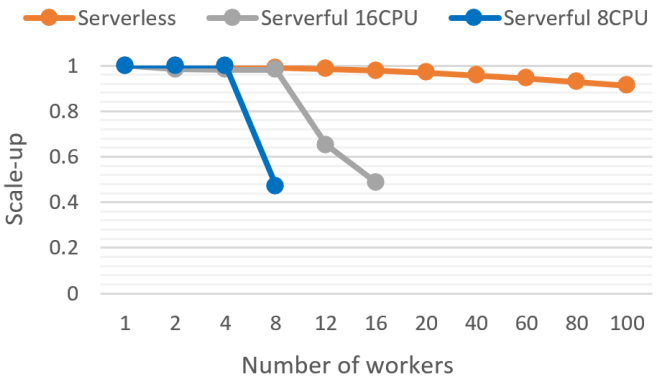

Fig. 7. Scalability — Serverless Compared to Serverful

that are multiple of 50, up to 400. The peak performance is achieved with 350 workers, obtaining a speedup factor of 87, i.e., an 87-fold speedup. The performance of the serverless implementation improves with the number of workers up to 350 workers, and then the performance begins to decline. The performance decline is to be expected, as the size of the data set is constant throughout the entire experiment.

C. Serverless Peformance Compared to Serverful

This experiment compares the performance of the serverless (lock-based) implementation of the *k*-means algorithm with its serverful (i.e., parallel and single-machine) implementation. Both implementations have used the same 3GB data set. Moreover, each serverless function has been allocated with 1 full vCPU. The serverful implementation is based on Lithops and contains the same code as the serverless implementation, but it uses the *localhost* deployment option of Lithops. Furthermore, it was executed on a compute-optimized *c5.4xlarge* (16 vCPU, 32GB RAM) EC2 instance. The parallelization of the serverful implementation is enabled via processes and a local (i.e., on the same EC2 instance) Redis instance has been set up for storing the shared state of the processes.

Figure 6 presents the execution time of the serverless implementation compared to the serverful implementation. The horizontal axis of the chart represents the number of workers (i.e., the number of concurrent serverless functions or processes) used for executing the *k*-means algorithm, whereas the vertical axis represents the execution time. The execution times have been compared when using [8, 12, 16] workers. The chart shows that the serverless implementation is slower when using 8 concurrent serverless functions than the serveful implementation when using 8 processes. Furthermore, the serverless implementation is slightly faster than the serverful implementation when using 12 workers, however, the difference is negligible and is likely caused by fluctuating network latencies. Moreover, the serverful implementation is slower than the serverless implementation when using 16 workers. When using all vCPUs of the EC2 instance, the performance of the serverful implementation degrades. It can be noted that the performance of the serverful implementation only slightly increases with the increase in the level of parallelism, however, the performance declines when all vCPUs are used. In contrast, the performance of the serverless implementation improves steadily with the increase in the level of parallelism.

A breakdown of the execution times has been carried out to determine the reason why the performance of the serverless implementation improves steadily with the increase in the number of workers. After analyzing the breakdown of execution times, it can be concluded that the performance of the serverless implementation improves steadily with the increase in the number of workers due to the parallelization of computation and of reading the data set from object storage

