SplitServe: Efficiently Splitting Complex Workloads Across FaaS and IaaS[*]

Aman Jain, Ata F. Baarzi, Nader Alfares, George Kesidis, Bhuvan Urgaonkar and Mahmut Kandemir
CSE Dept, School of EECS, Pennsylvania State University

Abstract

Serverless computing products such as AWS Lambdas and other "Cloud Functions" (CFs) can offer much lower startup latencies than Virtual Machine (VM) instances with lower minimum cost. This has made them an attractive candidate for autoscaling to handle unpredictable spikes in simple, mostly stateless workloads both from a performance and a cost point of view. For complex (stateful, I/O intensive) and latency-critical workloads, however, the efficacy of using CFs in combination with VMs has not been fully explored.

In this paper, we motivate a "split-service" application framework that can, for a given job (workload), *simultaneously* exploit both infrastructure-as-a-service (VM) and function-as-a-service (CF) products. Specifically, we design and implement a SplitServe-Spark embodiment of our proposal by modifying Apache Spark to use both Amazon VMs and Lambdas. Rather than letting performance degrade following the arrival of such jobs, we show that SplitServe-Spark is able to effectively use CFs to start servicing them immediately while new VMs are being launched, thus reducing the overall execution time. Further, when the new VMs do become available, SplitServe-Spark is able to move ongoing work from Lambdas to the new VMs, if that is deemed desirable from the cost or performance perspectives.

Our experimental evaluation of SplitServe-Spark using four different workloads (K-means clustering, PageRank, TPC-DS, and Pi) shows that SplitServe-Spark improves performance up to 55% for workloads with small to modest amount of shuffling and up to 31% in workloads with large amounts of shuffling, when compared to only VM based autoscaling. Furthermore, SplitServe-Spark along with novel segueing techniques can help us save up to 21% of cost by still giving almost 40% improvement in execution time.

1 Introduction

Many customers (tenants) are motivated to migrate to the public cloud because of the significant savings in infrastructure costs for their private clouds [28]. Having migrated, cloud tenants often realize that they can further reduce costs, subject to performance requirements, by more careful management of their workloads and of the resources they procure. In particular, they can take advantage of the diversity in available cloud services, including consideration of how resources are provisioned for them and how they are billed in concert with more intelligent characterization of the resource needs of their workloads.

Public cloud providers now offer a plethora of service types ranging from Infrastructure-as-a-Service (IaaS, most prominently virtual machine (VM) instances), Software-as-a-Service (SaaS), and the freshly trending *serverless* computing. In its form that is currently the most popular, called Function-as-a-Service (FaaS) or cloud functions (CFs), serverless computing allows a tenant to execute custom functions within lightweight containers on the cloud. In the rest of this paper, we use terms "FaaS," "serverless computing," and "CF" interchangeably. Commercially available examples of CFs include offerings from Amazon (AWS Lambda) [3], Google (Google Cloud Functions) [9], Microsoft (Azure Functions) [4], and IBM (IBM Cloud functions) [11].

Much work is emerging on using CFs for designing/redesigning cloud-bound applications. An important subset of this work leverages the superior startup latency of CFs ($\approx$100ms when warm) vs. VMs (a couple of minutes when cold) by using them to service excess workload during brief periods where workload demand exceeds provisioned VM capacity. For prolonged periods of this type, VMs are still preferred due to their lower costs but CFs help keep performance degradation under check during the VM launching period [19, 22, 32]. All of this work, however, is restricted to "simple" workloads by which we mean largely stateless requiring little or no shuffling of intermediate data. To address autoscaling for complex (stateful, I/O intensive and/or multi-staged with dependencies), latency-critical workloads subject to the limitations of CFs (particularly AWS Lambdas), we propose the SplitServe framework.

Approach and Contributions: We demonstrate the practical use of our proposed SplitServe approach by embodying it in Apache Spark, creating *SplitServe-Spark*. The

[*]This research was supported in part by NSF CNS 1717571 grant and an AWS cloud-credits gift.

three most important enhancements made to Spark can be summarized as follows:

- Spark Master can launch executors on both VMs and Lambdas and divide a *single* job's tasks across them.
- High-throughput storage layer for intermediate data shuffling using HDFS, which is suitable for both VM and Lambda based executors (note that the Spark master must be on a VM.)
- Terminating executors running on Lambdas to efficiently segue to VM-based executors without triggering extensive execution roll-back due to Spark's fault recovery.

We tested SplitServe-Spark using diverse benchmark workloads with different data shuffling intensities – Intel's HiBench [10] machine learning (K-means clustering) and web search (PageRank) workloads, DataBrick's Spark-SQL-Perf [25] TPC-DS benchmark, computation of Pi – and with a wide range of input sizes and degree of parallelism for each. Our evaluations show that SplitServe-Spark is suitable for fast autoscaling using AWS Lambdas. Specifically, by jointly using VMs and Lambdas to execute complex workloads, we show around 55% performance improvement for workloads with small to modest amount of shuffling and up to 31% improvement in workloads with large amounts of shuffling when compared to only VM based autoscaling. Furthermore, SplitServe-Spark along with novel segueing techniques can help save up to 21% of cost by still giving almost 40% improvement in execution time.

Outline: The rest of this paper is organized as follows. In Section 2, we motivate our problem and offer background. Section 3 describes our implementation of SplitServe-Spark using AWS Lambdas. We give our experimental setup and present the collected results on AWS in Section 4. Related work is discussed in Section 5, and the paper is concluded in Section 6 with a brief discussion of the planned future work.

2 Background and Motivation

In this background section, we compare CFs with VMs using examples from AWS. Similar trade-offs apply to products from other cloud providers as well. Following this, we describe what sets our proposed SplitServe system apart from the state of the art. Related work is discussed in greater detail later in Section 5 below.

2.1 Why Combine VMs and Lambdas?

From a tenant's perspective, two advantages of AWS Lambdas over VMs are of significance to our work as well as to recent related work (Section 5). First, Lambdas offer significantly lower startup latency than VMs. Existing studies [29, 19, 32, 26, 27] and our own measurements show that an AWS VM may take up to 2 minutes or more after the user requests it to start working. While a "cold-start" Lambda (*i.e.,* requiring a fresh VM bootup and container launching) incurs a similar startup latency, a "warm-start" Lambda (*i.e.,* an already existing recently-used container with relevant code/data likely to be in memory) incurs a much smaller startup latency of only about 100ms [18].[1]

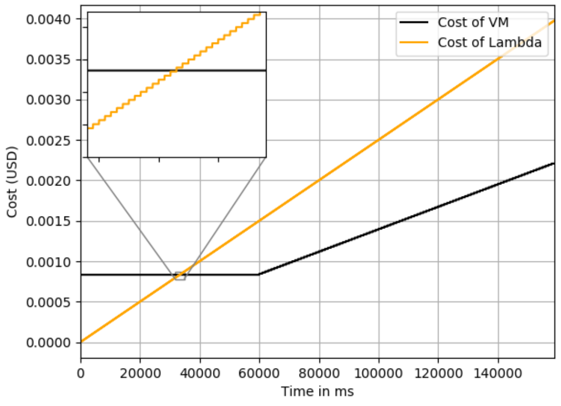


Figure 1: Comparing the cost of one vCPU on a m4.large instance vs an AWS Lambda function with 1536 MB memory which gives an effective capacity of one vCPU. The x-axis represents time in milliseconds while the y-axis is Cost

