Cross-Platform Performance Evaluation of Stateful Serverless Workflows

Narges Shahidi
dept. of CSE
The Pennsylvania State University
University Park, PA
nxs314@psu.edu

Jashwant Raj Gunasekaran
dept. of CSE
The Pennsylvania State University
University Park, PA
jashwant@psu.edu

Mahmut Taylan Kandemir
dept. of CSE
The Pennsylvania State University
University Park, PA
mtk2@psu.edu

2021 IEEE International Symposium on Workload Characterization (IISWC) | 978-1-6654-4173-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/IISWC53511.2021.00017

***Abstract*—Serverless computing, with its inherent event-driven design along with instantaneous scalability due to cloud-provider managed infrastructure, is starting to become a de-facto model for deploying latency critical user-interactive services. However, as much as they are suitable for event-driven services, their stateless nature is a major impediment for deploying long-running stateful applications. While commercial cloud providers offer a variety of solutions that club serverless functions along with intermediate storage to maintain application state, they are still far from optimized for deploying stateful applications at scale. More specifically, factors such as storage latency and scalability, network bandwidth, and deployment costs play a crucial role in determining whether current serverless applications are suitable for stateful workloads.**

In this paper, we evaluate the two widely-used stateful serverless offerings, Azure Durable functions and AWS Step functions, to quantify their effectiveness for implementing complex stateful workflows. We conduct a detailed measurement-driven characterization study with two real-world use cases, machine learning pipelines (inference and training) and video analytics, in order to characterize the different performance latency and cost trade-offs. We observe from our experiments that AWS is suitable for workloads with higher degree of parallelism, while Azure durable entities offer a simplified framework that enables quicker application development. Overall, AWS is 89% more expensive than Azure for machine learning training application while Azure is 2× faster than AWS for the machine learning inference application. Our results indicate that Azure durable is extremely inefficient in implementing parallel processing. Furthermore, we summarize the key findings from our characterization, which we believe to be insightful for any cloud tenant who has the problem of choosing an appropriate cloud vendor and offering, when deploying stateful workloads on serverless platforms.

***Index Terms*—Serverless Computing, FaaS, Stateful, AWS, Azure, AWS Step Function, Azure Durable Function.**

I. Introduction

Serverless computing is a cloud execution model that makes the cloud responsible to handle all the infrastructure complexity and allows the user to solely focus on the application code, rather than getting buried into the deployment infrastructure/platform specific details. Function-as-a-service (FaaS) is an example of cloud serverless computing that allows the user to write code and conveniently deploy it in the cloud [1]–[3]. A growing number of tenants are adopting to the serverless model owing to its prolific advantages such as instantaneous function scalability, completely cloud provider managed infrastructure, disaggregation, and pay-per-use billing. Essentially, serverless computing completely hides server maintenance and management from the tenants.

Current serverless platforms predominantly support 'stateless' applications. Stateless applications consist of independent entities such as real-time inference queries, that do *not* store/share information or state across invocations. While stateless applications are the most suitable fit for serverless, several traditional 'stateful' applications are also tending to move into serverless owing to the advantages we stated earlier [4], [5]. However, to leverage the benefits of serverless computing, tenants are required to completely redesign applications, especially the stateful ones.

Typically stateful applications are long running applications that require working with data, keep a shared mutual state between compute engines, and require the cloud infrastructure to keep the state of application. This entails binding the function to remote cloud storage, leading to additional network latency, and storage latency. Besides that, the remote storage does not provide guaranteed consistency, which is often required in the applications that require coordination of the distributed states. As much as it is easy to scale-out the independent and isolated compute units, it is quite hard to scale out the shared state between the compute units, the data storage, and the state of the workflow. Hence, once we couple the the application to its data, the benefits of portable, scalable pure compute functions no longer hold good. Essentially, it is unfavorable to retain the superiority of FaaS model, when it becomes stateful.

The attention to the FaaS, along with the demand for state sharing, motivated cloud providers such as Amazon and Microsoft to offer service for stateful applications. Amazon has recently developed the AWS Step function, which allows customers to define state machine transitions for the workflow with the ability to make it interactive with the customers [6], [7]. Microsoft has also recently developed an extension called Durable Functions, which allows workflows to be built as stateful orchestrators and entities which can be shared and addressed by other serverless components [8]. While these offerings support stateful deployments, we identify **several crucial inefficiencies** in the decision making process for a user in choosing the right platform:

• To deploy applications using AWS Step functions, users need

978-1-6654-4173-5/21/$31.00 ©2021 IEEE
DOI 10.1109/IISWC53511.2021.00017

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

to define a state machine in an AWS-specific custom representation which requires manual effort. Moreover, complex control flow within the application are difficult to represent using the AWS state machine.

• Azure's durable offering provides a straightforward programming model (C#, Python.), but the performance often suffers due to inefficiencies within its execution model [9], [10].

• In addition, each provider has a different 'pricing model', which makes it difficult to make cost-aware decisions. The workflow has to be carefully and conservatively designed to reduce the number of state transitions, since they directly contribute to the price. Certain providers [11] are also known to charge for idle periods of the functions, thereby regressing from the serverless paradigm.

To overcome some of these challenges, several research studies in the literature have proposed new designs for stateful FaaS offering [12]–[15]. However, our primary focus in this paper is to unearth the different cost, performance and latency trade-offs of commercial stateful serverless providers. While prior characterization efforts for serverless [11], [16] are focused on stateless single function model, to the best of our knowledge, this is the first paper which evaluates the two predominantly used stateful offerings, namely, Azure Durable Functions and AWS Step Functions.

To this end, the **key contributions** of this paper can be summarized below.

1) We deploy two different, widely used applications – machine learning pipelines (training and inference) and video processing – on AWS step functions and both Azure durable orchestrators and entities.
2) From our deployment, we characterize the cost versus performance trade-offs as well as the impact of cold start delay for both applications across different configurations, including both stateful and stateless deployments.
3) Our characterization reveals several key insights. Specifically, Azure durable extension provides a better programming model in that it is much simpler to develop stateful workflow on Azure when compared to AWS. However, the execution model is better in AWS because Azure orchestrators show high scheduling latency especially when scheduling a large fan-out of parallel threads.
4) We find that, in terms of pricing, AWS fares better primarily because Azure durable extension also charges for certain idle time periods. Furthermore, we ascertain that AWS Step price model is more close to the serverless model, as the customer pays for only for usage periods and not for the idle time.
5) Overall, our experimental evaluations indicate that AWS is 89% more expensive than Azure for the machine learning training application, whereas Azure is 2× faster than AWS for the machine learning inference application. Our results indicate that Azure durable extension is ineffective for implementing parallel workflows.