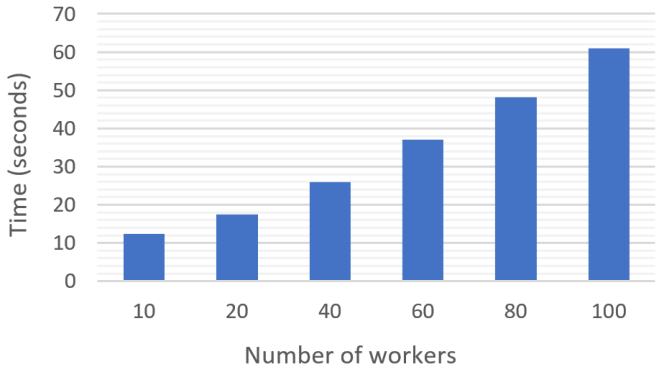

Fig. 8. Serverless Scalability — Synchronization Time

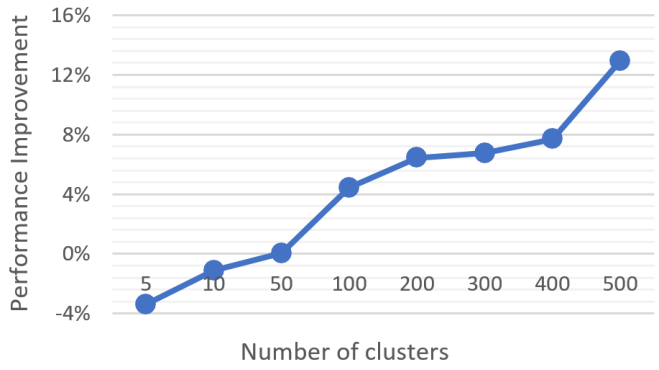

Fig. 9. Serverless Lock-Based and Lock-Free Performance Comparison by Number of Clusters

in parallel. Table I (columns #1 and #2) shows that the total time of reading the data set decreases with the increase in the number of workers. Furthermore, the duration of the computation phase of the *k*-means algorithm decreases significantly with the increase in the level of parallelism.

Similarly, a breakdown of the execution times has been carried out to determine the reason why the performance of the serverful implementation slightly increases with the increase in the number of processes but then declines when a number of processes equal to the number of vCPUs are used. Table I (columns #3 and #4) shows that the total time for reading the data set and for performing the computation phase slightly decreases when increasing the number of processes from 8 to 12. However, when using all vCPUs of the EC2 instance, the total time of reading the data set and carrying out the computation phase of the *k*-means algorithm slightly increases.

D. Serverless Scalability Compared to Serverful

An experiment has been carried out to compare the scalability of the serverless (lock-based) implementation of the *k*-means algorithm with its serverful implementation. The serverless implementation was executed over serverless functions with 256MB. The serverful implementation is based on Lithops, by using *localhost* deployment, and it has been executed on two compute-optimized EC2 instances, one having 8 vCPU (*c5.2xlarge*) and another having 16 vCPU (*c5.4xlarge*).

The size of the baseline data set is 50MB, i.e., this represents the size of the data set when executing the *k*-means algorithm with one worker. Data sets of sizes that are multiples of 50MB are used proportionally to the number of workers. For example, a data set of 100MB is used when running two workers, and a data set of 200MB is used when running four workers.

Figure 7 presents the scalability of the serverless implementation compared to the serverful implementation. The horizontal axis of the chart represents the number of workers (i.e., the number of serverless functions or processes) used for executing the *k*-means algorithm, whereas the vertical axis represents the scale-up. The scalability is measured in terms of $scale\text{-}up = T_1/T_n$, where T_1 is the execution time using one worker and a data set of 50MB, whereas T_n is the execution time using n workers and a data set of $n\times$50MB. Perfectly linear scalability would be denoted with $scale\text{-}up = 1$, i.e., the increase in the number of workers can handle the increase in the workload size. The chart shows that the scalability of the serverless implementation outperforms the serverful implementation. The scale-up of the serverless implementation degrades at a slower pace than the serverful implementation, as it can be noted that the scale-up of the serverful implementation quickly degrades. The serverless implementation achieves a scale-up factor of 0.96 with 40 workers, but it lowers down to 0.91 with 100 workers. The serverful implementation running on the EC2 instance with 8 vCPU achieves a scale-up factor of 0.99 with 4 workers, but it lowers down to 0.47 with 8 workers. Correspondingly, the serverful implementation running on the EC2 instance with 16 vCPU achieves a scale-up factor of 0.98 with 8 workers, but it lowers down to 0.65 with 12 workers, and to 0.48 with 16 workers.

