# SONIC: Application-aware Data Passing for Chained Serverless Applications

Ashraf Mahgoub
*Purdue University*

Karthick Shankar
*Carnegie Mellon University*

Subrata Mitra
*Adobe Research*

Ana Klimovic
*ETH Zurich*

Somali Chaterji
*Purdue University*

Saurabh Bagchi
*Purdue University*

## Abstract

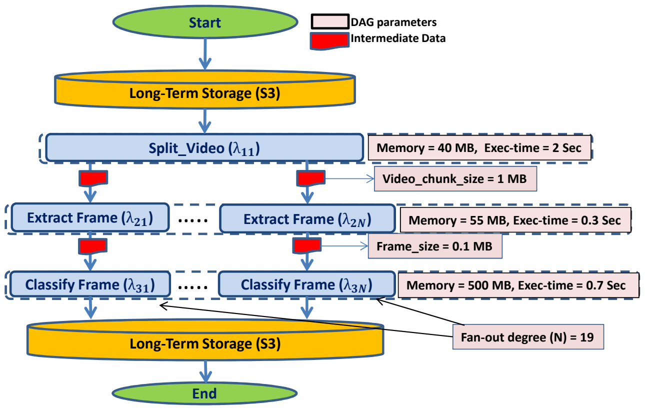
Data analytics applications are increasingly leveraging serverless execution environments for their ease-of-use and pay-as-you-go billing. The structure of such applications is usually composed of multiple functions that are chained together to form a workflow. The current approach of exchanging intermediate (ephemeral) data between functions is through a remote storage (such as S3), which introduces significant performance overhead. We compare three data-passing methods, which we call *VM-Storage*, *Direct-Passing*, and state-of-practice *Remote-Storage*. Crucially, we show that no single data-passing method prevails under all scenarios and the optimal choice depends on dynamic factors such as the size of input data, the size of intermediate data, the application's degree of parallelism, and network bandwidth. We propose SONIC, a data-passing manager that optimizes application performance and cost, by transparently selecting the optimal data-passing method for each edge of a serverless workflow DAG and implementing communication-aware function placement. SONIC monitors application parameters and uses simple regression models to adapt its hybrid data passing accordingly. We integrate SONIC with OpenLambda and evaluate the system on Amazon EC2 with three analytics applications, popular in the serverless environment. SONIC provides lower latency (raw performance) and higher performance/$ across diverse conditions, compared to four baselines: SAND, vanilla OpenLambda, OpenLambda with Pocket, and AWS Lambda.

## 1 Introduction

Serverless computing platforms provide on-demand scalability and fine-grained resource allocation. In this computing model, cloud providers run the servers and manage all administrative tasks (e.g., scaling, capacity planning, etc.), while users focus on the application logic. Due to its elasticity and ease-of-use advantages, serverless computing is becoming increasingly popular for advanced workflows such as data processing pipelines [35,53], machine learning pipelines [11,55], and video analytics [5,22,64]. Major cloud providers recently introduced serverless workflow services such as AWS Step Functions [4], Azure Durable Functions [45], and Google Cloud Composer [24], which provide easier design and orchestration for serverless workflow applications. Serverless workflows are composed of a sequence of execution stages, which can be represented as a directed acyclic graph (DAG) [17,53]. DAG nodes correspond to serverless functions (or λs[1]) and edges represent the flow of data between dependent λs (e.g., our video analytics application DAG is shown in Fig. 1).

Exchanging intermediate data between serverless functions is a major challenge in serverless workflows [34,36,47]. By design, IP addresses and port numbers of individual λs are not exposed to users, making direct point-to-point communication difficult. Moreover, serverless platforms provide no guarantees for the overlap in time between the executions of the parent (sending) and child (receiving) functions. Hence, the state-of-practice technique for data passing between serverless functions is saving and loading data through remote storage (e.g., S3). Although passing intermediate data through remote storage has the benefit of cleanly separating compute and storage resources, it adds significant performance overheads, especially for data-intensive applications [47]. For example, Pu *et al.* [53] show that running the CloudSort benchmark with 100TB of data on AWS Lambda with S3 remote storage can be up to 500× slower than running on a cluster of VMs. Our own experiment with a machine learning pipeline that has a simple linear workflow shows that passing data through remote storage takes over 75% of the computation time (Fig. 2, fanout = 1). Previous approaches reduce this overhead by implementing exchange operators optimized for object storage [47,52], replacing disk-based object storage (e.g., S3) with memory-based storage (e.g., ElastiCache Redis), or combining different storage media (e.g., DRAM, SSD, NVMe) to match application needs [12,37,53]. However, these approaches still require passing data over the network multiple times, adding latency. Moreover, in-memory storage services are much more expensive than disk-based stor-

[1]For simplicity, we denote a serverless function as lambda or λ.



Start
DAG parameters
Intermediate Data
Long-Term Storage (S3)
Split_Video ($\lambda_{11}$)
Memory = 40 MB, Exec-time = 2 Sec
Video_chunk_size = 1 MB
Extract Frame ($\lambda_{21}$)
Extract Frame ($\lambda_{2N}$)
Memory = 55 MB, Exec-time = 0.3 Sec
Frame_size = 0.1 MB
Classify Frame ($\lambda_{31}$)
Classify Frame ($\lambda_{3N}$)
Memory = 500 MB, Exec-time = 0.7 Sec
Fan-out degree (N) = 19
Long-Term Storage (S3)
End

*Figure 1: DAG overview (DAG definition provided by the user and our profiled DAG parameters) for Video Analytics application.*

age (e.g., ElastiCache Redis costs 700× more than S3 per GB [53]).

We show how cloud providers can optimize data exchange between chained functions in a serverless DAG workflow with communication-aware placement of lambdas. For instance, the cloud provider can leverage data locality by scheduling the sending and receiving functions on the same VM, while preserving local disk state between the two invocations. This data passing mechanism, which we refer to as *VM-Storage*, minimizes data exchange latency but imposes constraints on where the lambdas can be scheduled. Alternatively, the cloud provider can enable data exchange between functions on different VMs by directly copying intermediate data between the VMs that host the sending and receiving lambdas. This data passing mechanism, which we refer to as *Direct-Passing*, requires one copy of the intermediate data elements, serving as a middle ground between *VM-Storage* (which requires no data copying) and *Remote storage* (which requires two copies of the data). Each data passing mechanism provides a trade-off between latency, cost, scalability, and scheduling flexibility. Crucially, we find that no single mechanism prevails across all serverless applications, which have different data dependencies. For example, while *Direct-Passing* does not impose strict scheduling constraintsr of receiving functions copy data simultaneously and saturate the VM's outgoing network bandwidth. *Hence, we need a hybrid, fine-grained data passing approach that optimizes data passing for every edge of an application DAG.*

**Our solution**: We propose SONIC, a management layer for inter-lambda data exchange, which adopts a hybrid approach of the three data passing methods (*VM-Storage*, *Direct-Passing* and *Remote storage*) to optimize application performance and cost. SONIC exposes a unified API to application developers which selects the optimal data passing method for every edge in an application DAG to minimize data passing latency and cost. We show that this selection depends on parameters such as the size of input, the application's degree of parallelism, and VM network bandwidth. SONIC adapts

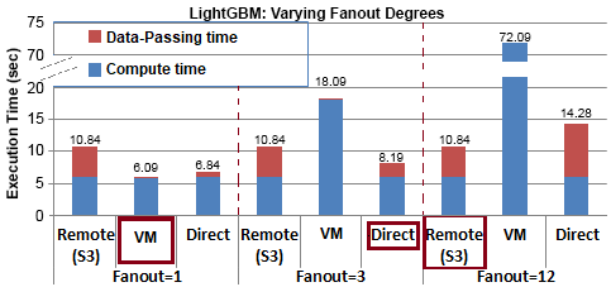


*Figure 2: Execution time comparison with Remote storage, VM storage, and Direct-Passing for the LightGBM application with Fanout = 1, 3, 12. The best data passing method differs in every case.*

its decision dynamically as these parameters change. Since locally optimizing data passing decisions at given stages in a DAG can be globally sub-optimal, SONIC applies a Viterbi-based algorithm to optimize latency and cost across the entire DAG. Fig. 3 shows the workflow of SONIC. SONIC abstracts its hybrid data passing selection and provides users with a simple file-based API, so users always read and write data as files to storage that appears local. SONIC is designed to be integrated with a cluster resource manager (e.g., Protean [29]) that assigns VM requests to the physical hardware and optimizes provider-centric metrics such as resource utilization and load balancing (Fig 4).