Second, Lambdas are cheaper for short-lived resource usage patterns through sub-second minimum cost and billing granularity.[2] Specifically, AWS Lambdas are priced based on memory allocated (ranging from 128MB-3GB of memory per instance with one vCPU per 1.5GB) and execution time. More specifically, their cost is given by the product of memory allocated and time-in-use rounded up to the nearest 100ms (there are additional costs related to number of invocations). On the other hand, VM prices are typically based on their type (*e.g.,* reserved, on-demand, burstable, spot) as well as resource capacities (CPU, memory, storage). In particular, AWS VM cost is given by their price times the time-in-use rounded up to the nearest second. Figure 1 compares the cost of one vCPU on a m4.large instance to an AWS Lambda function with 1.5GB memory which gives it an effective capacity of one vCPU. Amazon EC2 instances impose a minimum 1-minute charge on the users following which the cost is calculated in 1 second increments. In the figure, we can see this as a straight line till 60 seconds, and a step-wise monotonically increasing pattern following that. In comparison, AWS Lambda bills the

[1] AWS keeps Lambda containers alive for ∼90 minutes.

[2] There are other cost advantages of Lambdas stemming from the fact that they relieve the user of managing OS and runtime environment. These cost aspects are harder to quantify in the context of our work and are beyond the scope of this paper.

tenants in 100 milliseconds increments with a minimum charge of 100 milliseconds, presenting itself as a continuous step function as soon as the billing cycle starts. The graph shows how quickly a Lambda can overshoot a VM in terms of cost of execution. Cost curves like this can help the tenants choose the duration for which they want their workloads to run on Lambdas while working under strict budgets.

Given the benefits, it is natural for a cost conscious tenant to exploit them as follows: instead of overprovisioning VMs to deal with workload mispredictions, the tenant can provision closer to predicted requests and use the agile Lambdas to reactively handle unexpected spikes in workload. Indeed this is the basic idea behind recent works such as [19, 32, 22] which deal with simple workloads.

At the same time, in their current form, Lambdas have a number of restrictions that pose hurdles for complex workloads:

- *Limited resource capacity:* Lambda containers do not possess enough memory for the needs of many workloads. We have found that due to their relatively modest memory allocation, garbage collection may begin posing significant overheads after only a few minutes execution of a Lambda, even for moderately memory-intensive workloads. Finally, each instance is provided a relatively small local storage (/tmp directory) of size 512MB to store intermediate state [17].
- *Limited lifetime and user control:* Lambdas are terminated after 15 minutes, practically rendering them unsuitable for longer-running jobs [17]. Also, a user cannot control the order in which interacting Lambdas are actually started by the provider; this order may be different from the user's invocation order.
- *Poor support for sharing of intermediate state:* Since Lambdas do not expose an IP address, there is no direct channel for an entity (whether VM or Lambda) to send data over the network to a Lambda after its initial invocation. This in combination with (i) their limited lifetime and (ii) the user's lack of control over when Lambdas are initiated by the provider, imply that state transfer between Lambdas must rely on a storage facility external to the source Lambda's container. Existing solutions (see Section 5) using either: (a) Amazon's S3 or SQS [14], which are slow.[3]; or (b) in-memory datastores such Redis [20], which are relatively expensive.
- *Steeper cost curve for longer-lasting work:* Currently, Lambdas charge a higher price per unit resource as compared to VMs which results in higher costs for long-lasting resource needs, as discussed above (see Figure 1).

[3][15] attributes their slowness to their fault tolerance mechanisms that it deems excessive for the relatively short-lived intermediate state transferred.

The key contribution of this paper is to demonstrate that combining Lambdas and VMs can indeed provide cost/performance benefits even for complex workloads. Realizing these benefits, however, requires careful design around the idiosyncrasies of existing Lambda offerings. Among the key challenges that need to be addressed compared to existing work on simple workloads are: (i) simultaneously using VMs and Lambdas even within a single job taking into account significant differences in cost and performance obtainable from them, (ii) segueing work related to stateful tasks from Lambdas to newly available VMs, when Lambdas start to become more costly (recall Figure 1), and (iii) addressing resource and lifetime constraints that are more problematic for complex workloads.

2.2 SplitServe: Context and Our Focus

Suppose the tenant is expecting a stream of latency critical, repetitive work (jobs of the same type) over a forthcoming time-window (that could be on the order of an hour or days). Specifically, say $m(t)$ is the mean and $\sigma^2(t)$ is the variance of the workload, measured in terms of the size of its input dataset, of the job arriving time t, and assume that the required resources to execute this job by a given time, as discussed above, are accurately characterized. Reserve instances can be procured for the portion of the workload which is certain to come, *e.g.,* corresponding to $m(t) - 2\sigma(t)$ when this quantity is positive, *i.e.,* with over 95% confidence the reserve instances will be fully utilized. A very conservative tenant may, *e.g.,* schedule on-demand instances corresponding to $3\sigma(t)$, *i.e.,* so that the total VM instances scheduled, including reserve, is $m(t) + 2\sigma(t)$ and, with over 95% confidence, the incident work will be less than the IaaS capacity. But a less conservative tenant may try to save money by allocating on-demand instances corresponding to much less than $3\sigma(t)$. In this case, one can expect that, frequently and at random, there may not be enough resources in existing VMs to execute the incident latency-critical job by the required time, *i.e.,* auto-scaling may be warranted, at least temporarily.

On the other hand, batch work can be performed using resources procured for latency-critical work but left idle. If substantial VM resources are idling, the *cost manager module* of the tenant customer may decide to terminate them. However, recall that it takes about two minutes to spin-up VMs (during which time the tenant may be charged).

To autoscale latency-critical work, the following options differ in both cost and performance:

- Wait for previously provisioned IaaS resources (executors in existing VMs) to become available (assuming such availability is predictable and associated waiting times are tolerable);
- Preempt existing work that is not latency critical (as-

suming such batch work is running - again note that IaaS may be allocated *solely* for latency-critical work – batch work may only access resources when they are idling and should be running on an application framework that will avoid substantial rollback overhead when preempted, *e.g.,* [30]);
- Spin-up new VMs (recall cold-start delays until VMs can execute work may on the order of 2 minutes, during which time they may incur cost);
- Quickly (warm start) spin-up cloud functions (FaaS), possibly jointly with some executors in the existing VMs as in our proposed SplitServe framework[4]; or
- Exploit new cloud functions and then segue to new VMs, *i.e.,* the cloud functions bridge to executors on VMs. There are two cases: new executors become available on existing VMs, or new VMs are launched when the Lambda functions are launched. Either way, Lambda functions are terminated (carefully, so as not to induce execution roll-back, *cf.* next section) when new executors become available. If new executors are required for more than a couple of minutes, it may be more cost-effective to segue from Lambdas to VMs[5].