After analyzing a breakdown of the execution times, it has been determined that the scalability of the serverless implementation slowly degrades due to synchronization overheads. Figure 8 shows that the total synchronization times of the serverless implementation rise with the increase in the number of workers. The synchronization times are caused by the total time spent waiting on barrier objects. One of the likely influencing factors for this is that the Redis node may not handle gracefully requests from multiple connection points, i.e., the concurrent serverless functions.

Similarly, after carrying out a breakdown of the execution times, it has been determined that the scalability of the serverful implementation degrades due to the duration of the computation phase of the algorithm. Table II presents the execution time of the computation phase when running the algorithm on the two EC2 instances. The results show that as the number of used vCPUs increases, the performance of the computation decreases.

E. Serverless Lock-Based and Lock-Free Performance

An experiment using serverless functions with 1GB of memory has been carried out to evaluate the performance of the two types of serverless implementations, in the context of the *k*-means algorithm. As a reminder, the first implementation

No. Workers	Read Dataset *Serverless* #1	Compute *Serverless* #2	Read Dataset *Serverful* #3	Compute *Serverful* #4
8	28.83	527.27	17.06	378.82
12	19.47	341.15	11.64	361.79
16	14.08	262.12	17.37	398.02

TABLE I
BREAKDOWN OF EXECUTION TIMES (SECONDS) — SERVERLESS AND SERVERFUL IMPLEMENTATIONS

No. Workers	Compute Phase c5.4xlarge (16 vCPU)	Compute Phase c5.2xlarge (8 vCPU)
4	51.22	49.13
8	51.72	49.59
12	70.76	109.26
16	106.80	-

TABLE II
SERVERFUL SCALABILITY — COMPUTATION TIME (SECONDS)

k clusters	Serverless Implementation	Update Shared State	Aggregate Shared State
100	Lock-Based	11.10	10.57
	Lock-Free	5.58	31.31
200	Lock-Based	14.19	8.20
	Lock-Free	11.04	62.96
300	Lock-Based	31.52	32.27
	Lock-Free	15.17	88.62
400	Lock-Based	45.35	44.15
	Lock-Free	22.17	122.13
500	Lock-Based	57.46	55.27
	Lock-Free	27.00	153.01

TABLE III
SERVERLESS LOCK-BASED AND LOCK-FREE — BREAKDOWN OF EXECUTION TIMES (SECONDS) BY NUMBER OF CLUSTERS

version is lock-based as it requires locks for updating the shared state, whereas the second version is lock-free.

The experiment compares the execution times of the iteration phase of the two serverless implementations, with an increasing number of clusters. The experiment aims to determine the effect of the number of clusters on execution times. Data sets of 100MB have been used in the experiment, where each data set contains data that can be clustered into a different number of k partitions. Note that in this experiment, 10 workers have been used in all executions. Figure 9 shows the performance improvement gained by using the lock-based rather than the lock-free serverless implementation. The performance improvement is measured as a percentage and has been calculated as $(100\% - lockBased\_duration \,/\, lockFree\_duration)$, where the $duration$ represents the execution time. The horizontal axis of the chart represents the number of clusters contained in the data sets, whereas the vertical axis represents the performance improvement. The chart shows that with a small number of clusters, the performance improvement has a negative value, whereas the performance improves steadily with the increase in the number of clusters. Therefore, the lock-free implementation is slightly faster than the lock-based implementation when using a number of clusters of ($k < 50$). However, the performance of the lock-free implementation declines with the increase in the number of clusters, to the point in which it is less performant than the lock-based implementation when using a number of clusters of ($k > 50$).

A breakdown of the execution times has been carried out to determine the reason for the performance degradation of the lock-free implementation with the increase in the number of clusters. Table III shows that the time spent aggregating the shared state in the lock-free implementation increases significantly with the increase in the number of clusters. The reason for this is that more data objects from the shared state must be aggregated, as in the shared state of the lock-free implementation there is one copy of each data object for every worker. However, the results show that the lock-free implementation does speed up the process of updating the shared state, due to the removal of locks. Nevertheless, the performance improvement for updating the shared state by removing the locks does not compensate for the performance degradation in the aggregation of the shared state. Hence, the performance of the lock-free implementation declines with the increase in the number of clusters.