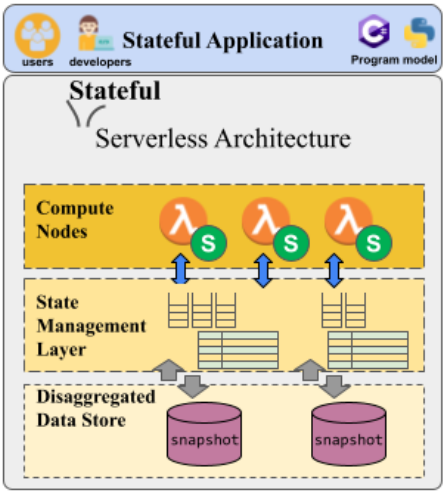


Fig. 1: A stateful serverless architecture.

II. Background

In this section, we discuss the challenges in designing a stateful architecture for serverless platforms and further discuss the extensions provided by current cloud providers.

A. Stateful Serverless Architecture

As mentioned earlier, there are several design and deployment challenges when deploying stateful applications using the serverless model. Long running workflows such as machine learning pipelines and data analytic codes need to be carefully redesigned and further make use of the ephemeral external storage systems, e.g., AWS S3 to maintain state to transfer data between multiple stages of the workflow [4], [5], [17].

The attempt to build this applications by tying isolated stateless functions through slow remote data store results in applications that are slow and not scalable. In effect, this approach basically moves the complexity of managing the data coherency and consistency to the very slow storage layer, making the application even slower in managing data.

The above observation highlights the need for an intermediate layer to take the responsibility of managing the application ephemeral data. Figure 1 shows an architecture for stateful serverless platform that (i) leverages the middle layer to bring the shared data closer to the compute node to increase performance, and (ii) talks to the lower storage layer for increasing redundancy and reliability. The middle layer also provides a message-passing bus for event dispatching, and load balancing. It also allows the compute node to handle fine-grain state of the application, while using coarser-grain data to talk to the storage layer, to increase throughput and efficiency. In the next two sub-sections, we discuss two cloud platforms that provide stateful architecture as an extension to the stateless FaaS.

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

B. Azure Durable Functions

Azure Durable Function is an extension to Azure function, which is the serverless compute solution from Microsoft. Azure Durable extension allows the customers to write stateful workflows in a serverless stateless environment. To fit more general purpose applications into the Azure function programming model, Azure adds two stateful components, namely, Durable Orchestrators and Durable Entities. The durable orchestrator is responsible for managing independent activities or entities, into a single sequential workflow. Durable entities on the other hand are 'class-like' structures which define a state and a set of functions to act on the state. In the rest of this section, we discuss the programming, execution, and price models of the Azure Durable components.

Programming Interface: The programming interface for Azure Durable functions is defined in the most commonly used programming languages (e.g., Python, Java, C#). Orchestrators can get use of the normal programming control flow to define the workflow. They can also communicate easily with stateless activities, as well as by building or accessing an instance of the stateful entities and invoking an operation to alter the state of the entity. One limitation with the orchestrators is that they need to be deterministic, since they will be replayed several times during the execution. Another limitation is the payload size on the result and argument passing for function calls, which is currently limited to only 64KB. Azure entities, on the other hand are created implicitly and permanently persisted with much large storage size (few MBs) for the states. They are further accessible from other functions including the orchestrator. In some implementations, the stateful entities can also communicate with each other; for example, one entity can invoke an operation on another entity.

Execution Model: The Orchestrators operate based on an event-source mechanism. The trace of orchestrator events is maintained by the durable task framework in a reliable event history table. The durable task framework is responsible for managing the state, and transferring the messages to/from the orchestrators. The orchestrators are unloaded from the memory while waiting for the function calls to return back. Once they are waken up again, the orchestrator is re-executed, but this time will consult with the event history table to skip the functions that are already executed. Entities are also implemented on top of logical containers called Task Hubs, which allow the entities and orchestrators to communicate freely with each other. Task hub enables this messaging via control queues and history tables. To prevent race condition on the Entity state, the operations of the entity are serialized.

Price Model: The stateful cost of the Azure Durable extension is determined by the Task Framework used to implement the connection layer. The customer will be charged based on the number of queue and table transactions. The queue polling happens constantly waiting for events to be received for the orchestrators or entities. As mentioned before, orchestrators are unloaded from the memory while waiting for activity or entity call backs. During this idle time, the customer is not charged for the orchestrators; however, the orchestrator replay is counted towards the GBps consumption of the application.

C. AWS Step Functions

Programming Model: AWS Step functions allow the user to develop a workflow of Lambda functions using a state machine. The state machine allows fan-out and fan-in structures for parallel execution of multiple tasks. The state machine allows the transition of results and arguments between the states of the machine. It is to be noted however that there is a limitation of 256KB on the payload size. Additionally, there are workflow structures that are not easily implementable by the AWS Step functions. For example, implementing dynamic parallelism or an iteration is a tedious job using these state machines.

Execution Model: A client scheduler is responsible for scheduling the functions invoked from the state machine. Every action and state transition are recorded in a table to be easily recovered to continue the execution.

Price Model: The pricing model for the AWS Step Functions is based on the state machine transitions, and the user is charged based on the number of state transitions that took place during the execution. We believe that this pricing model matches better with the pay-per-use model for the serverless services, since the user only pays for the number of transitions when the workflow is actually running.

III. Case Studies

In this section, we discuss two real-world applications that could benefit significantly from a stateful serverless platform – a machine learning model training and inference, and a highly parallel video processing application.