We integrate SONIC with the open-source OpenLambda [31] framework and compare performance to several baselines: AWS-Lambda, with S3 and ElastiCache-Redis (which can be taken to represent state-of-the-practice); SAND [2]; and OpenLambda with S3 and Pocket [37]. Our evaluation shows that SONIC outperforms all baselines for a variety of analytics applications. SONIC achieves between 34% and 158% higher performance/$ (here performance is the inverse of latency) over OpenLambda+S3, between 59% and 2.3X over OpenLambda+Pocket, and between 1.9× and 5.6× over SAND, a serverless platform that leverages data locality to minimize execution time.

In summary, our contributions are as follows:

**(1)** We analyze the trade-offs of three different intermediate data passing methods (Fig. 5) in serverless workflows and show that no single method prevails under all conditions in both latency and cost. This motivates the need for our hybrid and dynamic approach.

**(2)** We propose SONIC, which automatically selects data passing methods between any two serverless functions in a workflow to minimize latency and cost. SONIC dynamically adapts to application changes.

**(3)** We evaluate SONIC's sensitivity to serverless-specific challenges such as the cold-start problem (set-up time for the application's environment when it is invoked for the first time), the lack of point-to-point communication, and the provider's lack of knowledge of lambda input data content. SONIC shows its benefit over all baselines with three common classes of serverless applications, with different input sizes and user-specified parameters.

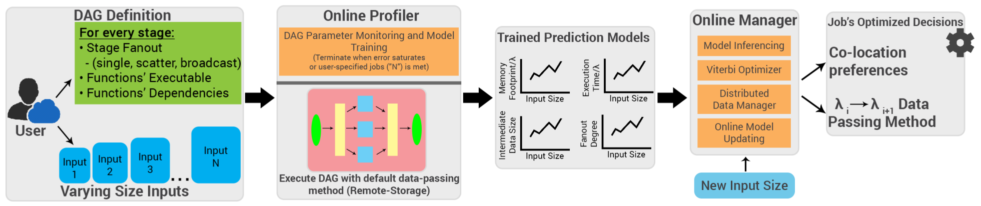


*Figure 3: Workflow of* SONIC*: Users provide a DAG definition and input data files for profiling.* SONIC*'s online profiler executes the DAG and generates regression models that map the input size to the DAG's parameters. For every new input, the online manager uses the regression models to identify the best placement for every λ and best data passing method for every pair of sending\receiving λs in the DAG*

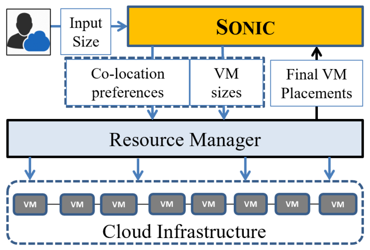


*Figure 4:* SONIC*'s interaction with an existing Resource Manager in the system.*

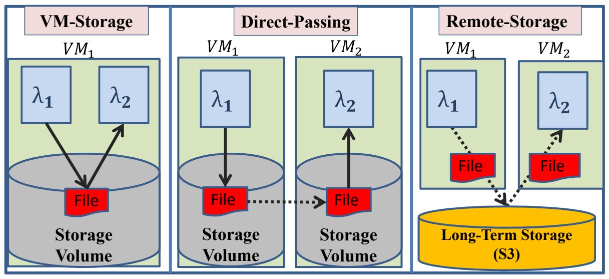


*Figure 5: The three data passing options between two lambdas ($\lambda_1$ and $\lambda_2$). VM-Storage forces $\lambda_2$ to run on the same VM as $\lambda_1$ and avoids copying data. Direct-Passing stores the output file of $\lambda_1$ in the source VM's storage, and then copies the file directly to the receiving VM's storage. Remote storage uploads and downloads data through a remote storage (e.g., S3).*

## 2 Rationale and Overview

We discuss the trade-offs of SONIC's three data passing methods and motivate our hybrid and dynamic approach.

### 2.1 data passing Methods

SONIC chooses from among the three data passing options shown in Fig. 5 for *each* edge in the application DAG.

***VM-Storage:*** This method saves the local state of the sending λ in the VM's storage and schedules the receiving λ(s) to execute on the same VM. This method leverages data locality to minimize latency, but imposes scheduling constraints. If the sending VM's memory cannot fit all receiving λs, this method forces the scheduler to run the receiving λs serially or in batches, sacrificing parallelism in favor of data locality. This method is infeasible if the memory requirement of a single receiving λ exceeds the VM's capacity, or when the receiving λ collects data from multiple λs running on different VMs. Moreover, this data passing method may not be preferred by the resource manager in high load scenarios, where spreading the receiving functions over many servers is needed to avoid hot-spots and achieve better load balancing.

***Direct-Passing:*** This data passing method saves the output of the sending λ on its VM storage and sends the λ's access information (IP address and File Path) to SONIC's metadata manager. When one of the receiving λs is scheduled to execute, the metadata manager uses the saved access information to copy the data file *directly* to the destination VM with the receiving λ. *Direct-Passing* allows higher degrees of parallelism and poses no restrictions on λ placements compared to *VM-Storage*, but requires data to be sent over the network between source and destination VMs.

***Remote-Storage:*** This data passing method involves uploading output files to a remote storage system and downloading them at destination λ(s). This is the state-of-practice in commercial serverless platforms and has been optimized in several recent papers [12, 37, 53]. This method provides high scalability with no restrictions on λ placement. It also has the advantage of almost uniform data passing time with increasing fanout degrees as shown in Fig. 6 due to the high bandwidth of the storage layer. The disadvantage of *Remote-Storage* is having two serial data copies in the critical path — one from the source lambda to the remote storage and one from the remote storage to the destination lambda.

### 2.2 Dynamic Data Passing Method Selection

The optimal choice of data passing method for a job depends on the DAG's parameters, which can vary due to dynamic conditions such as the input size, changes in network bandwidth, or changes in user-specified parameters (e.g., degree of parallelism). For example, we run the LightGBM application (application details are given in §5.3) with varying degrees of fanout and find that *VM-Storage* achieves optimal end-to-end execution time when the fanout is low (Fig. 2). However, when the maximum degree of parallelism across all stages is three, *VM-Storage* requires executing functions serially to fit on the same VM, making *Direct-Passing* superior. With a sufficiently high fanout, the sending VM faces a network bottleneck, making *Remote-Storage* the optimal data passing mechanism. Hence, no single data passing method prevails under all conditions.

SONIC prefers the *VM-Storage* method in cases where the receiving λ(s) can be scheduled on the same VM as the sending λ while executing in parallel. However, in cases where *VM-Storage* is infeasible or sacrifices parallelism, SONIC selects between *Direct-Passing* or *Remote-Storage* methods. This selection depends on two factors: (1) VM's network bandwidth (2) the fanout (i.e., parallelism) type and degree. To understand how these two parameters impact the selection between *Direct-Passing* and *Remote-Storage*, we show an experiment on the LightGBM application, which has a Broadcast Fanout stage (identical data from the sending λ is sent