A tenant cost manager module may need to weigh performance and cost options in order decide on one of the foregoing options for a given latency-critical workload. For example, if prior workload characterization has determined that r additional single-core executors are sufficient to meet the execution time requirements of $t \leq$ two minutes, then the cost $C_\lambda(r,t)$ of using AWS Lambda functions would be given by

$$C_\lambda(r,t) = \qquad (1)$$
$$(0.00001667 \times m(0.1\lceil 10(t-0.1)\rceil) + 2 \times 10^{-7})r,$$

where $.1\lceil 10(t-.1)\rceil$ rounds t up to the nearest 100ms and there is a 100ms $=$ 0.1s minimum charge. If the requirements were for, say, t $=$ 360s ($=$ 6 minutes), and garbage-collection issues will not pose a problem for Lambda functions executing this long for the workload under consideration, then the option of bridging to newly spun up VMs should be considered. That is, assuming VMs become available at 2 minutes, $C_\lambda(r,360)$ should be compared with $C_\lambda(r,120) + C_{\text{VM}}(r,360)$, where again we note that the IaaS costs C_{VM} may accrue even though the VMs are not available to work for two minutes, otherwise the term would be $C_{\text{VM}}(r,360-120)$. If $r = 4k$ for some integer k and each executor is realized as a single-core container on an AWS m4.xlarge instance (4 vCPUs, 16 GB Memory), the cost in dollars is $C_{\text{VM}}(r,t) = (.00333 + 0.0000556\lceil (t-60)^+)(r/4)\rceil$, i.e., \$0.2/hour with a one-minute minimum charge and time is rounded-up to the nearest second. Note that Lambda-function executors have 1.5GB (<4 GB) per core (this small amount of memory per core may cause garbage collection problems for long-running Lambda-function executors of memory-intensive workloads).

The costs associated with the other autoscaling options can be similarly characterized – such detailed cost management issues are not the focus of this paper.

3 SplitServe-Spark

We have implemented SplitServe-Spark by modifying Apache Spark version 2.1.0 (rc5) and porting certain features from Qubole's Spark-on-Lambda [21], which is a rendition of Spark that uses AWS Lambdas as executors. In this section, we describe the most important modifications we made to implement SplitServe-Spark, which we will be made open-source upon publication.

3.1 Background on Apache Spark

Apache Spark [24] is an open-source distributed general-purpose cluster computing engine for large-scale data processing, and is widely used. A Spark cluster has a single master and one or more slaves (worker nodes). A Spark driver is the entry point of a Spark application – it runs the main function of the application and is the place where the "Spark context" is created. A job is a unit of work that a user is interested in. Upon submission, Spark breaks down a job into a series of stages comprising tasks (*e.g.,* map, reduce). This breakdown is represented as a Directed Acyclic Graph (DAG) which is in a sense an "action plan" for the execution of the job. Note that a stage may have multiple parallel tasks. The Spark driver contains various components - DAGScheduler, TaskScheduler, BackendScheduler and BlockManager – that are collectively responsible for the translation of a given Spark user code into jobs. The driver schedules job execution and negotiates with the cluster manager (*e.g.,* Mesos, Yarn, Kubernetes or Spark's Standalone cluster manager) for resources in terms of "executors." An executor, launched as a JVM process, is a distributed agent responsible for the execution of tasks. Every Spark application has its own executor process(es). Spark offers two modes of executor allocation: static allocation, wherein all executors run for the entire lifetime of the application, and dynamic allocation, that lets an application to start with a predefined minimum number of executors, which can grow in numbers (up to a specified maximum) as and when the resources becomes available in the cluster. If however, an executor is idle for some time, it is killed and the resources are returned to the cluster.

Spark creates stages at state transfer boundaries, *i.e.,* when the execution of the next stage depends on the

[4]Cold start AWS Lambdas can also take ∼2 minutes to spin-up and become available; so, it may be more cost-effective in some cases to simply spin-up new VMs if warm-start Lambdas are not available. [29] suggests sending short mock work on Lambda functions periodically to keep them warm.

[5]Note that AWS Lambda-at-edge functions, *i.e.,* proximal to VMs, are about 3× more costly than "regular" Lambda functions, which are, in turn, more expensive per unit time than comparably provisioned VMs.

output of the tasks from the previous stage. This movement of data between nodes at stage boundaries is known as *shuffling*. In static allocation of executors, the shuffle data can be kept in memory of the executor which can act as a server for other executors who want to access this data. However, in the case of dynamic allocation, since an executor might be killed due to inactivity, to prevent the intermediate shuffle data from being lost, all of the intermediate shuffle output is written to the local disk. Note that, for large applications, this can be a bottleneck in terms of overall performance.

3.2 Master, Scheduler, and Allocation Manager

SplitServe-Spark can run an incoming job's tasks on i) only VM executors, or ii) only Lambda executors, or iii) a combination of both VM and Lambda executors by *splitting* the tasks across them. Essentially, SplitServe-Spark can distinguish between VM and Lambda based executors. This information is provided by the executor, based on the type of cloud offering it is launched upon (VM or Lambda) within the cluster. This is achieved by adding the ability of launching both VM and Lambda executors to *StandaloneSchedulerBackend* and modifying the appropriate data structures to account for the two types along with the timestamp that records the start of an executor process (this information will used in Section 3.4). *CoarseGrainedSchedulerBackend* uses this information to keep track of all the executors, their availability, and to perform their graceful decommissioning.

On the executor side, we modified the *CoarseGrainedExecutorBackend* to propagate this extra information to Spark whenever an executor is registered with the cluster. This also required modifying *SparkEnv* to check the executor type and create an appropriate environment for the executors, along with modification of all the relevant function calls in the workflow throughout the lifetime of the application to accept the extra information and make decisions based on the type of the executor it is working with. As a result, a combination of these modifications are used in *ExecutorAllocationManager*. We use Spark's dynamic allocation of executors policy, which means that executors are requested only if the allocation manager sees pending tasks in the scheduler queue, and these executors are killed if they have been idle for a pre-defined timeout value. Hence, when DAGScheduler submits a new stage and TaskScheduler adds the corresponding tasks of this stage to the scheduler queue, the allocation manager calculates the number of executors required to handle the backlog of these pending tasks and requests those many executors from the cluster manager (in this case, from SplitServe-Spark).[6] *ExecutorAllocationManager* then decides (*cf.* Section 3.4) to request either VM or Lambda executors through *CoarseGrainedSchedulerBackend*.

[6] In practice, such requests may need to be approved by a cost manager module.

3.3 External Shuffler

Complex workloads generally tend to engage in significant data transfers across program stages, *i.e.,* the shuffle operation. Taking the case of MapReduce workloads, for M mappers and R reducers, the number of intermediate shuffle files generated is proportional to $M*R$. Recall that to prevent the loss of this shuffle data, the files containing it are written to the local storage by the VM-based executors in Spark. This becomes a problem for Lambda-based executors where (a) users have to pay a steeper price per unit resource than VMs to preserve the shuffle data, and (b) availability of only a small amount (512 MB) of local storage per Lambda. A natural solution to overcome this problem is to use some ephemeral storage solution like AWS S3 (sometimes in conjunction with high speed caches like Redis [23]) or AWS SQS [2]. However, these solutions tend to be either too slow (for various reasons, *e.g.,* throttling) or too expensive or both. In any case, in their current form, these solutions are not well suited for the kind of operations of interest herein, *i.e.,* a large number of small writes. Consequently, we wanted to use a single common high throughput storage layer, which can be accessed by both VM and Lambda based executors. We found an *HDFS-based storage layer* to offer a better cost/performance trade-off for external shuffling than existing solutions.