A. Machine Learning Pipelines

The machine learning model training process in real-life applications is a workflow of several steps, including a series of data cleaning and preparation steps, followed by an iterative training step to find models and parameters that fit the training data, and finally, the prediction workflow to test the model on the test data set. Although the serverless paradigm can be a good fit for this kind of applications, designing such a workflow with serverless and stateless function instances becomes an arduous task, due to the complex workflow structure and data transfer and sharing between several isolated functions. In this case study, we design and implement a serverless workflow for a real-life application in model training. Our design has several components:

Model Training workflow: Figure 2 shows the training workflow for this workload. It includes several steps: First, data preparation, where non-numerical data are encoded, and scaled to a specific range; Second, dimension reduction that selects the representative features and reduce the problem complexity; and finally, Model Selection, where a set of parameters are explored, and different models are trained and the best one is selected for the training dataset.

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

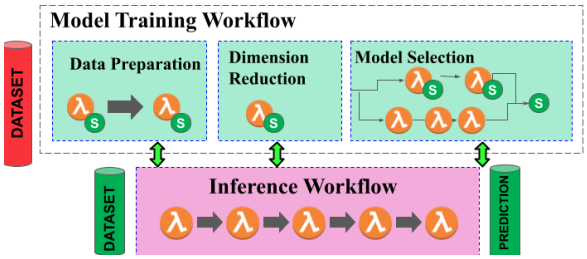


Fig. 2: Model training workflow includes three main steps; data preparation, dimension reduction, and model selection.

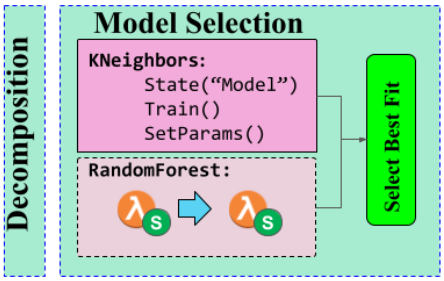


Fig. 3: The model selection step starts training several parallel tasks, to explore different models, or different set of parameters. A collector entity collects the results and select the best model among the trained models.

Model Selection Workflow: This workflow, depicted in Figure 3, consists of several parallel workflows, each focusing on a different algorithm, and parameter space, to train a model. The last step is to select the best fit, which aggregates the results of all parallel model training workflows, and finds the most fitted model. Each algorithm can be implemented as a simple stateless function, or a more complicated function chain, or a stateful workflow. Each separate workflow should report the selected model to an entity, which is responsible for selecting the best fit among the trained models. In the model training step, depending on the training complexity, different durable components are used. For larger models (e.g., Randomforest), we used a sub-orchestrator which uses a combination of stateful *Entities* and stateless *Activities* to build the model. For smaller and faster models (e.g., KNeighbors) we used a stateful entity with a few operations to train the model. The parallel tasks for model training report their results to an Entity, which is responsible for collecting all the models and determining the best one among them. The state of this entity is updated once a new model is found with less error reported than the current model.

Inference Workflow: This workflow tests the selected model on the test dataset, and outputs the predicted results. It requires accessing the data preparation and dimension reduction matrices, and the selected models to test on the data. One interesting aspect of the Azure Entities is that they are addressable by any Orchestrator or even other Entities. We leveraged this

```
1  import azure.durable_functions as df
2  import pandas as pd
3
4  def orchestrator_function(context: df.
       DurableOrchestrationContext):
5
6      # get the input data ...
7      data = pd.read_csv(download_path, ',')
8
9      # Access pre-trained feature engineering
       entities.
10     encEntity = df.EntityId("Encoding", "OneHot
       ")
11     scalarEntity = df.EntityId("Scalar", "
       scalar")
12     pcaEntity = df.EntityId("DReduction", "PCA"
       )
13
14     # Function chain for feature engineering.
15     data_encoded = yield context.call_entity(
       encEntity, "encode", data)
16     data_scaled = yield context.call_entity(
       scalarEntity, "scale", data_encoded)
17     data_decomposed = yield context.call_entity
       (pcaEntity, "decompose", data_scaled)
18
19     # Get the best model from related entity.
20     modelEntity = df.EntityId("ModelSelection",
        "best_fit")
21     best_model = yield context.call_entity(
       modelEntity, "get")
22
23     # Call a function to apply the model and
       return the prediction results.
24     prediction_results = yield context.
       call_activity('Inference', [best_model,
       data_decomposed])
25
26     return [prediction_results]
```

Fig. 4: Inference workflow with Azure durable extension: #9-12 accesses the entities to get pre-trained data preparation matrices. #14-17 run a function chain for data preparation. #19-21 query the best fit model from the ML trainer, and applies to the input data (#24).

feature in the inference workflow to access the data preparation objects, and the best fitted model object from the entities that keep the latest results. Figure 4 shows a code snippet of the inference workflow implemented with Azure Orchestrator and Entities. The workflow accesses the previously trained data preparation instances (Line 10-12), then form a function chain to pass input data through data preparation steps (Line 15-17). In the next step, the workflow request the best fit model from the `ModelSelection` entity which returns the model object (Line 20-21). Finally, once the input data passed the feature engineering steps, and the model is also available, they will be passed to a stateless function named `Inference` to get the result of the prediction (Line 24).

In our experiments, we had two types of implementation for each workflow, namely, "stateless" and "stateful". The stateless implementation chains the functions and leverages the traditional communication channels, e.g., remote storage, queues, etc. to connect the results and arguments of func-

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

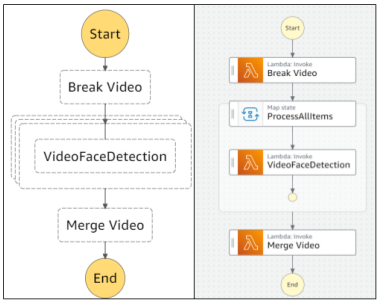


Fig. 5: The design of video processing application with AWS Step functions. Dynamic parallelism is implemented using Map State.

tions, while the stateful implementation employs the platform durable extensions to build the workflow combined with the stateless functions whenever required.

The AWS Step function implementation allows defining the workflow in a state machine. We used the Step Functions to define the model training and inference workflows. In AWS implementation, the inference workflow needs to access the slow remote storage to receive the pre-trained model. We also used Lambda functions to implement each of the steps shown in Figure 2. The implementation with Azure Durable extension is more exciting as the programming model allows user to define class-like entities, similar to an object-oriented programming model. The inference workflow as shown in Figure 4 accesses the entities directly to get the pre-trained models. This allow the training workflow to work continuously on making a better model, while deploying the same model for the prediction.