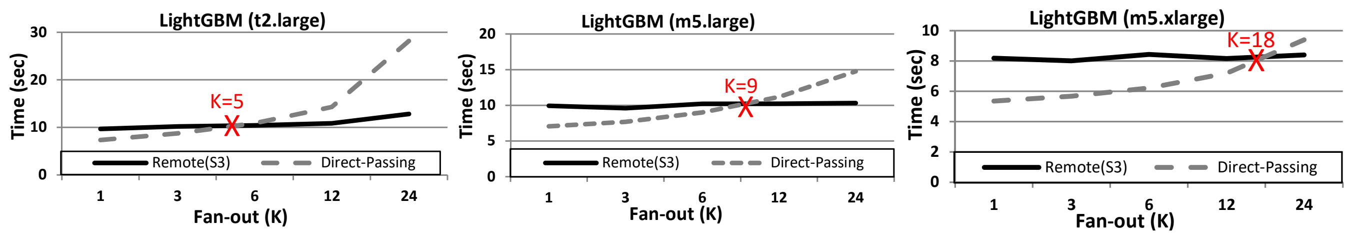


*Figure 6: Comparison between Remote-Storage and Direct-Passing for LightGBM workload with varying fanout degrees. Typically, beyond a certain fanout, Remote has lower execution time than Direct-Passing. A more well-provisioned VM (m5.xlarge) will shift the crossover point to the right.*

to all the receiving λs). We vary the fanout degree ($K$) while using different VM types with varying network bandwidths in Fig. 6. With low degrees of parallelism (i.e., low values of $K$), *Direct-Passing* achieves lower latency than *Remote-Storage*. This is because copying the data directly across the two VMs is faster than copying the data twice over the network, to and from the remote storage. However, with increasing values of $K$, *Direct-Passing* suffers from the limited network bandwidth of the VM of the sending λ, which becomes the bottleneck as it tries to copy the intermediate data to $K$ VMs simultaneously. In contrast, *Remote-Storage* can provide faster data passing in this case, since the sending λ saves its output file once to the remote storage and then every receiving λ downloads that file simultaneously. Thus, *Remote-Storage* provides better scalability over *Direct-Passing* in cases of large fanout degrees. The cross-over point shifts to the right as we go to more well-resourced VMs, from a network bandwidth standpoint (m5.xlarge > m5.large > t2.large). From this example, we see that selecting the best data passing method depends on the VM's network bandwidth *and* DAG parameters.

We identify a number of trade-offs to be considered when performing this optimization. *First*, a trade-off between data locality and parallelism arises when the available VM's compute and memory capacities are not sufficient to execute all receiving λ(s) in parallel. This preference of data locality over parallelism can be beneficial for lightweight functions communicating large volumes of data, while it can be harmful for compute-heavy functions with small volumes of data. *Second*, executing functions serially has the benefit of avoiding cold-start executions, which can significantly increase the execution time for functions that need to fetch many dependencies before execution [58]. *Finally*, we differentiate between two types of fanout stages: Scatter and Broadcast, depending on whether the output data is split equally among all outgoing edges (Scatter) or the same data is sent on all outgoing edges (Broadcast). With Scatter fanout, the intermediate data volume being sent is constant with the fanout degree while with Broadcast fanout, the intermediate data volume being sent increases linearly with the fanout degree. Therefore, the optimal data passing method also depends on the type of fanout and thus we cover both in our evaluation applications (video analytics and MapReduce sort for Scatter fanout and LightGBM for Broadcast fanout). In the next section, we describe the design of each component in SONIC.

## 3 Design

In § 3.1, we provide an overview of SONIC's usage model. § 3.2 describes SONIC's online profiling. § 3.3 discusses how SONIC estimates a job's execution time for different input data sizes. § 3.4 shows how SONIC selects globally optimized data passing decisions. Finally, § 3.5 highlights further design considerations.

### 3.1 Usage Model

We discuss SONIC's usage model from the perspective of application developers and the cloud provider's resource manager.

**Application DAG:** As shown in Fig. 3, application developers provide SONIC with an application represented as a DAG. We assume users execute application DAGs multiple times with potentially different input data. Each such execution instance is referred to as a *job*. SONIC does not require the FaaS provider to have access to the source code for the serverless functions or hints about the resource utilization of these functions. The DAG definition from the user includes: (1) an executable for every λ; (2) the dependencies between λs; (3) the fanout type in every stage (i.e., single, scatter, or broadcast). Users can also provide an upper limit on the execution time or $ budget for a single job or a batch of jobs. Notice, SONIC does not require users to specify the memory requirements for the functions in the DAG (this is a well-known pain point for users of serverless frameworks [20, 54, 57]). This is because SONIC predicts the memory footprint for every function, using our simple regression modeling (§ 3.2) and selects the right host VM size accordingly.

**Data Passing Interface**: SONIC abstracts the selection of data passing methods from application developers. λ functions write intermediate data to files using a standard file API (read and write), like writing to local storage. All λs within a job share a file namespace and if an application DAG has an edge $\lambda_s \rightarrow \lambda_r$, SONIC ensures that $\lambda_r$ reads from the same file path that $\lambda_s$ wrote to. It also ensures that all of a $\lambda_r$'s input files are present in its local storage before it starts execution.

**Resource Manager:** SONIC's target is to minimize communication latency and cost, which are *user-centric* metrics. However, optimizing *provider-centric* metrics (e.g., load-balancing, resource utilization, and fairness) is also important. Fortunately, many resource management systems such as Graphene [25], Protean [29], AlloX [38] and DRF [23] are designed to optimize *provider-centric* metrics efficiently, while respecting the dependencies between the functions (i.e., parent-child execution order). Hence, SONIC is designed to integrate with a system from this category (which we refer to as *Resource Manager*), as shown in Figure 4. First, the user executes the DAG with a new input. Second, SONIC uses the input size information to predict the memory footprints, execution times, and intermediate-data sizes for the DAG, which are key DAG parameters in selecting the best VM size for every function and the best data passing method for every edge in the DAG. Finally, the Resource Manager uses SONIC's hints and decides which VMs to allocate on the available physical machines, and responds to SONIC with the final placement information, i.e., which functions go on which VMs. The Resource Manager may override SONIC's hint to use *VM-Storage* whenever its scheduling constraint is not acceptable to the manager. Therefore, for that selection, SONIC always proposes the second-best option.

## 3.2 Online Profiling and Model Training

SONIC profiles jobs *online* to determine the impact of changes in input size to the following parameters: (1) each λ's memory footprint, (2) each λ's compute time, (3) intermediate data volume between any two communicating λs, and (4)the fanout type and degree in every stage. Since the above parameters are *not* dependent on the data passing method, initially SONIC uses a default data passing policy (*Remote Storage*) while it collects DAG parameters for the first $N$ runs of the job to train its models. $N$ is either the number of jobs needed to reach convergence (default) or explicitly set by the user. Next, SONIC trains a set of prediction models that estimate the DAG parameter values for new inputs. SONIC develops polynomial regression models and splits the data collected into training and validation sets, then performs 5-fold cross validation to find the best model to avoid overfitting. We use polynomial regression models as they can learn from limited data points, are lightweight, and are interpretable [8].

"Online" profiling means that SONIC serves workloads while the models are being trained, which is important for practical adoption in production environments. In discussions with commercial cloud providers, we have repeatedly sensed an anathema to solutions that require offline training due to the concern that one will constantly be taking the system offline to (re-)train the models. However, if the system owner has ready access to representative input traces, she can feed SONIC with these representative traces *offline* to initialize the prediction models and reduce the online training burden.

SONIC measures the compute time for every λ under two conditions: cold execution (i.e., a new VM/container needs to be created) and hot execution (i.e., a warm VM/container with pre-loaded models/dependencies already exists). SONIC uses the difference between the two execution times in deciding whether to queue λs on the same VM and sacrifice parallelism (hot execute), or to execute the λs on different VMs in parallel with the additional data passing cost and startup latency (cold execution).