In HDFS-based external shuffling, we expose the file system in the local storage of the (always VM based) Master node through a *hdfs://* REST API. We added two new configuration options in Spark's default configuration file: *spark.shuffle.hdfs.enabled*, which, when set to *true*, indicates to the executors that all the intermediate shuffle files are to be written to an HDFS node; and *spark.shuffle.hdfs.node=hdfs://XX:XX*, the address of the HDFS node where the data is to be written to and read from. We used Hadoop-2.7.7 in our SplitServe-Spark prototype to keep it compatible with the Spark version. Following this, additional modifications were made to Spark to get a consistent directory layout which is followed by *both* types of executors.

In Spark, executors originate from worker nodes which are available as infrastructure to the Spark cluster. Note however that, in case of Lambda executors, since there are no machines (physical or virtual) to deal with, there is no notion of worker nodes. Executors directly join the cluster as processes available to execute works from a job. Hence, both of these executors inherently acquire different properties, some of which have to be consistent to allow them to work in tandem. For example, since both types of executors have their own way of getting assigned an executor ID, it was important to keep this assignment consistent. This allows the executors to write in separate directories on HDFS by using their executor ID as an entry point, and allows the user to perform a fine-grained analysis on the work distribution between the executors.

In summary, the choice of using an HDFS-based external shuffle service was based on the following factors:

- Spark already comes with library support for HDFS writes.
- There is at least one VM always available (for the Spark Master), even in pure Lambda-based executor solutions, which can be used to deploy HDFS without any supplemental provisioning.
- HDFS gives performance close to local writes and reads and a better performance for Lambdas (than currently available ephemeral storage solutions) since there is no throttling.
- HDFS does replication on multiple data nodes which scales better than NFS for better read performance by mitigating network bandwidth bottlenecks.
- HDFS nodes can span VMs separate from where the master node resides, *e.g.,* a storage-optimized EC2 instances for better performance. Or, HDFS can simply be run on the master VM (in case a single node suffices), lowering costs compared to solutions that employ in-memory stores like Redis.

3.4 Segueing from Lambdas to VMs

One of the primary concerns when working with Lambdas is the steep price the users have to pay as the total execution time of the function increases. Referring to Figure 1, we can see how Lambdas can quickly overshoot a VM in terms of cost, which means that it will *not* be cost effective to keep running Lambdas for longer than a certain time-threshold that can be determined by workload characterization and using such cost curves. Note that, a lot of factors go in trying to come up with such thresholds. However, since the workloads are largely repetitive, the data required to decide these thresholds can potentially be learned either online or offline or both.

Another important issue which has been discussed extensively in recent works is the memory limitation on Lambdas. For Spark, where each executor is a JVM, the issue of *garbage collection* becomes very important.[7] In particular, smaller memory on these Lambdas results in more frequent invocations of the garbage collector, which in turn hurts workload performance. This may make it difficult, or in some cases impossible, to run a workload processing large datasets on a small number of Lambdas. Recall that AWS Lambdas have a fixed CPU to memory ratio and additional memory based resources for a Lambda-based executor (one per Lambda function) could be quite costly. This is in contrast to VMs running plural executors, where there is generally more memory available per core, and one can simply allocate even more memory per core by employing fewer executors than cores.

These observations motivate the need for inhibiting running Lambdas for longer periods. In other words, if we observe (or predict) that an ongoing job is going to run longer than a threshold (on the order of two minutes for AWS Lambdas), we need to segue the remaining work to executors on newly spun up VMs or those on existing VMs which were busy when the job under consideration was launched but have now become available. However, using Lambdas as a ”bridging tool” requires careful consideration. If a Lambda is killed with work still remaining in its queue, Spark treats the event as a ”resource failure”. Though Spark is designed to handle such failures with its recovery-from-fault mechanisms, the system enters into an ”execution roll-back”, which typically involves a very high cost due to cascading re-computations. This is similar to working with transient resources as discussed in [30], and usually greatly increases job execution times and creates a large amount of traffic on the network channels. Note that such segue mechanisms are not required if the job itself is small enough to finish within the threshold in which case the Lambda executors are simply terminated. So, the Lambda executors can act as an internal temporary autoscaler, which allows SplitServe-Spark to dynamically scale the working set of resources at a per-job granularity.

Considering the frequency of Lambda invocations, it is easy to determine how well provisioned our cluster is for a given job arrival pattern. For example, frequent Lambda invocations may indicate an increased burstiness in the job arrival pattern for a cluster that is not conservatively IaaS provisioned (with VM-based executors). Additional discussion of autoscaling can be found in [6], where they advocate for an external shuffle service that SplitServe-Spark achieves by exploiting HDFS.

When dynamically allocating executors, Spark gives an option to downsize the number of live executors in the cluster if they have been idle for some predefined time. In this context, an idle executor means that no task is currently running on it. We leverage this mechanism in SplitServe-Spark. The overarching idea is to make a Lambda executor idle so that *ExecutorAllocationManager* can safely remove it without treating it as a failure. We added a configuration *spark.lambda.executor.timeout*, which serves as a ”threshold” for the Lambda executor as its total running time. We modified *CoarseGrainedSchedulerBackend* to record the time when an executor starts serving the job upon its registration. Every time *CoarseGrainedSchedulerBackend* makes resource offers for the executors to pull tasks from the scheduler queue, it first filters out all the Lambda executors which have exceeded their allowed execution time from the set of ex-

[7]Java garbage collection is the process by which Java runtime system performs automatic memory space recycling.

ecutors which are viable to run pending tasks. Thereafter, filtered Lambda-based executors are responsible for completing only the pending tasks that have already been scheduled on them, and no additional tasks are assigned to them. As soon as these Lambda-based executors finish executing their pending tasks, *ExecutorAllocationManager* starts the idle clock for them, following which they can be gracefully terminated. Note that, *spark.lambda.executor.timeout* is a configurable "knob", which can be set based on actors like budget, classification of big or small jobs, ability of autoscaler to spin-up new VMs, or an expectation that existing VM executors will soon become available.

4 Experimental Evaluation

In this section, we first describe our experimental setup including offline[8] workload profiling. We then give a comparative performance evaluation of SplitServe-Spark using diverse workloads from two widely-used benchmark suites: Intel HiBench [10] and Spark-SQL-Perf [25]. Intel HiBench is a big-data benchmark suite that lets one evaluate big data frameworks in terms of speed, throughput, and system resource utilizations. From Intel HiBench, we present results for distributed K-Means, a compute intensive machine learning based workload with some shuffle I/O, and WebSearch (or PageRank), which is compute and shuffle I/O intensive. In addition, from DataBrick's Spark-SQL-Perf, we present results for TPC's Decision Support (TPC-DS) ETL-type queries which vary in their compute and shuffle I/O requirements. Finally, we also give results for Pi, a compute heavy workload with little shuffle I/O.

4.1 Experimental Setup

We now illustrate basic workload profiling for PageRank; similar profiling was done for the other workloads described and evaluated below. The overall job execution time and cost incurred to run the job are assessed while varying the degree of parallelism by changing the number of executors in the cluster. Each executor is assigned a vCPU and each Lambda function is a single executor assigned 1 vCPU; so, the term "executor" is synonymous with one core (but, in AWS, all Lambda functions are assigned 1.5 GB memory per core and there is typically more memory available for a VM based executor). As the number of executors is increased, the (equal) task size per executor decreases proportionately.