B. Video Processing

Cloud serverless instances have recently become popular among video processing applications, since they allow thousands of tiny threads to be running in parallel. Through this extreme parallelism, serverless functions provide real-time performance for these applications. However, many of these several threads might require sharing the data across stages of the tasks, which breaks the very parallel nature of the application. Previous implementations [5] often have used a hybrid model to incorporate a coordination server to control the threads, and to help for communicating data between the threads. The stateful programming components offered by Azure and AWS however provide ways for many parallel threads to talk to each other without the need for any coordination or control server.

In this case study, we considered using the stateful functions to create a video processing application that can benefit from the highly parallel functions to reduce the response time. Figure 5 shows the state machine designed with AWS Step functions. In our implementation, we used a multi-step stateful workflow that has three steps: First, a sequential step to break the video into chunks, small enough to be transferred to the next step without the need for remote storage; Second, a homogeneous army of CPU-intensive, computationally expensive threads to run a deep learning algorithm on the video chunks; Finally, a merging component that aggregates the results of the parallel workers to form the complete result. As shown in Figure 5 three Lambda tasks are used for each step, and the parallelism is designed using MapState in the AWS Step function workflow.

IV. Evaluation Methodology

We evaluated our implementation on two different cloud platforms with varying configuration parameters, the details of which are explained below. .

A. Platforms and Metrics:

Platforms: In this study, we focused on two of the known cloud platforms for serverless computing: *AWS* and *Azure*. All the experiments in this paper used 'Azure Consumption Plan' and 'AWS Developer Plan'. We also used Azure Durable Extension [8] for stateful implementation of Azure workflows. The orchestrators are used to form the workflows which are triggered by an HTTP Client. Stateful entities are used for each step that requires state maintenance, and Azure Activity are used whenever an stateless isolated function was required.
Latency: To measure the end-to-end latency of the Azure stateful implementation, a timer is fired when the main orchestrator function is changing the state from 'Pending' to 'Running', and it stops when the orchestrator state reports 'Completed'. All AWS Step state machines needs to have a 'Start' and an 'End' state, which capture the beginning and end of the workflow. The end-to-end latency is measured as the time elapsed between timestamp reported by 'Start' state, until the 'End' state. We also implemented the ML training workflow with a set of Azure functions connected by Azure queues. For this implementation, the end-to-end latency is reported from the HTTP trigger timestamp, until the last function of the chain finishes the execution.

To measure the cold start delay, each workflow is run for four days, with the rate of one request per hour. The cold start delay for the orchestrators is measured as the time between the 'Pending' and 'Running' states of the orchestrator. The cold start delay for AWS Step functions is measured based on the delay between the timestamp of the 'Start' state, and the first function of the workflow.
Log Collection: During the experiments, we often relied on **AWS cloudWatch** and **Azure Application Insight** to collect the results. Table I shows the serverless platform configuration and run-time environment. For both AWS Step Functions and Azure Durable Functions, there is a payload size limit on the amount of data that can be transferred between the functions in the workflow. For AWS Step Functions, the payload size limit is 256KB according to the latest update in 2020 [18]. Azure Queues also allow 256KB of payload size; however, the Azure Durable extension only allows a 64KB payload size for cross-function communication [19].

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Serverless platform configuration.

	Run Time	*Region*	*Memory Configuration*	*Time Limit*	*Payload Size*
AWS	Py 3.7	West US 2	1.5GB	15min	256KB
Azure	Py 3.7	US East	1.5GB	30min	64KB

Workloads: We evaluated the two case studies described in Section III. The Machine Learning Training workflow, which builds a regression predictive model for car pricing, is implemented using Python library `sklearn`; it searches through different algorithms with a range of parameters to find the best fit model. The model sizes are ranging from 100KB to 5.2MB in this example. Two datasets have been tested, small and large, with 200 and 10K rows, respectively. The datasets have 26 features, 12 of which are not numerical and require encoding and scaling during the feature engineering steps. Dimension reduction is based on the Principal Component Analysis (PCA), and makes use of the `sklearn.decomposition` library in Python. Model selection is searching through `RandomForestRegressor`, `KNeighborsRegressor`, and `Lasso` to find the best fit model. Table II reports the number of isolated functions used to build the workflow for each of the implementations. Note that, we used the cross-function communication on ML training workflow only to transfer training objects, e.g., encoding, scalar, dimension reduction matrices, and the trained models. However, since the dataframes are often larger than 256KB, we had to transfer them via the remote storage.

Since the operations on entities are serialized, we noticed that adding long running operations that require high throughput to the entities is not a good option. For the read-only operations that only require reading the state and performing heavy operations on it, it is better for the state to be fetched out of the entities and the operation to be implemented in the stateless functions. For example, in the inference workflow (Figure 4) rather than sending the data to the `ModelSelection` entity, along with an operation to do the prediction, we used `get` operation to read the model, and then call a stateless and scalable activity (`Inferenece`) to do the prediction. This allows several inference instance to work at the same time, and not to be bottle-necked by the `ModelSelection` entity.

For the Video Processing workflow discussed in Section III-B, we used the `OpenCV` library which is one of the commonly used tools to perform face detection, with an input video of size 100 MB from Sintel animation [20]. The first step of the workflow breaks the input video into smaller video chunks to run a face detection algorithm using a pre-trained deep learning model. The size of each chunk depends on the underlying payload size limit of each platform. The model size is 1MB which is fetched by each worker from the remote storage. The implementation spawns multiple workers that work in parallel on small chunks of the original video. Each worker runs a function for face detection and returns the video chunk back to the caller. The last step of the workflow merges the video chunks to construct the resulting video.

Table II describes various implementations of each workflow, and the code size of each implementation. The same name has been used in the graphs in the next section when discussing the results. The results are collected from running over one hundred iterations of each implementation, and the price is calculated without considering the free tier discount.

Price Calculation: We measured two components of the price to compare the cost of different implementations: computation cost, and transaction cost.