## 3.3 Minimizing End-to-End Execution Time

We estimate the execution time τ of a stage $S_i$ as follows:

$$\tau(S_i) = DataPass(S_{i-1}, S_i) + Compute(S_i) \quad (1)$$

When *VM-Storage* is selected, all λs in a given stage are forced to run on the same VM. If only one λ can run at a time, the first λ incurs a cold execution time and all subsequent λs experience hot executions. SONIC estimates $\tau_{VM}$ as follows:

$$\tau_{VM}(S_i) = 0 + Cold(\lambda_{i,1}) + \sum_{j=2}^{K} Hot(\lambda_{i,j}) \quad (2)$$

Here we set $DataPass(S_{i-1}, S_i)$ to zero since no additional latency is incurred by *VM-Storage* passing. For simplicity, we give here the upper bound, if all the λs are serialized. In practice, we estimate based on batches that are serialized and all λs within a batch can run concurrently. For *Direct-Passing* and *Remote Storage*, we take the nature of the fanout type into account. For *Direct-Passing*, with Broadcast-type fanout, the runtime is given by:

$$\tau_{Direct_B}(S_i) = \frac{F \times K}{min(BW(VM_{i-1}), BW(VM_i))} + Cold(\lambda_{i,1}) \quad (3)$$

Where $F$ is the intermediate data size, $K$ the fanout, and $BW(VM_i)$, the bandwidth of the VM type hosting λs in the $i$-th stage. Notice that the bandwidth is limited by the slowest of the sending and receiving VMs and that all execution times are *cold*, yet only one execution is accounted for, since they all run in parallel. For Scatter-type fanout, the equation becomes:

$$\begin{aligned}\tau_{Direct_S}(S_i) &= \frac{F/K}{min(BW(VM_{i-1})/K, BW(VM_i))} + Cold(\lambda_{i,1}) \\ &= \frac{F}{BW(VM_{i-1})} + Cold(\lambda_{i,1}) \quad (4)\end{aligned}$$

Often cloud providers overprovision network bandwidth [14], hence we assume location of the source and destination VMs (intra- vs. inter-rack) does not affect the network bandwidth. Also, we find that the bandwidths are symmetric between VMs for all VM types; however, for remote storage, writing was faster than reading. Next, we show how we use the previous equations for our optimization.

## 3.4 Online VM and Data Passing Selection

A major challenge to optimize the Perf/$ for the entire DAG is to do local selections for *each* stage to achieve the global optimum. Consider Fig. 1 for example: if we used *VM-Storage* passing between `Split_Video` and `Extract Frame`, all the extracted frames will reside in a single VM. Selecting *VM-Storage* between `Extract Frame` and `Classify Frame` will force `Classify Frame` to execute serially since that single VM does not have enough memory. However, if we select *Direct-Passing* or *Remote Storage* passing between

`Split_Video` and `Extract Frame`, every extracted frame will now reside in a separate VM. Therefore, we can use *VM-Storage* between `Extract Frame` and `Classify Frame` without sacrificing parallelism, lowering the DAG's execution time. Thus, greedily optimized decisions for individual stages can lead to sub-optimal global DAG performance.

To overcome the issue of sub-optimal greedy decisions, we apply the Viterbi algorithm [18] to find the globally optimized solution. SONIC uses a recursive scoring algorithm to generate all possible lambda assignments in every stage in the DAG based on the VM's compute and memory capacities, along with the λs' predicted memory footprint. SONIC explores all possible λ-placement options under the constraints of VM resources (CPU and memory primarily) and selects the best data passing method for every pair of stages. Then, SONIC constructs a dynamic programming table with all the generated solutions. Finally, SONIC applies the Viterbi Algorithm to find the best sequence of options (i.e., the optimum Viterbi path) in the dynamic table. The selected solution is the one with the best Perf/$ that also meets the user bounds on execution time and cost budget. This approach relies on the fact that the execution time up until stage $i$ is equal to the execution time to stage $i-1$ + data passing time between the two stages. This makes the Markovian assumption true (next state depends only on the current state) and makes the computation tractable.

We choose the Viterbi algorithm as it is guaranteed to find the true maximum a posteriori (MAP) solution [18, 19] unlike heuristic-based searching algorithms such as Genetic Algorithms or Simulated Annealing. The runtime complexity of the algorithm is $O(P^2 \times S)$, where $P$ is the number of feasible λ placements on VMs for a given stage and $S$ is the number of stages. $P$ is upper bounded by the degree of parallelism of the stage (e.g., AWS sets a limit of 1,000) and in practice is much smaller considering many co-locations on VMs are infeasible. The runtime increases with the number of stages in the DAG, not the number of nodes or edges or fanout degree. This is desirable as the number of stages is small in practice, where a DAG of 8 stages is considered long for current serverless applications [53]. This reduction in complexity happens because SONIC applies the *same* data passing method to *all* the functions in a given stage.

## 3.5 Further Design Considerations

**Fault tolerance**. Most FaaS providers apply an automatic retry mechanism upon execution failure (e.g., AWS Lambda, Google Functions, and Azure Functions) to ensure that functions are executed *at-least-once*. For this retry mechanism to be successful, functions are required to be idempotent. To achieve idempotence, SONIC's file API assigns an ID to every intermediate file in the DAG that is used by the function that writes that file. Hence, a re-execution of this function simply overwrites the files from the previous execution. However, as highlighted in RAMP [7] and AFT [60], idempotence in itself is not sufficient to achieve fault tolerance in serverless environments, since it does not guarantee *atomic visibility*. The problem happens when a sender λ generates only a subset of its output files, and then fails, triggering an incomplete subset of receiving λs, resulting in a corrupted state. Therefore, SONIC applies the concept of *atomic visibility* by delaying the execution of *all* receiving λs that belong to the same logical request until *all* their input files are successfully written to storage (either EBS in case of *Local-VM or Direct-Passing*, or S3 in case of *Remote Storage*). Although this delaying mechanism can potentially increase the E2E latency of the DAG, our evaluation shows that this additional latency is negligible: 6.3% for MapReduce, 3.3% for Video-Analytics, and 0.5% for LightGBM. All the SONIC results in the evaluation are with atomicity enabled.

**DAGs with Content-Sensitive Structures**. Our design assumes that the stages in the DAG are static and known, while the fanout degree in every stage can vary based on the input. This is analogous to state machines created by serverless orchestrators such as AWS Step-Functions [4]. Our video-analytics application is an example of an input-dependent fanout degree DAG and SONIC successfully predicts the fanout for new input sizes. If the structure of the DAG's stages depends on the input content, then our estimate of the DAG will be inaccurate. We evaluate the sensitivity of Sonic to prediction errors in Sec. 5.6.3.

**Scalability of** SONIC. For scalability, SONIC adopts a simple distributed design in which no state is shared across application jobs. When a new job arrives, SONIC's centralized component makes the decision whether to use an existing instance (if it is not overloaded) or to spin up a new instance to handle the new job. There is a central scoreboard maintained to keep track of the number of jobs being handled by each instance. The central component is lightweight in terms of its computational load and state. However, should it be necessary, this itself can be distributed through standard state machine replication (SMR) strategies [44].

## 4 Implementation

We implement SONIC as a data passing management layer and we use OpenLambda as our serverless platform [31]. OpenLambda is an open-source platform that relies on Linux containers for isolation and orchestration [51]. We choose OpenLambda for its flexibility—SONIC needs to control the lambda placement and needs IP addresses and administrative access to the hosting VMs. This level of control is infeasible to achieve on AWS Lambda or any other commercial offering. We implement SONIC in C# (482 LOC) and deploy it on EC2 instances, providing the same isolation guarantees as AWS Lambda across users [6]. In our setup of SONIC on OpenLambda, we use one separate container for each function (OpenLambda's design), while one VM can host multiple containers for the same application.

SONIC's data manager consists of two parts: a centralized manager that stores IP addresses and file paths, and a distributed manager, deployed in every VM, executing the SONIC-optimized data passing method. The distributed manager also measures the actual DAG parameters during the online phase and sends them to the online manager, which updates the regression models in an incremental fashion. Moreover, since the network bandwidth can vary over time, we monitor it using Cloudwatch [13] (default of every 5 min). We use a weighted average of historic measures (as is common [56, 63]) when estimating data passing times. Thus SONIC adjusts its decisions based on bandwidth fluctuations. We use EBS storage as our storage for *VM-Storage* and *Direct-Passing* methods. EC2 EBS-optimized instances (e.g., our choice m5) have dedicated bandwidth for EBS I/O (minimizes network contention with other traffic), and can be rightsized for the predicted intermediate data volume.

## 5 Evaluation

### 5.1 Performance Metrics

Our primary performance metric is Perf/\$. Since we are minimizing end-to-end (E2E) execution time (i.e., latency), this is given by: $\frac{1}{Latency(sec)} \times \frac{1}{Price(\$)}$. We also use raw latency as a secondary metric, to separate the \$-cost normalization effect. For the first metric, higher is better, and for the second, lower is better. When we refer simply to performance, we mean Perf/\$. When we refer to the secondary metric, we explicitly say (E2E) execution time or latency.