The following experiments are conducted on three input dataset sizes: "large" (100,000 pages), "medium" (50,000 pages), and "small" (25,000) pages (much larger input datasets are considered in the next subsection). The PageRank algorithm ran for 3 iterations. The first set of experiments was run on SplitServe-Spark with the master/driver running on a m4.xlarge AWS VM[9] and the executors in this case were *all* Lambda functions. As can be seen from Figure 2a, the profiling presents a classic "U" curve expected for the execution time of a parallel workload [13]. The curve suggests that, for a fixed input dataset size, there exists a "degree of parallelism", which results in the optimal performance (increasing the degree of parallelism further results in poorer performance owing to the communication overheads). Additionally, the total cost is plotted (depending on both the number of executors and the total execution time). In case of a (relatively) "large" PageRank job (here 10^5 pages), if the execution time needs to be less than 70s, then two executors would be the lowest-cost choice; however, if the execution time needs to be less than 60s, then the only choice is 4 executors.

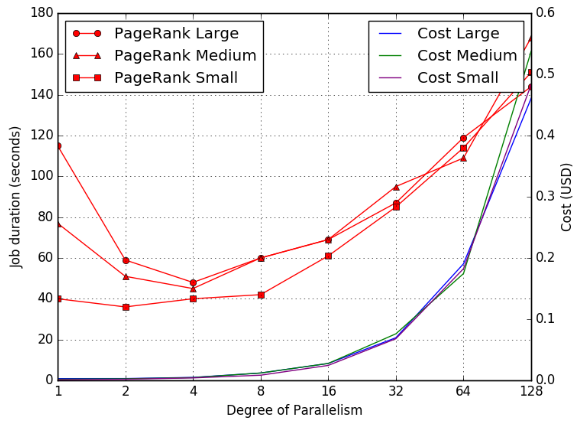
A second set of experiments was performed using "vanilla" Spark with VM-based executors (worker nodes are EC2 VM instances) with the master/driver running on single m4.xlarge machine. For each degree of parallelism, we used the fewest number of instances (to minimize inter-VM communication overhead): m4.large for 1 and 2 vCPUs (executors), m4.xlarge (4 vCPUs), m4.2xlarge (8 vCPUs), m4.4xlarge (16 vCPUs), m4.8xlarge (32 vCPUs), m4.16xlarge (64 vCPUs) and two m4.16xlarge (128 vCPUS). Note that, generally, the cost per core will depend on the VM size – larger VMs mean more cost per core but also less execution time because of the substantially less inter-VM communication, *e.g.,* for data shuffling – see the cost curves of Figure 2b. Also, note from Figure 2b that, even though the optimal degree of parallelism is the same as that when running only with Lambda-based executors, the overall execution time for the job is much lower when running on VMs (as expected).

A similar "U" shaped curve can be obtained for the other workloads as well, for both VMs and Lambdas; some of them may be different for more memory intensive workloads, particularly those that exceed the resources of a cloud function but not those of an executor on a VM.

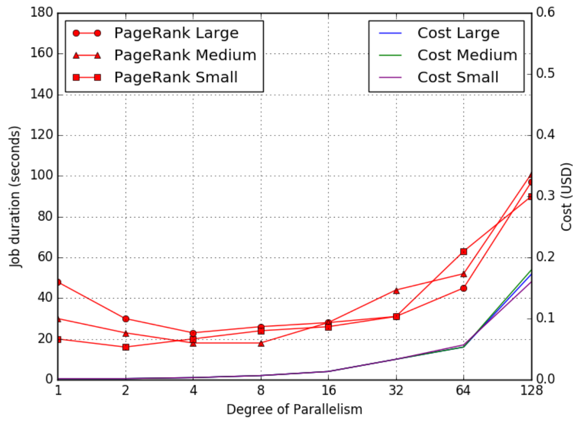
In the following experiments, two of the "baselines" could represent prior workload characterization, as in Figures 2a and 2b, *i.e.,* using only AWS Lambdas or only executors in VMs. Additionally, we present baseline results for Qubole over Lambda and autoscaling using only VMs, where the autoscaling scenario involves some executors on "existing" VMs (*i.e.,* executors available when the job is launched). These are compared against autoscaling i) using only VMs, ii) using only Lambdas, and iii) using Lambdas to segue/bridge to VMs, where the last one requires terminating the Lambda functions without trigger-

[8]In practice, offline profiling can be continually reinforced by online learning.

[9]All of the application frameworks considered herein, even those focusing on the use of cloud functions for executors, employ a large VM for the their master/driver.



(a) Varying the degree of parallelism for PageRank running on SplitServe-Spark over (only) AWS Lambda



(b) Varying the degree of parallelism for PageRank running on Vanilla Spark over (only) AWS EC2 instances

Figure 2: Performance/Cost vs Degree of Parallelism

ing an execution rollback by Spark.

4.2 Our Findings

We evaluate the performance of SplitServe-Spark as well our baselines under the various cluster-state scenarios which a latency-critical job might encounter upon arrival. Such scenarios and different corresponding decisions are as follows (recall Section 2.2):

- Job arrives to find the required resources R are available from the cluster's VMs and can be readily used to launch its tasks.
- Job arrives to find only a portion $r < R$ of the required resources are available for its use:
  - The job is launched (only) on this partial set of resources.
  - The job starts executing on this partial set of resources, while the cluster autoscales by procuring $\Delta = R - r$ more VM resources (which will be used by the job when new executors are ready).
  - The job is launched on R resources available from r existing VMs and Δ (warm) Lambda functions. If the expected total execution time of the job is sufficiently long, the cluster also procures Δ resources on VMs and, as soon as these new VM based executors are available, the job segues from Lambdas to them to complete any remaining tasks.

4.2.1 K-means Clustering

K-means clustering is a type of unsupervised learning for unlabeled data points. The goal of this workload is to group the data points into one of $k > 1$ clusters. The algorithm works iteratively to assign each data point to one of the k groups based on the features that are provided (map), and then compute a new cluster center for each group (reduce). Data points are clustered based on their feature similarity. We ran the Intel HiBench ML K-means workload on a data set of size of 3×10^6 points, where each point is a 20 dimensional feature vector, and with $k = 10$ groups. The job runs for a maximum of 5 iterations and tries to achieve a convergence distance of 0.5.