Computation cost is the stateless component of the price, which is often reported as Gigabytes-Seconds (GB-s) of the function execution. This value is calculated by multiplying the memory capacity by the execution time of the function. For Azure, the memory capacity is not configurable by the user. The maximum memory capacity is set to 1,536 MB in the consumption plan. While running the workflow, Azure infrastructure is recording the memory consumption of the workflow and report the consumption to the user based on the MB-milliseconds. In AWS however, for each AWS Lambda function, the user is able to change the memory configuration to multiples of 128MB in order to reduce the billed cost. The billing is based on the configured memory, and not the consumed memory, so the user has to carefully tune the memory configuration for each single Lambda function to avoid extra charges. The run-time execution of the function is rounded do the nearest 100 ms.

The transaction cost is the stateful component of the price which is based on the transaction cost of the tables and queues that are added either manually or used by the platform to keep the state of the workflow. The Azure platform charges the user based on the number of tables and queues transactions, including the queue polling of entities and orchestrators and the table read and write transactions for the event sourcing of the orchestrators. For the AWS Step functions, the stateful price component is calculated based on the number of state transitions of the state machine. All the pricing reported in the next section are without considering the free tier discount.

V. Results and Discussion

In this section, we report the results for the implemented workflow, and discuss the performance in terms of overall latency, cold start overheads, and finally the pricing of executed workflows. At the end of each application, we summarize the key insights.

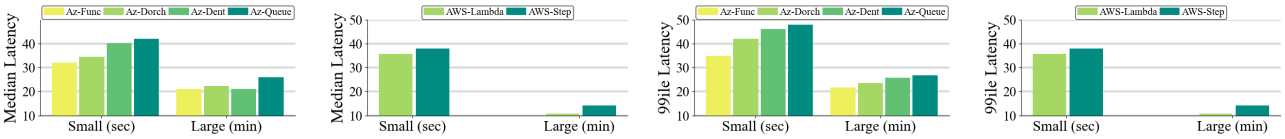
A. Machine Learning Training Workflow

Overall Latency: Figure 6a compares the end-to-end latencies for different implementations of the machine learning training workflow. As can be seen from these results, the pure stateless function (shown as *Az-Func*) results in the best overall latency. A function chain formed manually using queues as the connecting channels (*Az-Queue*) adds 30% and 24% to the overall end-to-end latency, for the small and large datasets, respectively. This is due to the latency that queue transition imposes on the overall workflow execution.

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Different implementations of the workloads.

Graph Reference	*Description*	*Stateful*	*ML Training* # of Func - Code Size	*Video Processing* # of Func - Code Size
AWS-Lambda	One stateless Lambda function.	No	1 λ - 63.1 MB	1 λ - 70.8 MB
AWS-Step	Workflow implementation using AWS Step Functions, calling AWS Lambda functions on each state.	Yes	4 λ - 271.2 MB	3 λ - 214.8 MB
Az-Func	One stateless Azure function.	No	1 λ - 304 MB	1 λ - 204 MB
Az-Queue	Isolated functions connecting through Azure queues.	No	4 λ - 304 MB	-
Az-Dorch	Workflow implemented using Azure Durable orchestrators, calling isolated function through `call_activity`	Yes	6 λ - 304 MB	3 λ - 219 MB
Az-Dent	Workflow implemented using Azure Durable orchestrators, calling stateful entities for operations through `call_entity`.	Yes	7 λ - 304 MB	-



(a) Median latency for Azure. (b) Median latency for AWS. (c) 99ile latency for Azure. (d) 99ile latency for AWS.

Fig. 6: End-to-end latency for Azure and AWS various implementations for ML Training workflow.

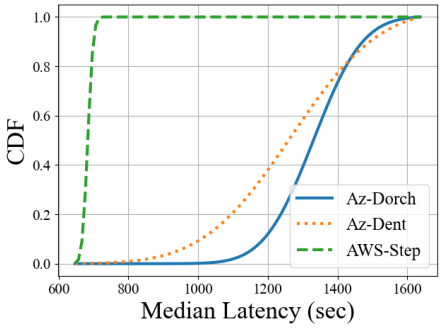


Fig. 7: CDF of the end-to-end latency on ML training workflow (large dataset).

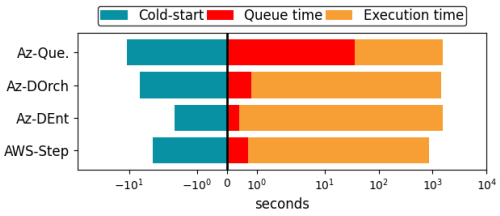


Fig. 8: ML training latency breakdown (large dataset).

Now, if we look at the durable implementations with Azure Orchestrator and Entities, shown as *Az-Dorch* and *Az-Dent*, the overall latency is somewhere between the stateless function (*Az-Func*) and queue-based implementation (*Az-Queue*), indicating that the transition overhead is lower with Durable implementations. For example, *Az-Dorch* shows only 5-7% increase in the latency, and *Az-Dent* results in almost the same latency as *Az-Func* for the large dataset – both performing

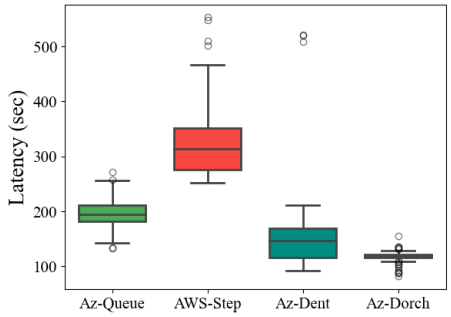


Fig. 9: End-to-end latency of the ML inference.

better than *Az-Queue*.

The same trend is also observed in the AWS Step function end-to-end latency plotted in Figure 6b. The step function implementation adds 6%, and 32% latency overhead due to the delay in calling the function chain, for small and large dataset respectively. Note that the AWS Step functions exhibit better overall latency compared to the Azure implementations, which can be due to the fact that the memory allocation to AWS Step is adjustable and a larger memory configuration can result in better latency.

The 99ile latency is reported in Figure 6d and Figure 6c. These results indicate the same trend observed in the latency case comparing different implementation of the same platform; however, AWS often shows better tail latency than Azure. To better show this difference, Figure 7 shows the CDF for the end-to-end latency for the 99ile latency of both Azure durable implementations, in comparison with AWS-Step, for the large dataset. The figure shows a sharp CDF graph for AWS-Step, whereas a long tail latency is observed on Azure Durable implementations. This can be due

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

to the long unpredictable latency of accessing the entity's state as well as the queue time of scheduling functions in the Azure.