VI. RELATED WORK

The work from [2] leverages CRUCIAL to implement ML algorithms, i.e., *k*-means clustering and logistic regression, with a serverless architecture. Similar to our work, the scalability of the serverless implementation of *k*-means has been compared against a serverful implementation, achieving a scale-up factor of 94% with 160 workers, and 90% with 320 workers. Moreover, the iteration's phase running time of the CRUCIAL-based implementation of *k*-means and logistic regression was compared against Apache Spark. The results show that logistic regression is 18% faster in CRUCIAL than in Apache Spark, whereas *k*-means is 40% faster in CRUCIAL than in Apache Spark with k = 25 clusters.

MLLESS is presented in [6], a FaaS-based ML training prototype that proposes optimizations tailored to the characteristics of serverless computing. First, it implements a significance filter for making indirect communication more effective by following a synchronization protocol that is a variant of Approximate Synchronous Parallel (ASP). Second, it implements a scale-in auto-tuner for reducing cost by leveraging the sub-second billing model of FaaS providers. Unlike our work which focuses on performance, the research from [6] focuses on cost-efficiency, as it has evidenced that running ML models on serverless can be more cost-efficient than running them as serverful, for models that converge quickly.

SMLT [5] is an automated serverless framework for scalable and adaptive ML design and training. The framework provides an overarching view of dynamic ML workflows for enabling adaptive and efficient serverless scaling and supports user-centric deployment goals such as training deadlines and budget limits for running ML workflows on serverless. SMLT has achieved 8× faster training speed and 3× lower monetary cost compared to similar state-of-the-art approaches.

Other works have demonstrated the benefits of using serverless architectures for the implementation of ML algorithms [3], [4], [7]. CIRRUS [3] is an ML framework that leverages serverless infrastructures to enable robust and efficient iterative ML training. CIRRUS has been evaluated with logistic regression against PyWren on AWS Lambda. SIREN is proposed in [4], an asynchronous distributed ML framework based on serverless architectures. A serverless implementation of logistic regression with Stochastic Gradient Descent (SGD) based on SIREN has been compared against its equivalent serverful implementation. The evaluation results have shown that the serverless implementation needed 22.9% less training time to reach the same loss. Furthermore, a comprehensive study has been carried out in [7] to outline the tradeoffs between training ML models on serverless and serverful architectures, which has been further extended in [8]. Additional research efforts have ported ML algorithms to serverless [9]–[11], [13]–[15], [25], however, they do not address cloud-agnostic serverless architectures that rely on existing parallel implementations of ML algorithms, as we have engineered with Lithops in our work. Furthermore, intra-function parallelism [26] is a technique that has demonstrated that the performance of ML algorithms running on serverless can be improved, by parallelizing the execution of serverless functions [12].

VII. CONCLUSIONS AND FUTURE WORK

This paper proposed a cloud-agnostic serverless architecture for distributed ML. Two stateful ML algorithms were ported to serveless via Lithops, *k*-means and logistic regression. In addition, two serverless implementations were realized for each algorithm, one requiring locks for updating the shared state, and another that is lock-free. The lock-based serverless implementation of *k*-means was superior in terms of performance and scalability compared to a serverful implementation when using a number of workers that is equal to or slightly lower than the total number of vCPUs available on the VM running the serverful implementation. Additionally, it achieved an 87-fold speedup compared to a sequential implementation. Moreover, the average time waiting on a synchronization barrier was 2.72 seconds when running *k*-means over 300 workers. Furthermore, it was shown that the performance of the lock-free serverless implementation of *k*-means declines with the increase in the number of clusters. We plan to evaluate the generality of our serverless architecture by porting other ML algorithms to AWS Lambda and to other serverless platforms. Additionally, we plan to carry out further experiments involving logistic regression and other ML algorithms, and use data sets with more features and of larger sizes.