The different data passing methods considered by SONIC vary in their billing cost, which is sensitive to the selected configurations for each method. *Hence, our solution should consider both latency **and** cost when selecting the best data passing method.* Consequently, we use the Perf/\$ metric. We also empirically demonstrate that SONIC performs comparably to the baselines in terms of raw performance, and in many cases, outperforms the baselines (Figures 7, 8, 13, 14). Finally, there is precedence of prior work in cloud optimization using performance normalized by price [32, 40, 62]. SONIC's *VM-Storage* and *Direct-Passing* methods add additional cost due to the extra local storage (e.g., EBS storage). However, this additional price is comparable to that of remote storage such as S3, and we include it in our evaluation.

### 5.2 Baselines and Methodology

We compare SONIC to the following baselines:
1. **OpenLambda + S3** [31]: This is the OpenLambda framework deployed on EC2 with S3 as its remote storage. A new VM is created to host each λ in the DAG. The smallest VM that has enough memory to execute the λ is selected.
2. **OpenLambda + Pocket** [37]: This is a variant of the OpenLambda framework with Pocket (deployed in EC2) as the remote storage. We use Pocket's default storage tier (DRAM) with *r5.large* instance types. The DRAM storage tier strikes the balance between performance and cost and provides the best Perf/\$ (Table 4 in [37]). Comparing the price for one DRAM node to one SSD node in Pocket, DRAM is also the cheapest tier. We vary the number of Pocket nodes for each application to reach the best Perf/\$. We include the price of the storage nodes only, excluding master and metadata nodes. We use EC2's per-second level pricing.
3. **SAND** [2]: This baseline leverages data locality by allocating all lambda functions on a single host with rich resources and performing data passing between chained functions through a local message bus.
4. **AWS-λ**: The commercial FaaS platform using two different remote storage systems: S3 and ElastiCache-Redis.
5. **Oracle** SONIC: This is SONIC with fully accurate estimation of DAG parameters and no data passing latency (mimicking local running of all functions). Although impractical, this serves as an upper-bound on performance for any of the three data-exchange techniques selected by SONIC.

Similar to prior OpenLambda evaluations [50], we deploy OpenLambda (and SONIC) on AWS EC2 General Purpose instances (*m5.large*, *m5.xlarge*, *m5.2xlarge*) for their balance between CPU, memory, and network bandwidth. We use the same EC2 family with SAND for fairness. *m5* instances have a network bandwidth of upto 10 Gbps, which we rely on for both *Direct-Passing* and *Remote-Storage*. All costs follow pricing by Amazon for 01/2021, N. California (Region).

### 5.3 Applications

We use three analytics applications popular as serverless applications and that span the variety of DAG structures.

**Video Analytics:** Fig. 1 shows the DAG for the Video Analytics application, which performs object classification for frames in a given video. It starts with a lambda that splits the input video into chunks of fixed length (10 sec in our case). Then, a second lambda is called for every chunk to extract a representative frame. Next, a third lambda uses a pre-trained deep learning model (MXNET [49]) to classify the extracted frame. It outputs a probability distribution across 1,000 classes over which MXNET is trained. Finally, all the classification results are written to long-term storage.

**LightGBM:** This application trains decision trees, combining them to form a random forest predictor using `LightGBM` Python library [39]). First, a λ reads the training examples and performs PCA. Second, a user-specified number of λs train the decision trees in parallel (every λ randomly selects 90% for training, 10% for validation). A third λ collects and combines the trained models; then tests the combined model on held-out test data. Handwritten images' databases: NIST (800K images); MNIST (60K images) used as inputs [15, 26].

**MapReduce Sort:** This application implements MapReduce sort with serverless functions. In the first stage, $K$ parallel lambdas (i.e., mappers; $K$ = user parameter) fetch the input

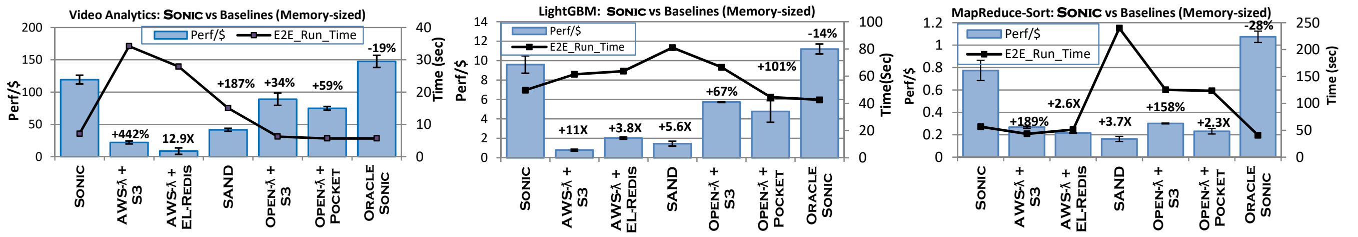


*Figure 7: Performance of* SONIC *and the baselines for our three applications (memory-sized). Two performance metrics are shown: Perf/$ (bars; left axis) and end-to-end execution time (lines; right axis). Relative improvements in Perf/$ due to* SONIC *are at the top of each bar for the corresponding baseline.*

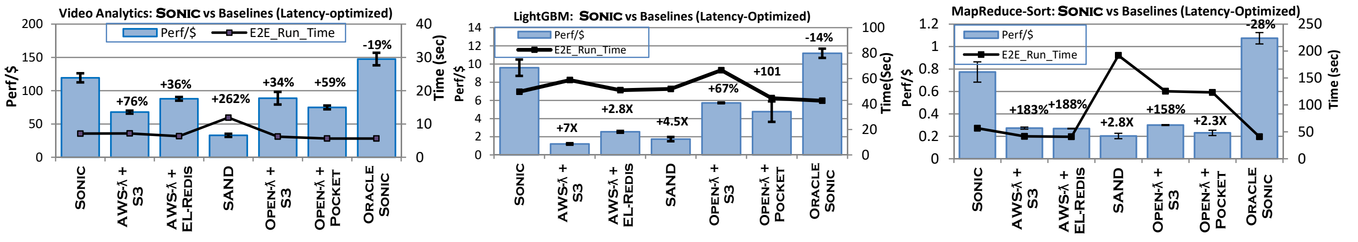


*Figure 8: Performance of* SONIC *and the baselines for our three applications with the latency-optimized configuration.*

file from a remote data store (e.g., S3) and then generate the intermediate files. Next, $K$ parallel lambdas sort the intermediate files and write their sorted output back to storage.

## 5.4 End-to-End Evaluation

We compare the E2E Perf/$ and latency of SONIC vs. other baselines for our three applications. For Video Analytics, we use a video of length 3 min and size of 15MB, which generates a fanout degree of 19 (we vary the video size in § 5.6.3). For LightGBM, we use an input size of 200MB and fanout degree of 6, generating a random forest of 6 trees (we vary the input size in § 5.5.2). For MapReduce Sort, we use an input size of 1.5GB and a fanout degree of 30. All the results of SONIC include its overheads (11ms for the DAG parameter inference phase; 120ms for the Viterbi optimization phase).

We perform online profiling and training with 35 jobs, by which we reach convergence for all applications with a low average Mean Absolute Percentage Error (MAPE) $\leq 15\%$. We show the accuracy of SONIC's predictions in Section. 5.6.1.

**Memory configurations**. SONIC infers memory requirements automatically from online profiling. Here, we describe how we select the memory allocation for each baseline. AWS Lambda and other serverless offerings scale compute resources relative to a user-specified memory allocation. We run the baselines under two different configurations, which we call "memory-sized" and "latency-optimized". These are respectively shown in Fig. 7 and Fig. 8. For the memory-sized configuration, we give each lambda just enough memory that it needs to execute. We determine each λ's memory requirement by measuring the actual memory used when executing the DAG once with all λs using the maximum memory limit (3GB for AWS-λ). For the latency-optimized configuration, we progressively increase the memory allocation as long as a reduction in latency is observed. We report the results for the run with the lowest latency. For OpenLambda, we use SONIC's memory footprint predictor to select the cheapest VM that fits each lambda.