Figure 8 shows the breakdown of the 99%ile latency, and separates the'Queue Time' from the active 'Execution Time'. The 'Queue Time' in this chart refers to the total delay of queue polling and transferring data in a chain of function calls. As the chart shows, the queue waiting time in *Az-Queue* is around 30 seconds, which is significantly higher compared to the queue waiting time in Azure durable implementations, which is often less than 1 second. The same figure shows higher execution time for Durable implementations, while running the same function logic. The reason for this is probably the orchestrator replay described in Section II-B. The orchestrator functions are replayed, often multiple times, during the workflow execution. For Az-Dent, the execution time is 8% more than Az-Dorch. This is due to the fact that the stateful entities are slower in running the same operations as the stateless activities.

We also evaluated the machine learning inference Workflow separately, with a new dataset for prediction. The workflow is leveraging the same model trained by the machine learning training workflow. Figure 9 gives the overall latency of the model serving algorithm on the large dataset. Az-Dent shows 24% more end-to-end latency than the orchestrator implementation in Az-Dorch. Also, AWS-Step reports 2× higher latency with 1.5GiB of memory configuration. The benefit on latency is due to the fact that Azure implementations allow the objects to be read from other entities, rather than accessing remote slow storage.

Cold Start: We measured the cold start delay for the durable implementations over the course of four days, by sending one request per hour. Figure 10 shows the delay in the cold start for each of the durable implementations, ranked from the highest to the lowest. We found that, in this workflow, Azure Durable extensions (Orchestrators and Entities) often lead to less than 2 seconds of start time, whereas AWS-Step start time is 3-5 seconds, and the Az-Queue implementation experiences 10-20 seconds for cold start delay. We suspect that this is due to the queuing of requests on a static pool of containers as observed by [11].

Cost: Figure 11a and Figure 11b show the stateless cost as GB-s for both the non-durable and durable implementations for Azure and AWS, respectively. AWS-Step shows the same GB-S as the AWS-Lambda, since the computation (execution time) is the same. The Azure implementation with queues also shows the same GB-s as the stateless implementation, named Az-Func. However, the durable extensions of Azure seem to add to the GB-s of the workflow execution. For the large dataset, Az-Dorch and Az-Dent result in 44% and 88% increases, respectively, in the GB-s compared to the stateless implementation. This is due to the fact that orchestrators and entities are often replayed, which adds to both the execution time and memory consumption.

Figures 11d and 11c compare the transition costs of the

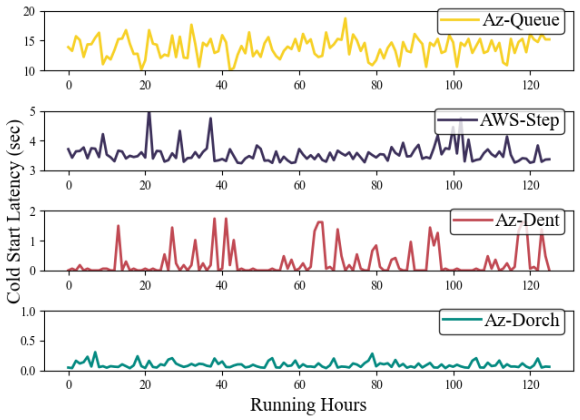


Fig. 10: ML training cold start delay (sec).

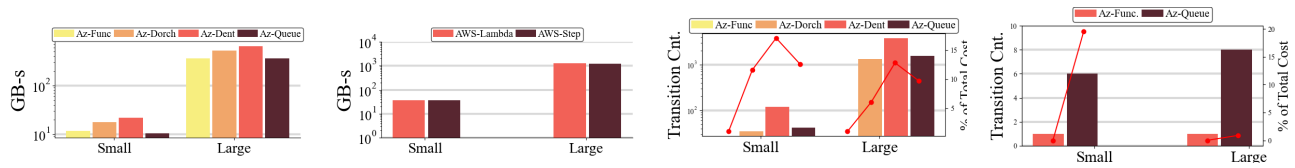
various implementations. It can be observed from these results that while Az-Dorch show almost the same amount of queue transactions with respect to Az-Queues, the cost of transaction is double in the Az-Queues. Overall, the stateful transition cost is often less than 10% of the total cost in the Azure implementations. Figure 11d indicates, the transition cost is 2% of the total cost of AWS for large data-set, hence it does not play an important role in the AWS pricing. The transition cost for AWS depends on the number of states in the state machine. The percentage is higher for small data-set (around 20%) confirming the fact that AWS step functions have to be used only for long running functions. For Azure, the transition cost is up to 10-15% of the total cost; however, the GB-s cost is much lower than the AWS computation cost (11a and 11b).

We noticed that there is a difference between the Azure and AWS costs for statefulness. Azure is charging the customers for the number of transactions on the queues and tables. Even though the queue polling rate is adjusted based on the function activity, the queue polling continues even when the function is not active. This adds to the user cost when the workflow is idle. This however does not happen on AWS Step functions, since the cost of statefulness depends on the state transitions of the state machine, whenever the function is running.

Key Takeaways:

- Although, Azure Durable extension excels in performance (overall latency), it imposes additional costs on the customer – in terms of both GB-S and transition cost.
- AWS Step functions show comparable performance with respect to the AWS Lambda, and offer superior performance with respect to the Azure implementations.
- We believe that the AWS Step function price model matches better with the serverless model, as it does not charge user for the idle periods of the function. Azure, however, imposes some charge on the user even when the workflow is not active.
- We noticed that running an operation with Azure Entities is slower than running the same operation in the stateless Azure activities.

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.



(a) GB-s of Azure variations. (b) GB-s of AWS variations. (c) Transition cost for Azure. (d) Transition cost for AWS.

Fig. 11: Cost comparison for Azure and AWS various implementations of ML Training workflow.

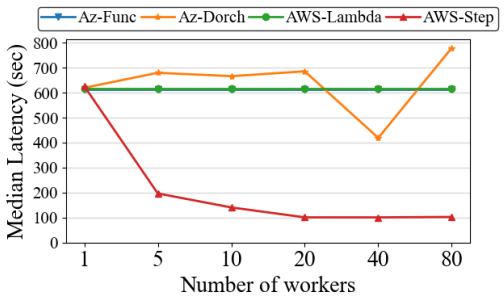


Fig. 12: End-to-end latency for video processing. *Az-Func* and *AWS-Lambda* report the same latency.