Figure 3 shows the performance results of running K-means on 16 executors, a number chosen after doing a similar workload profiling as for PageRank above. We see that running the same K-means job on only a subset of desired resources, *i.e.,* 4 executors (instead of 16), degrades the overall job execution time by a factor of $10\times$. Further, even with cluster size scaling, we see that the job still takes as much as $3.3\times$ more time when compared to the baseline (16 VM based executors on "Vanilla" Spark). Even though the VMs are available to use in $\sim$1 minute, the slowdown is due to the fact that a large fraction of the tasks are already scheduled on the existing executors which are overloaded and cannot be dynamically migrated to the newly available executors. Due to these queuing issues, VM based scaling may not be a good option for a latency-critical job. Since K-means is compute as well as somewhat I/O intensive, we see the effects of shuffling when running the job over Qubole's Spark-on-Lambda which shuffles over S3: it takes about 51% more time to finish the job. When comparing with SplitServe-Spark, we see that with an all-VM setup, we perform almost as well as Vanilla Spark even with an external shuffle over HDFS running on a node with a very modest I/O bandwidth. When we run the same job on SplitServe-Spark with only Lambdas, we see that as compared to Vanilla Spark, we do only 11% poorer. This extra time is attributed to the fact that, the HDFS node shares resources with the master/driver node, which is running on a m4.xlarge instance with only 750 Mbps dedicated EBS bandwidth, whereas, in Vanilla Spark the executors are running on a m4.4xlarge machine with 2,000 Mbps

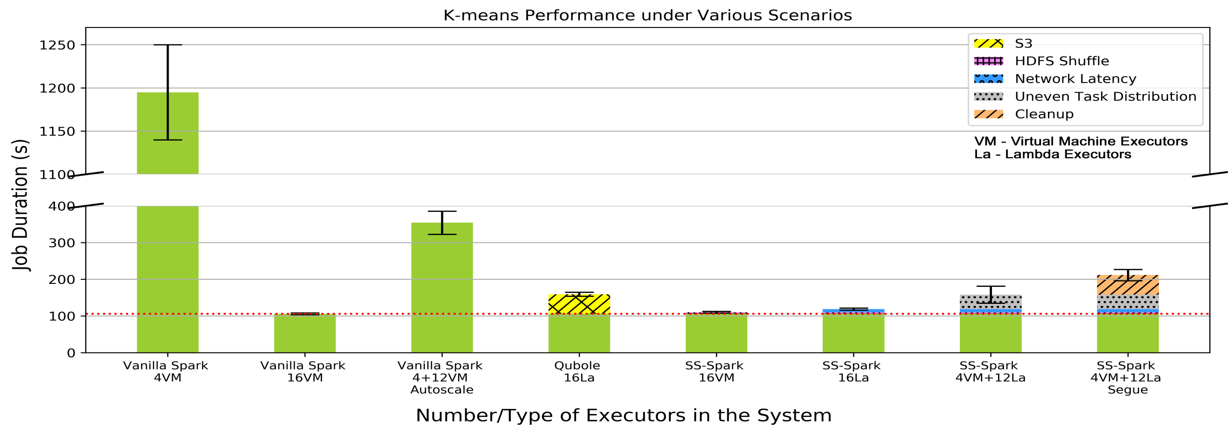


Figure 3: Comparing K-means performance on SplitServe-Spark and other systems under various scenarios.

of dedicated EBS bandwidth. This trade-off is tolerable since *distributed* K-means clustering is not very shuffle intensive.

Analyzing the case of splitting the work across both Lambdas and VMs, we see that our approach is 55% faster than scaling up the cluster. Since K-means is typically a longer running job, it may be vital that while the job is running some of its tasks on Lambdas, the cluster also procures more VMs (or perhaps waits for some existing resources to become available if this can be confidently anticipated) and segues the remaining work to these VMs as soon as possible. This is important as not only because Lambdas have a limited effective lifetime (owing to garbage collection issues for memory-intensive workloads), but also because, as seen in Figure 1, Lambdas can quickly become expensive. For SplitServe-Spark running both on VMs and Lambdas along with segueing from Lambdas to VMs, we still perform 40% better than only VM based scaling while saving 21% cost when compared to running the same job without segueing to VMs.

4.2.2 Websearch - PageRank

PageRank is an algorithm used by Google Search to rank web pages in their search engine results. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important a website is. The underlying assumption is that, more important websites are likely to receive more links from other websites. Specifically, we used Intel HiBench's WebSearch (PageRank) workload. This workload spends most of its time on iterations of several CPU-intensive tasks with moderate disk I/O and memory utilization (but considerably more than distributed K-means clustering). We run this workload with a data set size of 850,000 pages. The entire job concludes in 3 iterations. The baseline experiment involved 16 cores/executors of m4.4xlarge EC2 under Vanilla Spark.

Figure 4 shows a trend similar to K-means clustering.

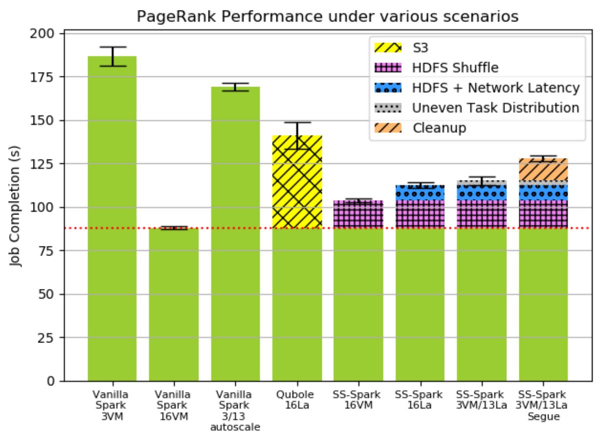


Figure 4: Comparing PageRank performance on SplitServe-Spark and other systems under various scenarios.

Running on only 4 cores, instead of 16, results in a performance degradation of around 2.1×. Even with VM based scaling, total execution time is worse by as much as 2×. Since PageRank is much more shuffle intensive than K-means clustering, we see the effects of large amounts of shuffling becomes more apparent. Since Qubole's Spark-on-Lambda uses S3, the overall execution time increases by more than 60%, but SplitServe-Spark's HDFS based shuffling increases by only 27%. SplitServe-Spark's execution time can be further reduced by running the master on a more provisioned EC2 instance.

As discussed in the previous sub-section, the worker nodes provisioned on m4.4xlarge get about 3× more dedicated EBS bandwidth as compared to the Master node provisioned on a m4.xlarge machine. In our experiments, we trade off performance with cost-savings to show the efficacy of our system even with stringent budgets. Similarly, other overheads such as network latency in case of Lambdas and "clean-up" overhead in case of segueing to VM based executors are amplified due to the increased shuffle data traffic. Even in a conservative setup, executing the workload over both VMs and Lambdas under SplitServe offers about 32% improvement on overall execution time when compared to VM based scaling. Combining the joint execution with segue, we still see a performance improvement of 24% along with a cost benefit of 8%. A smaller cost improvement is an outcome of using a smaller instance (in terms of resources) on which to locate master node. Since the master is a longer running entity, a cluster manager should assign it to a core of one of its largest VMs when the application is I/O intensive.

4.2.3 TPC-DS

When characterizing certain workloads, we realize that a they intensively use certain resources more than others. For such workloads, it is generally favorable to run them on instances that are optimized for these certain resources. For instance, ETL workloads tend to be much more I/O intensive than clustering-based workloads. TPC-DS is a decision support benchmark that models several generally applicable aspects of a decision support system. The queries generally have diverse compute and I/O footprints across them and are a prime example of ETL workloads. Specifically, we used DataBrick's Spark-SQL-Perf, a benchmark to test Spark's SQL performance. The TPC-DS workload suite consists of 100 queries, out of which we picked 10 with a range of compute and memory requirements and are I/O intensive (*i.e.,* heavy on shuffle data generation) and tested them over a range of scaling factors. Out of those 10 queries, we present the results of 4 queries (Q5, Q16, Q94 and Q95) which were run on a scale factor of 8. The results of other queries follow a similar trend. The workload is run on 32 cores using a m4.10xlarge instance to launch VM based executors. Since we want to match the performance of different systems as closely as possible, we run the SplitServe-Spark Master and HDFS on a m4.10xlarge instance as well to get similar dedicated EBS bandwidth.