We draw several conclusions. First, although AWS-λ allows users to rightsize the allocated memory for their applications, it does not always lead to the best Perf/$ or latency. For example, with Video Analytics and memory-sized configuration, SONIC achieves 442% and 12.9× better Perf/$ over AWS-λ with S3 and ElastiCache-Redis respectively. However, with (memory) over-provisioning, the latency-optimized configuration improves AWS-λ performance significantly, reducing the gains of SONIC to 76% and 36% with S3 and ElastiCache-Redis respectively (similar observation is shown w.r.t. raw latency). The reason is that AWS-λ allocates all other resources (e.g., CPU capacity, network bandwidth, etc.) proportionally to the selected memory requirement [3]. Therefore, as the allocated memory is increased, the latency decreases and the cost also increases. But the latency decreases faster than the cost increases, thus Perf/$ increases. However, beyond a certain point of over-provisioning, the latency does not decrease further, and thus the Perf/$ begins to decrease.

For LightGBM and MapReduce Sort applications, we do not see a significant improvement in Perf/$ for AWS-λ baselines with the latency-optimized configuration, since the memory footprint of these applications is close to the 3GB limit to begin with leaving very little room for over-provisioning. For AWS-λ, using ElastiCache-Redis as the remote storage achieves 18% lower latency than using S3. However, ElastiCache-Redis increases the cost significantly, causing a *reduction* of Perf/$.

Compared to SAND, SONIC achieves 187% better Perf/$ with 2× lower latency in the memory-sized case for Video Analytics application. The gain increases to 5.6× and 3.7× for LightGBM and MapReduce Sort applications, respectively. This is again due to the higher memory footprints of these two applications compared to Video Analytics. This reduces SAND's ability to run more λs in parallel and forces a high

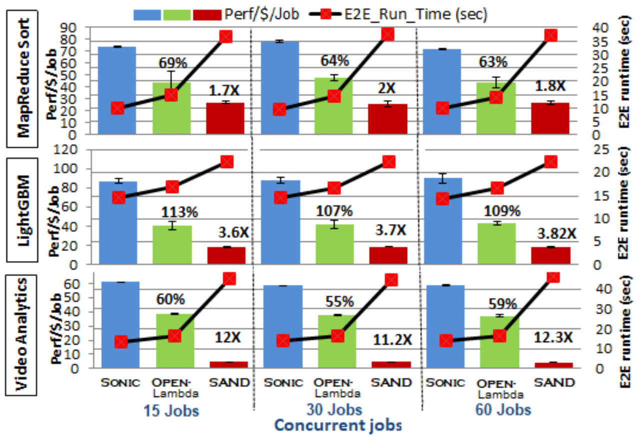


*Figure 9: Scalability of* SONIC*, OpenLambda+S3, and SAND with equal portions of our three apps running concurrently.* SONIC *maintains its improvement over the entire scale, in both Perf/\$/job and raw latency.*

degree of serialization. This explains the spike in latency for SAND in both applications.

Compared to OpenLambda, SONIC achieves 34% and 59% improvement in Perf/\$ with S3 and Pocket, respectively, for Video Analytics. The performance with Pocket suffers compared to vanilla OpenLambda (i.e., with S3) due to Pocket's higher-cost storage (e.g., r5.large) without proportional benefit. Note that for OpenLambda, its performance turns out to be identical for the memory-sized and latency-optimized configurations as increasing the memory beyond SONIC's predicted values for each function gives no latency benefit. Finally, Oracle-SONIC outperforms SONIC, as expected, but not hugely — within 19%-28% across all applications. Recall that Oracle-SONIC has perfectly accurate predictors and assumes no data passing latency.

## 5.5 Scalability

Here, we evaluate SONIC's ability to scale to concurrent invocations of a mix of the applications and larger data sizes.

### 5.5.1 Varying Degree of Concurrency

We compare SONIC to SAND and OpenLambda+S3 serving a mixture workload of our three applications, with equal portions of invocations (jobs) per application. We use a cluster of 52 VMs of type *m5.large* with 2 compute cores per VM. We select SAND as it always prefers to keep data local, while OpenLambda+S3 always uses *Remote-Storage* data passing. OpenLambda+S3 is also the closest baseline to SONIC in terms of performance (Fig. 7 and 8). We vary the level of concurrent invocations from 15 (5 per app) to 60 (20 per app). With 60 concurrent app invocations, the cluster executes a total of 480 functions in parallel and all compute cores are fully utilized (except for SAND). We show the Perf/\$/job and E2E runtime for every app in Figure 9[2]. We notice that the gain of SONIC over both baselines is consistent across the different degrees of concurrency, which shows SONIC's ability to seamlessly scale with the number of concurrent invocations. It is of course not surprising that the VM infrastructure scales up — the question was would SONIC scale up as well. This experiment answers that question in the affirmative (within the scales of the experiment), for SONIC as well as for the two baselines. This is expected from the design of SONIC where most of its components are stateless and different instances are spun up to handle more input jobs (Section 3.5). Since SAND schedules the entire job to execute on a single VM, it cannot utilize all the compute cores and hence suffers from very high latency. In contrast, OpenLambda uses *Remote-Storage* passing between functions, which allows scheduling functions on different VMs and utilizing the available compute resources. SONIC's hybrid approach achieves both lower E2E execution time along with high resource utilization, and therefore, achieves better Perf/\$ and raw latency than all baselines.

### 5.5.2 Varying Intermediate Data Size

In Fig. 12, we show the impact of changing the intermediate data size on SONIC's performance vis-à-vis two baselines. SONIC achieves lower E2E latency and higher Perf/\$ across all input sizes by predicting the corresponding DAG parameters and selecting the best co-location of lambdas and data passing methods. Second, the normalized Perf/\$ of SONIC and OpenLambda+S3 increases with higher input sizes. The reason is that the compute:data passing time ratio for this application increases with higher input sizes. This, in turn, is because the data passing time increases linearly with the input size, whereas its computation time grows faster than linear (PCA has quadratic compute complexity [65] and it dominates the total computation time). Recall that the Oracle assumes no data passing latency but it counts computation time. Accordingly, the higher the compute:data passing time ratio, the lower the gap between Oracle and the baselines (except SAND that has no data passing component).

## 5.6 Microbenchmarks

Here we evaluate SONIC's online training accuracy, sensitivity to input content, prediction noise, and cold-start times.

### 5.6.1 Prediction Accuracy with Online Refinement

As discussed in Sec. 3.2, our solution profiles jobs to characterize the relation between input size and the data passing relevant parameters. This training phase is done online while serving production jobs, and we use remote storage data passing (SONIC's default) until convergence is achieved. We show SONIC's accuracy in prediction of execution parameters for new input sizes across the three applications. First, we execute the DAG of each application with jobs of varying input sizes while we measure the DAG's execution parameters (i.e., $\lambda$'s

[2]We normalize by the number of jobs because naturally the total cost increases with the number of jobs completed and the normalization brings out the important trend that the metric is flat across the scales, implying perfect scalability.

memory footprint, λ's compute time, data volume on every edge, and fanout degree in every stage). Next, we divide the collected data into train and test sets. We fix the test set size while we vary the number of jobs used for training to show the reduction in error w.r.t. training with more jobs for every application. In each training run, we generate the regression models for all execution parameters in the DAG. For example, PCA's runtime regression estimated equation is: $Y = 0.0024\ X^2 + 12.764$, and PCA's estimated memory equation is: $Y = 16\ X + 185.5$, where $X$ is the input size in MBs .

For Video Analytics, we collect 60 YouTube videos with lengths that vary uniformly between 1 min and 1 hour and belong to 5 different categories with equal representation of each: News, Entertainment, Nature, Sports, and Cartoon. We similarly execute the LightGBM and MapReduce Sort applications with 60 jobs each. For LightGBM, we use the MNIST database of handwritten digits [15]. The data has 60,000 training images and 10,000 test images. To execute the DAG with varying input sizes, we sub-sample the training data and vary the sampling rate between 5% and 100% uniformly. For MapReduce Sort, we generate randomly shuffled data and vary the size of the data between 3M records (255MB) and 12M records (1.5GB). We also vary the number of Mappers and Reducers uniformly between 5 and 50.