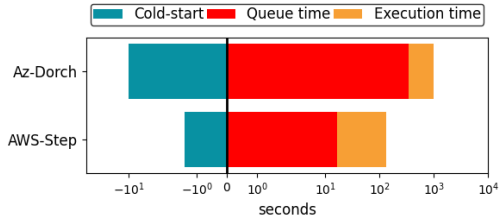


Fig. 13: Video processing latency breakdown.

- We found Azure durable extension to be very convenient for developing new stateful applications. Specially the Azure Durable Entities provide a 'class-like' model which is familiar for lots of developers. On the other hand, the AWS Step function interface facilitates the development and debugging of the state machine and allows much easier monitoring of the workflow through CloudWatch.

B. Video Processing

Overall Latency: Figure 12 shows the overall latency of video processing application when implemented using durable and non-durable functions. We increased the number of workers to see the effect of parallelism on the overall latency. With the AWS Step functions, when increasing the number of workers, the parallel part of the workflow speeds up. This results in more than 80% improvement in the latency with respect to the AWS Lambda function. Azure durable orchestrators however exhibit an unexpected behavior: when the number of parallel workers is increased, the overall latency does not decrease. In fact, in some cases, the overall latency increases by up to 25%.

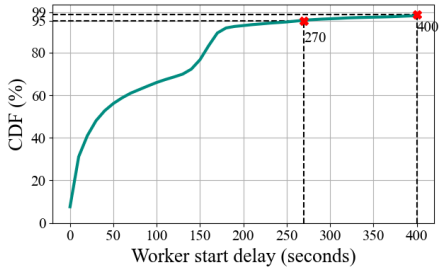


Fig. 14: Scheduling delay for 50,000 face detection workers.

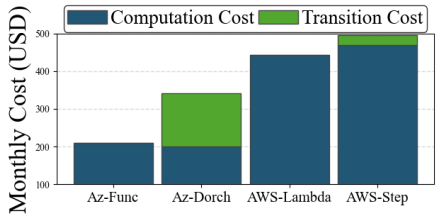


Fig. 15: Estimated monthly cost for Azure vs. AWS running the video processing application.

We could only see improvement with 40 workers running in parallel, but the trend did not keep up with 80 workers or more. We noticed two different behaviors in scheduling of Azure parallel workers. In some cases, the long tail latency was due to the cold start delay (e.g., 80 workers), when more containers had to start to accommodate the workers and in some other cases, this is due to the queue waiting time (40 workers) scheduled in one container.

To understand better what caused the Azure functions to experience slow-down when executing with a higher degree of parallelization, we collected the scheduling time of each worker. Figure 14 shows the scheduling delay collected from more than 50,000 workers. It is evident that almost half of the workers experience 40 seconds of scheduling delay, and 5% experience 270s (4.5 minutes) to start the function. This scheduling delay causes a very long tail for the execution time of the workers. And, the tail latency can affect the total round-trip time, when applied to the larger number of fan-out workers. Table III shows the execution time when the work is divided among 80 workers. The first line shows the latency of one worker, with a long tail of 744s. Even though this value is only reported by 1% of the workers, in the

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

scale of 80 workers, when the coordination between many tiny workers are required, it causes a very long median latency.

Cold start: Figure 13 plots the latency breakdown comparison between AWS-Step and Az-Dorch. Throughout our experiments, the AWS cold start delay for this applications remains in the range of 1-2 seconds, both for AWS Lambda, and AWS Steps. Azure Orchestrators however exhibit a wide range of delays to start the orchestrators, with an average being around 10 seconds which is 4-5$\times$ higher than AWS.

TABLE III: Finish time for the large fan-out in the video processing workflow in Azure.

	50%ile latency	***95%ile latency***	***99%ile latency***
One worker	244s	476s	744s
All workers	774s	798s	822s

Cost: Figure 15 shows the monthly cost of running the video processing workflow, with 20 workers. The computation cost observed with the Azure Durable implementation (Az-Dorch) is comparable to that of Azure Functions (Az-Func); however, the constant queue and event polling adds 70% transition cost for the Az-Dorch. AWS-Step and AWS-Lambda result in higher computation costs due to the fact that they require a larger memory configuration to deliver the same latency (2GB used in this case). The transition cost however is around 5% of the total cost, and is 83% less than the Azure.

Key Takeaways:

- Implementing parallel workers in AWS Step state diagram is very difficult, especially when dynamic parallelism is required in the workflow. Azure durable orchestrator library, however, allows dynamic parallel workers to be implemented with a single line of code.
- Azure durable functions show a resistance towards scheduling parallel workers, either due to the cold start of the containers, or due to the long queuing time, leading to a long tail latency for the workers completion time.
- The cost of transition in Azure durable implementation is often higher than the AWS state machine transition. This is due to the fact that the number of queue transactions is often more than what is required for the application.
- Azure computation cost (GB/s) is often lesser when compared to AWS. This we believe is due to the fact that the Azure platform optimize the memory configuration to what works best for the function, however, in the AWS user is responsible to tune the memory configuration.

C. Implications from this characterization

There are several implications which can be derived from our characterization as listed below. First, tenants can make well informed decisions while choosing the right service provider and/or configuration for running similar workloads as elucidated in our results. From our observations, we can conclusively say that compute intensive and high performance applications which majorly rely on parallel processing are more effective on AWS. In contrast, complex workflow such as DAG based applications are more convenient to develop on Azure due to quicker development time. Second, cloud providers can enhance their existing policies for resource management. For instance, the Azure execution model needs to improve the performance of the durable components, with faster access latency to the states and also lessen the container startup overheads. Similarly, AWS programming model needs to improve in order to support more dense workflows. We believe that characterization studies like this will pave way for more robust systems to be built.