Starting with Vanilla Spark results in Figure 5, we can observe that running the queries on only a subset of desired resource requirements can deteriorate the performance by up to 4×, or in some cases even more than that. Thus, not only are these queries I/O bound but they are also affected by the degree of parallelism they are subjected to, which underscores the need to characterize a workload as accurately as possible. Regarding I/O, one can see that running on Qubole's Spark-on-Lambda with 32 executors takes a staggering 21.7× more execution time on average[10]. SplitServe-Spark, when running on all-VM executors, compares closely with Vanilla Spark performing at par in most cases and doing only 1.6× worse in the worst case. Since there is a huge amount of intermediate shuffle data to be transferred over network, Lambda's unreliable and proportional to memory network bandwidth proves to be a bottleneck for SplitServe-Spark on all Lambdas, in the worst case performing ∼2.3× poorer than Vanilla Spark. Since SplitServe-Spark on VMs performs very similar to Vanilla Spark, combining VMs and Lambdas proves advantageous. As more tasks are pulled to faster VM based executors, we see a continuously improving performance. On average, SplitServe-Spark, when working with both VMs and Lambdas to address insufficient resources, takes 55.2% less execution time compared to VM based autoscaling. Since most of these queries finish executing under, or in some cases about, 60 seconds, the analysis omits experiments showcasing performance by seguing from Lambdas to VMs.

[10]Note that we were not able to get results on Q5 for Qubole's Spark-on-Lambda since their prototype encounters fatal errors while running this query.

4.2.4 Pi

SparkPi, an example of a compute-intensive application, approximates the value of Pi by performing a Monte-Carlo simulation. This is done by throwing n random darts on (selecting n random points in) a plane, upon which there is a circle of unit area. The value of Pi is then approximated by calculating the fraction of points which fell in the unit disk. This job is highly parallelizable by giving almost an equal number of tasks (darts to throw) to each executor in the cluster and finally accumulating the result by performing a simple count. Since *count* is basically a *reduce* operation, there is negligible shuffling involved. Hence, SparkPi is an example of purely compute-intensive workloads, with negligible memory footprint or I/O overhead. In our experiments, we approximate the value of Pi by generating 10^{10} random points and running the job on 64 executors. We use a m4.16xlarge VM as the worker node to run these executors.

In Figure 6, we can see how the various baselines work under different scenarios. We start by running the job on a Vanilla Spark cluster with the best possible case, i.e., the job finds the required resources available in the cluster. Comparing this with the case where only a portion of the required resources (4 executors) are available in the cluster, it can be seen that the job has taken more than twice as long to complete. For SplitServe-Spark with an all-VM executors setup (*i.e.,* all the executors run only on VMs and not on Lambdas), it can be observed that the performance is similar to that of Vanilla Spark. Using Lambda executors, we see that both Qubole's Spark-on-Lambda and SplitServe-Spark's all-Lambda setup give similar performance to that of Vanilla Spark. This is mainly due to the fact that there is no shuffling involved in this workload. Finally, a more interesting case is when we split the work across both VMs and Lambdas. Even here we see a similar performance to that of our (best) baseline. Again,

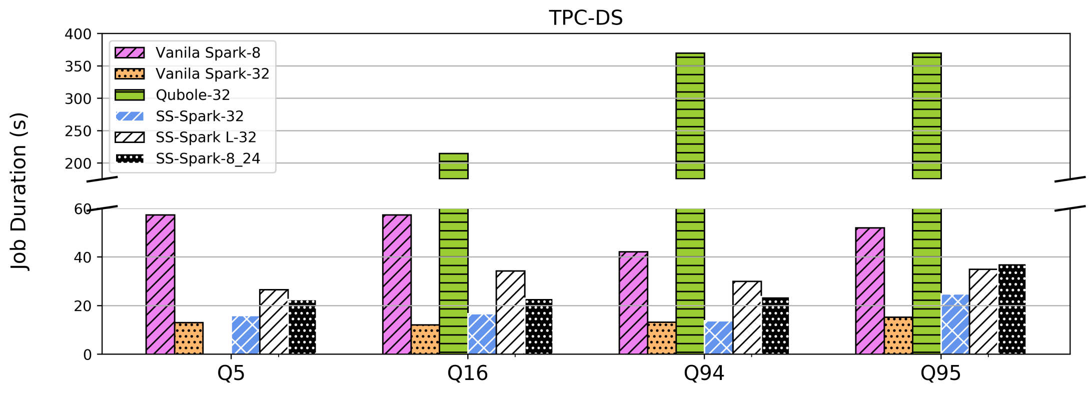


Figure 5: Comparing performance of Q5, Q16, Q94 and Q95 queries from Spark-SQL-Perf's TPC-DS workload suite.

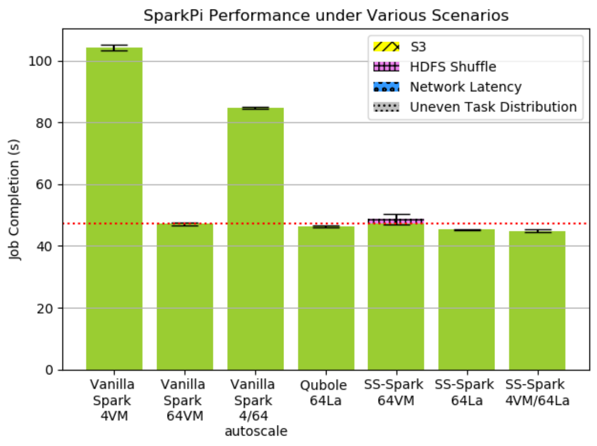


Figure 6: Comparing SparkPi performance on SplitServe-Spark and other systems under various scenarios.

we did not assess the Lambdas-segue-to-VMs setup under SplitServe-Spark because the job finished under 1 minute.

4.2.5 Summary of Results

Our implementation and experimental evaluations clearly show that SplitServe-Spark is suitable for fast autoscaling using AWS Lambdas for complex (stateful, I/O intensive and/or multi-staged with dependencies) workloads. By jointly using VMs and Lambdas, we show a 55% performance improvement for workloads with small to modest amount of shuffling and up to 31% improvement in workloads with large amounts of shuffling when compared to only VM based autoscaling. In addition, our results indicate that SplitServe-Spark along with novel segueing techniques can help save up to 21% of cost by still giving almost 40% execution time reduction.

5 Related Work

Table 1 summarizes the most important ways in which our proposed solution differs from the state of the art frameworks, most which making a case for either IaaS or FaaS. Application frameworks like Apache Spark [24], an open source distributed big-data processing engine, and Apache Flink [5], a framework and distributed processing engine for stateful computations over unbounded and bounded data streams, employ only VMs to run common, complex Map-Reduce and bulk synchronous processing (BSP) workloads [16, 8]. These workloads typically generate a large amount of intermediate shuffle data that are either communicated directly between functions or are persisted in storage. Even though these frameworks support dynamic executor allocation, scaling down the cluster has always been a tedious problem [6]. This is largely because, even after the task queues on these executors are completely flushed, they may continue their role as servers for the shuffle data for executors running on other slaves. Given the fault tolerance mechanisms built into these frameworks, there may be cost overheads and performance degradation due to the transferal of large amounts of state-data for execution roll-back. Further, if an executor is lost, the entire execution dependency is rebuilt using the transformation lineage that Spark internally maintains.