Increasing the number of jobs used in training expectedly reduces the prediction error for all applications. With 35 jobs we reach convergence for all predictions with a low average Mean Absolute Percentage Error (MAPE) $\leq 15\%$ as shown in Figures 10 for Video-Analytics. This low prediction error is essential for SONIC to determine the best lambda placement information and data passing decisions for new jobs with new input sizes. Optionally, users can set a higher level of acceptable error to reduce the number of training jobs.

We notice that Video Analytics incurs the highest prediction error among the three applications. This is because our collected videos vary significantly in their bitrate, which impacts the relation between the input size and the video's length. Recall that SONIC is content-agnostic, it does not consider any information that is dependent on the content of the data when it makes its prediction. Although this negatively impacts the prediction accuracy for content-dependent applications, it allows SONIC to generalize to a wide range of applications without the need for special processing for each type of application. Furthermore, policies for public cloud providers often prohibit any visibility into the client applications. We evaluate SONIC's sensitivity to input content in § 5.6.3.

### 5.6.2 SONIC's Performance Improvement Root Causes

Here we highlight the root causes for SONIC's improved performance over baselines SAND and OpenLambda + S3. We show an example DAG for LightGBM application along with the data passing and lambda placement decisions made by each approach in Figure 11. We also show SONIC's selected optimum Viterbi path in the table at the bottom. We run

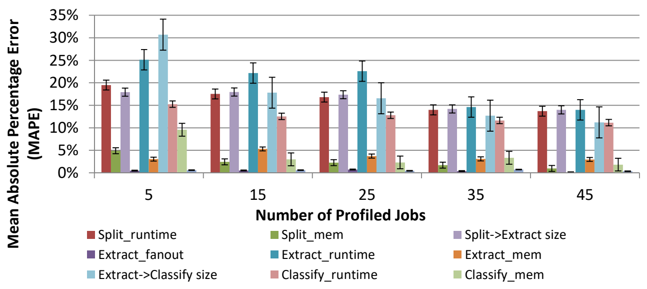
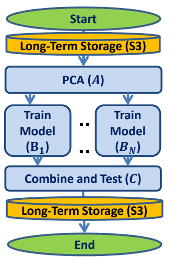


*Figure 10: Error in parameter estimation for Video Analytics application. Convergence point is reached with e.g., 35 jobs. Split_mem, Extract_mem, and Classify_mem represent memory footprints for Split, Extract and Classify functions respectively. All parameters are predicted using polynomial regression models, which take the application's input size in MBs as input.*



| Approach | SONIC | SAND | OpenLambda + S3 |
|---|---|---|---|
| Node A's placement | 1 x m5.large | 1 x m5.Xlarge | 1 x m5.large |
| A-B data passing | Direct-Passing = 1.9 sec | Local-Passing = 0.23 sec | Remote-Passing = 8.3 sec |
| Node B's placement | 3 x m5.large with 2 lambdas/VM | 1 x m5.Xlarge | 6 x m5.large with 1 lambda/VM |
| B-C data passing | Direct-Passing = 0.2 sec | Local-Passing = 0.03 sec | Remote-Passing = 0.23 sec |
| Node C's placement | 1 x m5.large | 1 x m5.Xlarge | 1 x m5.large |
| **E2E Latency** | **27.5 sec** | **71.4 sec** | **37.4 sec** |
| **E2E Cost 1K runs ($)** | **1,196** | **1,877** | **1,797** |

| | PCA | Direct Passing | Train | Direct Passing | Combine |
|---|---|---|---|---|---|
| m5.l | **VM-Count = 1<br>Exec-Time = 9.4<br>Price (x1K) = $686** | | **VM-Count = 3<br>Total Latency = 15.9 s<br>Price (x1K) = $480** | | **VM-Count = 1<br>Total Latency = 2 s<br>Price (x1K) = $24** |
| m5.xl | VM-Count = 1<br>Exec-Time = 9.4<br>Price (x1K) = $1,370 | | VM-Count = 2<br>Total Latency = 15.2 s<br>Price (x1K) = $640 | | VM-Count = 1<br>Total Latency = 1.97 s<br>Price (x1K) = $48 |
| m5.2xl | VM-Count = 1<br>Exec-Time = 9.4<br>Price (x1K) = $2,744 | | VM-Count = 1<br>Total Latency = 14.6 s<br>Price (x1K) = $840 | | VM-Count = 1<br>Total Latency = 1.96 s<br>Price (x1K) = $72 |

*Figure 11: Example showing the benefits of* SONIC *over baselines. The table at the bottom shows possible λ placements for each stage in the LightGBM application and their corresponding latency and cost. We highlight the best sequence of decisions that achieves the best Perf/$ for the entire DAG.*

the LightGBM application with a fanout degree of 6 and show the E2E latency and Cost for all baselines. First, SAND leverages data locality between all stages and hence uses the same single VM of size m5.Xlarge for the entire application. This causes the fanout stage (TrainModel) to experience serialized execution as this single VM does not have enough resources to execute all invocations in parallel and hence increases the E2E latency to 71.4 sec. Compared to OpenLambda + S3, we notice that passing data between PCA and TrainModel is very slow with remote passing as it takes 8.3 sec. However, if direct passing is used (as done by SONIC), the data passing time becomes 1.9 sec only. We also notice that SONIC places each pair of lambdas in one VM using a total of 3 VMs. This makes direct passing faster as it only needs to transfer 3 copies of the transformed training data. Recall that this application has a broadcast fanout and all lambdas in TrainModel stage get the same copy of PCA's output file. In conclusion, with SONIC's optimized data passing and placement decisions, the

E2E latency is reduced by 27% over OpenLambda + S3 and by 62 % over SAND, and the Cost is reduced by 33% over OpenLambda + S3 and 36% over SAND.

#### 5.6.3 Sensitivity to Input Content

SONIC uses only the input size information, as opposed to content awareness, to predict DAG parameters for a new job[3]. For example, for Video Analytics (Fig. 1) some of the parameters like the intermediate data size are sensitive to the video size (bytes), which depends on video bitrate specification. We want to examine how SONIC's performance would be impacted on test videos different from training. In Fig. 13, we show the performance gain for three variants of SONIC, testing on a 396-sec video from the Sports category. First, we train SONIC on 60 videos from the same Sports category, which shows the best performance among the 3 variants. Second, we show SONIC's performance with training videos from 5 different categories (60 in all, split equally), which shows an 8% performance reduction vis-à-vis the first. The third variant is trained with 60 videos from the *News* category, which has a 25% lower bitrate than the *Sports* category on average. This difference in categories causes a further performance reduction by 19% due to the error in predicting the fanout degree (40%) and intermediate data size between the `Split_Video` and `Extract_Frame` functions (21%). All three variants still show a significant gain over SAND and OpenLambda baselines. As expected, the higher the difference in critical features of the training and testing data (that can impact the DAG's parameters), the lower is SONIC's performance. Critical features are those that affect the compute time or the data passing volume, e.g., video bitrate. One solution to this limitation is to cluster the jobs based on the critical features and train a separate prediction model for each cluster.

#### 5.6.4 Tolerance to Prediction Noise

We examine SONIC's sensitivity to prediction noise. We use the MapReduce Sort application with 30 each of map and reduce functions and apply varying levels of synthetic noise to our memory footprint predictions. We show the impacts of over-predicting (i.e., the predicted memory footprint is *higher* than the actual), and under-predicting in Fig. 14. For calibration, the natural error of SONIC in prediction of memory parameters is 7%. We draw several conclusions. First, error levels of less than $\pm 20\%$ have little impact since with low levels of noise, the (categorical) decisions by SONIC for lambda-placement and data passing are unchanged. Second, under-predicting (the bars with -ve errors) has lesser impact on SONIC than over-predicting. Under-predicting causes SONIC to allocate fewer VMs (1 VM per 3 lambdas in this experiment) than without synthetic noise (1 VM per 2 lambdas). This causes the execution of only two lambdas in parallel while queuing the third lambda, increasing the job's E2E execution time. On the other hand, over-estimation of the memory causes SONIC to allocate more VMs than what the job actually needs (1 VM per lambda). The increase in latency with under-prediction is partly compensated for by the reduction in the \$ cost, while with over-prediction, the increase in the \$ cost dominates over the reduction in latency.