VI. Related Work

1) Commercial Serverless Providers: Among the commercial stateful platforms, Cloudstate [21] is an open-source project recently developed based on the Google Cloud infrastructure, and extends the Kubernetes ecosystem by a state management service for stateful applications. Note that Cloudstate proposes a design and a reference implementation of the protocols and API libraries that allows user to focus on writing user functions and stateful entities, while the state management is delegated to the Cloudstore library. Netherite [10] on the other hand proposes an improvement over Azure durable execution model by introducing optimizations such as partitioning stateless tasks and stateful instances into fine-grained groups, and committing the recovery logs into the high performance devices such as SSDs. On the other hand, Azure has also redesigned some of its resource management policies [22], [23] to overcome cold start overheads when starting new functions. Several start-ups [24]–[27] have started to adopt serverless-like solutions, either on existing cloud vendors or on their own private infrastructures.

2) Academic Proposals: Academic community also has proposed stateful serverless platform design [12]–[14], [28]–[31]. Cloudburst [12] is one such work that targets auto-scaling FaaS which requires distributed session consistency across the compute nodes. Their proposed design and protocol enable cross-function communication via TCP connection between function threads. Pocket [29] extends the storage tier to provide support for ephemeral data, while Shredder [28] proposes to move the compute to storage nodes. Cloud providers can incorporate certain design choices to alleviate the limitations alluded from our observations in this paper.

3) Application Redesign: There are several proposals [4], [5], [17], [32]–[35] that target at redesigning applications to suit the serverless model, which often involve designing a hybrid serverless/serverful model to overcome the shortcomings of not maintaining state. ExCamera [5], for example, implements a video processing application using thousands of tiny threads. Pywren [17], [36] redesigns data-analytics applications to execute in a serverless platforms, while [4], [37] implement machine learning pipelines in serverless settings.

VII. Concluding Remarks

Serverless computing has brought in tremendous attention to deploy modern day applications, with benefits for both

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.

tenants and cloud vendors. It predominantly supports stateless applications, and a major challenge lies in adapting this model to the versatile stateful applications which could potentially benefit form the serverless paradigm. Cloud vendors have started to accommodate this market demand by expanding their infrastructures to support stateful serverless applications, which comes several design and deployment challenges.

In this paper, we conduct extensive experiments with the goal of evaluating the two of these new stateful serverless platforms, namely, AWS step functions and Azure durable functions. We redesign two frequently used applications – machine learning training and parallel video processing – to fit both the AWS and Azure stateful models and exhaustively characterize the cost performance and latency trade-offs. Our key findings from our characterization include the following: (i) Azure durable components provide a very convenient programming model to develop a stateful application, while AWS Step functions require significant effort; (ii) Azure Durable functions do not scale well with parallel processing applications especially when increasing the number of threads beyond 20; (iii) Overall, AWS is 89% more expensive than Azure for for the machine learning training while Azure is 2× faster than AWS for the machine learning inference.

VIII. Acknowledgment

This work is supported in part by NSF grants 1908793, 2119236, 1931531, 2028929, and 2122155.

References

[1] "Aws lambda serverless functions," 2020. [Online].
[2] "Azure serverless functions," 2020. [Online].
[3] "Google cloud functions," 2020. [Online].
[4] 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)*, pp. 334–341, 2018.
[5] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, (Boston, MA), pp. 363–376, USENIX Association, Mar. 2017.
[6] "Aws step functions documentation," 2020. [Online].
[7] J. Prasad Buddha and R. Beesetty, *A Brief History of Time: From the Big Bang to Black Holes*. Springer, 2019.
[8] "Azure durable functions," 2020. [Online].
[9] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Ruiz Ollobarren, and D. Arroyo Pinto, "Comparison of faas orchestration systems," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 148–153, 2018.
[10] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Serverless workflows with durable functions and netherite," *CoRR*, vol. abs/2103.00033, 2021.
[11] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 133–146, USENIX Association, July 2018.
[12] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *CoRR*, vol. abs/2001.04592, 2020.
[13] J. Schleier-Smith, "Serverless foundations for elastic database systems," in *CIDR*, 2019.
[14] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433, USENIX Association, July 2020.
[15] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 923–935, USENIX Association, July 2018.
[16] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 1063–1075, Association for Computing Machinery, 2019.
[17] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," *CoRR*, vol. abs/1810.09679, 2018.
[18] "Aws step functions payload size," Sept. 2020 [Online].
[19] "Azure cross function communication," 2020 [Online].
[20] "Sintel, the durian open movie project."
[21] "Cloudstate: Distributed state management for serverless."
[22] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 205–218, USENIX Association, July 2020.
[23] J. R. Gunasekaran, P. Thinakaran, N. C. Nachiappan, M. T. Kandemir, and C. R. Das, "Fifer: Tackling resource underutilization in the serverless era," in *Proceedings of the 21st International Middleware Conference*, Middleware '20, (New York, NY, USA), p. 280–295, Association for Computing Machinery, 2020.
[24] "Nimbella: Kubernetes based serverless platform."
[25] "Stackery: Secure delivery of serverless applications."
[26] "Iron.io: Hosted serverless tools."
[27] "The serverless application framework."
[28] T. Zhang, D. Xie, F. Li, and R. Stutsman, "Narrowing the Gap Between Serverless and its State with Storage Functions," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC'19)"*, SoCC'19, 2019.
[29] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 427–444, USENIX Association, Oct. 2018.
[30] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, (New York, NY, USA), p. 152–166, Association for Computing Machinery, 2021.
[31] 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)*, (Boston, MA), pp. 193–206, USENIX Association, Feb. 2019.
[32] J. Spillner, C. Mateos, and D. A. Monge, "Faaster, better, cheaper: The prospect of serverless scientific computing and hpc," in *CARLA*, 2017.
[33] P. Vaziri and K. Vora, "Controlling memory footprint of stateful streaming graph processing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 269–283, USENIX Association, July 2021.
[34] S. Kotni, A. A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, July 2021.
[35] 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*, Middleware '19, (New York, NY, USA), p. 41–54, Association for Computing Machinery, 2019.
[36] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica, "Caerus: NIMBLE task scheduling for serverless analytics," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 653–669, USENIX Association, Apr. 2021.
[37] 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*, SoCC '19, p. 13–24, Association for Computing Machinery, 2019.

Authorized licensed use limited to: ECOLE POLYTECHNIQUE DE MONTREAL. Downloaded on March 17,2022 at 14:39:26 UTC from IEEE Xplore. Restrictions apply.