REFERENCES

[1] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 41–54.

[2] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful Serverless Computing with CRUCIAL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–38, 2022.

[3] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A Serverless Framework for End-to-end ML Workflows," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 13–24.

[4] H. Wang, D. Niu, and B. Li, "Distributed Machine Learning with a Serverless Architecture," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1288–1296.

[5] A. Ali, S. Zawad, P. Aditya, I. E. Akkus, R. Chen, and F. Yan, "SMLT: A Serverless Framework for Scalable and Adaptive Machine Learning Design and Training," *arXiv preprint arXiv:2205.01853*, 2022.

[6] P. G. Sarroca and M. Sánchez-Artigas, "MLLess: Achieving cost efficiency in serverless machine learning training," *Journal of Parallel and Distributed Computing*, vol. 183, p. 104764, 2024.

[7] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards Demystifying Serverless Machine Learning Training," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 857–871.

[8] J. Jiang, S. Gan, B. Du, G. Alonso, A. Klimovic, A. Singla, W. Wu, S. Wang, and C. Zhang, "A systematic evaluation of machine learning on serverless infrastructure," *The VLDB Journal*, vol. 33, no. 2, pp. 425–449, 2024.

[9] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring Serverless Computing for Neural Network Training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 334–341.

[10] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving Deep Learning Models in a Serverless Platform," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 257–262.

[11] A. Aytekin and M. Johansson, "Exploiting Serverless Runtimes for Large-Scale Optimization," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 499–501.

[12] I. Predoaia and P. García-López, "Leveraging Intra-Function Parallelism in Serverless Machine Learning," in *Proceedings of the 9th International Workshop on Serverless Computing*, 2023, pp. 36–41.

[13] A. Deese, "Implementation of Unsupervised k-Means Clustering Algorithm Within Amazon Web Services Lambda," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 626–632.

[14] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "A Case for Serverless Machine Learning," in *Workshop on Systems for ML and Open Source Software at NeurIPS*, vol. 2018, 2018, pp. 2–8.

[15] T. P. Bac, M. N. Tran, and Y. Kim, "Serverless Computing Approach for Deploying Machine Learning Applications in Edge Layer," in *2022 International Conference on Information Networking (ICOIN)*. IEEE, 2022, pp. 396–401.

[16] J. Sampe, M. Sanchez-Artigas, G. Vernik, I. Yehekzel, and P. Garcia-Lopez, "Outsourcing Data Processing Jobs With Lithops," *IEEE Transactions on Cloud Computing*, 2021.

[17] Python, *Concurrent Futures API*, URL: https://docs.python.org/3/library/concurrent.futures.html (Last Accessed: 2024-08-25).

[18] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," *arXiv preprint arXiv:1812.03651*, 2018.

[19] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network Requirements for Resource Disaggregation." in *OSDI*, vol. 16, 2016, pp. 249–264.

[20] Y. Shan, *Distributing and Disaggregating Hardware Resources in Data Centers*. University of California, San Diego, 2022.

[21] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The Rise of Serverless Computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[22] AWS, *Amazon S3 Guide*, URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide (Last Accessed: 2024-08-25).

[23] Python, *Multiprocessing API*, URL: https://docs.python.org/3/library/multiprocessing.html (Last Accessed: 2024-08-25).

[24] AWS, *Configuring AWS Lambda functions*, URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-memory.html (Last Accessed: 2024-08-25).

[25] E. Paraskevoulakou and D. Kyriazis, "Leveraging the serverless paradigm for realizing machine learning pipelines across the edge-cloud continuum," in *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. IEEE, 2021, pp. 110–117.

[26] M. Kiener, M. Chadha, and M. Gerndt, "Towards Demystifying Intra-Function Parallelism in Serverless Computing," in *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, 2021, pp. 42–49.