TR-Spark [30], which runs as a secondary background task on transient resources, goes about curbing performance degradation with fleeting executors using periodic checkpointing of the Spark's resilient distributed datasets (RDDs) [31]. I/O operations are fast and cost-effective since shuffle writes are to the local storage of the worker/slave nodes, instead of some external storage. However, since the executors are usually large in size, the straggler problems common to BSP workloads remain.

The other frameworks shown in the table propose running complex and in some cases particularly stateful workloads on stateless cloud functions like AWS Lambda. Most of these frameworks (including Qubole's Spark on Lambda [21] and PyWren [12]) use an external storage platform such as Amazon's Simple Storage Solution (S3) for the shuffle data. This allows individual Lambda functions to work as executors, relinquishing the CPUs as soon as the task is completed, and thus effectively mitigating the problem of stragglers.

Aspect ⟶	**Uses VMs?**	**Uses CFs?**	**Shuffling compares favorably with default Spark on VMs?**	
			Execution time	**Cost**
TR-Spark [30]	yes	no	no	n/a
Apache Flink [5]	yes	no	yes	yes
Qubole [21]	no	yes	no	no
Flint [14]	no	yes	no	no
PyWren [12]	no	yes	no	no
PyWren+Redis [20]	no	yes	yes	no
FEAT [19], MArk [32]	yes	yes	n/a	n/a
SplitServe	yes	yes	yes	yes

Table 1: A comparison of SplitServe against the state-of-the-art platforms exploiting VMs and Cloud Functions (CFs). The column on shuffling relates performance and cost with default Spark running on public-cloud VMs.

Since S3 is a multi-tenant service, there is a forced upper bound on the maximum number of I/O requests per S3 bucket. Although this may be relaxed as the number of buckets increases, the service usually tends to throttle when the aggregate throughput reaches a few thousands of requests per second. So, in general, associated workloads take a long time to finish even though the overall I/O bandwidth is comparable to that of a local disk write. Also, even though the per-write cost is relatively low, workloads like CloudSort [], which can trigger on the order of 10^{10} shuffle writes in single job execution, can incur enormous total S3 related costs. Note also that shuffle I/O may be reduced by employing data compression [7], but with added computational overhead.

Recently, [20] proposed to solve these problems by using a Redis cluster (fast storage) for all the intermediate shuffle I/O and S3 (slow storage) for only the initial input and final output. Redis, being an in-memory dictionary, significantly improves on I/O operations compared to disk writes, but is quite expensive as it requires the use of large VMs. Flint [14], another prototype of Spark on AWS Lambda, replaces AWS S3 with SQS [2] for intermediate data I/O using multiple distributed queues, which better suit a high number of small writes. SQS does better in terms of throughput but is costlier and less reliable compared to AWS S3.

FEAT [19] explores the idea of auto-scaling using Cloud Functions (CFs), particularly AWS Lambdas, by first launching an application on a pool of CFs. FEAT then determines the number of VMs and the number of cores on each VM required by the application, and subsequently launches these VMs in the background. When the VMs are ready, the control is transferred from the CFs. Note that FEAT considers workloads like publish/subscribe and request/response, which are largely stateless and scalable and thus a good fit for the serverless paradigm. In particular, these workloads do not face the problems of stragglers, intermediate data, or dependencies, and hence the do not need to work with external substrates for storage. Tasks of individual jobs are not, in any single point in time, divided between IaaS and FaaS services. Also, for stateless applications, CF-to-VM hand-off is straightforward with little threat of execution roll-backs.

6 Future Directions

Our future work will focus on SplitServe versions of other general-purpose (*e.g.,* Flink [5] and PyWren [12]) and special-purpose (Tensorflow [1]) application frameworks. Also, generally, a single core executor in a VM and a single-core cloud function may have different amounts of other IT resources. In future work, we will also explore the use of different task sizes for VMs and and CFs for better task-level load balancing. In principle, cloud functions could instead be orchestrated by a cluster manager (*e.g.,* Kubernetes and Mesos) in coordination with a cost manager, for a tenant simultaneously running multiple heterogeneous application frameworks. In this case, a SplitServe application framework would need suitable APIs to cluster and cost managers to procure and manage IaaS and FaaS resources – this will also be a part of our future work. Finally, our future work will also quantify the performance and cost impacts of SplitServe-Spark with other shuffle-intensive workloads such as TeraSort.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association.

[2] Amazon sqs. https://aws.amazon.com/sqs/.

[3] AWS Lambda. https://aws.amazon.com/lambda/.

[4] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[6] Databricks spark optimized autoscaling. https://databricks.com/blog/2018/05/02/introducing-databricks-optimized-auto-scaling.html.

[7] A. Davidson and A. Or. Optimizing shuffle performance in Spark. https://pdfs.semanticscholar.org/d746/505bad055c357fa50d394d15eb380a3f1ad3.pdf, 2013.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[9] Google Cloud Functions. https://cloud.google.com/functions/.

[10] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. pages 41 – 51, 04 2010.

[11] IBM Cloud Functions. https://cloud.ibm.com/openwhisk/.

[12] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proc. ACM SOCC*, 2017.

[13] A. Kansal, B. Urgaonkar, and S. Govindan. Using dark fiber to displace diesel generators. In *USENIX XIV Workshop on Hot Topics in Operating Systems (HotOS)*, May 2013.

[14] Y. Kim and J. Lin. Serverless Data Analytics with Flint. https://arxiv.org/pdf/1803.06354.pdf, 2018.

[15] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 427–444, Berkeley, CA, USA, 2018. USENIX Association.

[16] D. Krizanc and A. Saarimaki. Bulk synchronous parallel: Practical experience with a model for parallel computing. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, PACT '96, pages 208–, Washington, DC, USA, 1996. IEEE Computer Society.

[17] AWS Lambda Limits. https://amzn.to/2vH102F.

[18] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 423–430, Washington, DC, USA, 2012. IEEE Computer Society.

[19] J. Novak, S. Kasera, and R. Stutsman. Cloud functions for fast and robust resource auto-scaling. https://rstutsman.github.io/papers/feat.pdf.

[20] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association.

[21] Qubole Announces Apache Spark on AWS Lambda. https://www.qubole.com/blog/spark-on-aws-lambda/.

[22] J. Raj, M. Kandemir, B. Urgaonkar, and G. Kesidis. Exploiting Serverless Functions for SLO and Cost Aware Tenant Orchestration in Public Cloud. In *Proc. IEEE Cloud*, Milan, July 2019.

[23] Redis. https://redis.io/.

[24] Spark. spark.apache.org.

[25] Spark-SQL-Perf Benchmark. https://github.com/databricks/spark-sql-perf.

[26] Im afraid youre thinking about AWS Lambda cold starts all wrong. https://theburningmonk.com/2018/01/im-afraid-youre-thinking-about-aws-lambda-cold-starts-all-wrong/.

[27] Cold start / Warm start with AWS Lambda. https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/.

[28] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam. To move or not to move: The economics of cloud computing. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*. USENIX Association, 2011.

[29] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking Behind the Curtains of Serverless Platforms. In *USENIX ATC*, Boston, 2018.

[30] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 484–496, New York, NY, USA, 2016. ACM.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[32] C. Zhang, M. Yu, W. Wang, and F. Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *Proc. USENIX ATC*, Renton, WA, 2019.