[3]Although this hurts SONIC's prediction accuracy for content-dependent DAGs, it allows generalizing without application-specific processing. Further, public cloud providers often are not allowed to look into client data.

#### 5.6.5 Varying Cold-Start Overheads

In this experiment, we evaluate the effect of varying cold:hot execution times. We use a synthetic application of one stage containing 10 parallel functions and vary the function's startup:steady state compute ratios. We compare SONIC to two static baselines, SAND and OpenLambda+S3, in Fig. 15. SAND always prefers data locality and hot execution over parallelism. We notice that this approach is beneficial for lambdas that have a gap of $5\times$ or more between cold and hot execution times. However, this solution is counterproductive when the gap between cold and hot executions is lower, unnecessarily forcing lambdas to run sequentially. The exact opposite happens with OpenLambda — it is competitive with SONIC for cold to hot execution ratios of $2\times$ or less but suffers increasingly as the ratios become higher as it always incurs cold-start costs. SONIC achieves close-to-optimal performance across the entire range of cold-to-hot execution ratios due to its ability to estimate the execution times under cold and hot executions and to select the best lambda placement and data passing approach dynamically. In practice, we find that the ratio varies in the range $[1, 3.6]$ (highest for Video Analytics' Classify frame due to a heavy NN model); prior work has shown that the ratio can be as high as $9.6\times$ (Figure 21 in [50]). We also evaluate SONIC with varying fanout and ratios of compute time to total execution time (=compute time+data passing time). SONIC's gain over OpenLambda is more significant at lower compute ratios as data exchange dominates, while it is more significant over SAND at higher fanouts (data locality hurts parallelism).

## 6 Related Work

**Data-passing in serverless environments:** We are not the first to identify data-passing latency as a key challenge for chained lambda execution [11, 12, 30, 31, 67]. Pocket [37] and Locus [53] implement multi-tier remote storage solutions to improve the performance and cost-efficiency of ephemeral data sharing in serverless jobs. SONIC can leverage these remote storage systems (e.g., we have evaluated SONIC with Pocket) while automatically optimizing data-passing performance with *VM-Storage* (as in SAND [2]) and *Direct-Passing* methods, which minimize data copying. Pocket also requires hints from the user about parameters of the DAG, which we infer using our modeling approach.

Prior systems have shown that serverless functions can communicate directly using NAT (network address translation) traversal techniques. ExCamera [22] uses a rendezvous server and a fleet of long-lived ephemeral workers to enable

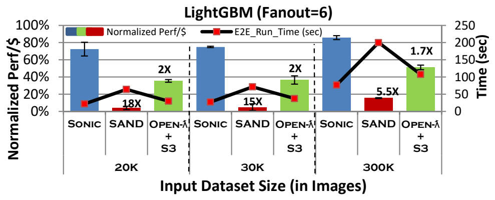


*Figure 12: Normalized Perf/\$ of* SONIC *vs SAND and OpenLambda+S3 with varying input sizes. We fix the Fanout-degree=6 and change the number of images used in training the Random-Forest model.*

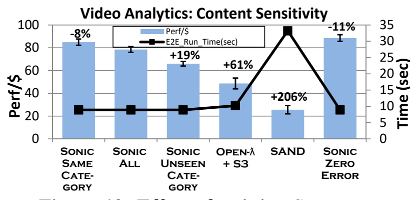


*Figure 13: Effect of training* SONIC *on YouTube video categories similar or dissimilar to test. % over the bars represent the* SONIC*'s (second bar from left) gain over that baseline.*

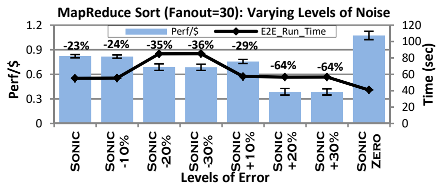


*Figure 14: Impact of noise in* SONIC*'s memory footprint predictions. Errors of less than ± 10% have small effect and over-prediction has higher effect than under-prediction. The values over the bars are w.r.t.* SONIC *with zero error (rightmost).*

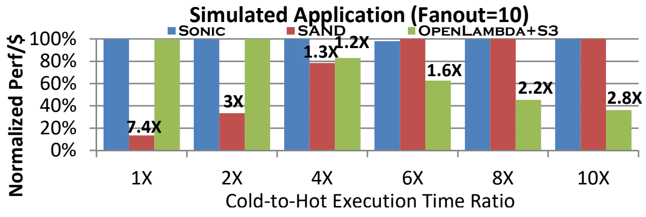


*Figure 15: Effect of varying ratios of cold to hot execution times on* SONIC*, SAND, and OpenLambda+S3. Normalized Perf/\$ is calculated by dividing the Perf/\$ by the max across the three techniques.*

direct communication. However, it needs both endpoint lambdas to be executing at the time of the data transfer, which SONIC does not. gg [21] is a framework for burst-parallel applications that supports multiple intermediate data storage engines, including direct communication between lambdas, which users can choose from. In contrast, SONIC abstracts and adaptively selects the optimal data-passing mechanism.

The need for processing state within serverless frameworks is being increasingly recognized [9, 46, 53, 59], e.g., for fine-grained state sharing or coordination among processes in ML workflows. This trend will emphasize the importance of efficient data passing among functions like SONIC provides. Automated tuning systems of clusters configurations have been proposed in [40–42]. However, these systems use black-box machine learning optimization and rely on hundreds of offline profiling runs to build accurate performance models. Cloudburst proposes using a cache on each lambda-hosting VM for fast retrieval of frequently accessed data in a remote key-value store [61], adding a modicum of statefulness to serverless workflows. SONIC does not cache data, but still exploits data locality with its lambda placement.

**Efficiency of serverless executions**. There is flourishing work to make serverless executions more efficient. One strategy optimizes cold-start latencies, which will influence SONIC's function placement decisions as in § 5.6.5 (e.g., SOCK [50], SEUSS [10], and Firecracker [1].) Another strategy optimizes in the isolation *vs.* agility spectrum (e.g., Firecracker [1], MVE [16] (supporting hugely concurrent services as in popular games), Spock [28], and Fifer [27] (hybrids of serverless and other cloud technologies for microservices). The data-passing selection in these works can benefit from SONIC. Costless [17] optimizes lambda fusion and placement, reducing the number of state transitions. This contribution is orthogonal and beneficial to SONIC.

**Cluster computing frameworks**: Many distributed computing frameworks, such as Spark [66], Dryad [33], CIEL [48], and Decima [43] use DAGs of jobs to schedule tasks. With some engineering effort, SONIC can be used to support the data passing on these DAGs. However, SONIC stays close to the spirit of serverless in that it requires a minimal number of user hints or configuration options.

## 7 Conclusion

Optimizing the cost and performance of analytics jobs on serverless platforms requires minimizing the data passing latency between chained lambdas. The optimal data passing method depends on application-specific parameters, such as the input data size and the degree of parallelism. We presented SONIC, a system that *jointly* optimizes the inter-lambda data exchange method and lambda placement. SONIC performs online profiling to determine the relation between the application's input size and its DAG parameters. Afterward, SONIC applies an online Viterbi algorithm, to globally minimize the application's end-to-end latency, normalized by cost. SONIC achieves lower (\$ cost-normalized) latency against four competitive baselines for three popular serverless applications. Moreover, SONIC is able to adjust the best data passing method based on infrastructure changes such as network bandwidth fluctuations. In ongoing work, we are designing SONIC to handle conditional control flows in the application DAG through content-aware prediction.

## 8 Acknowledgement

We thank our shepherd Mike Mesnier and all the reviewers for their insightful comments. This work is supported in part by NSF grants: 1919197, 2016704, NIH Grant 1R01AI123037, Lilly Endowment (Wabash Heartland Innovation Network - WHIN), Amazon Research Awards (ARA), and Adobe Research. This material was in part based upon research supported by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research, under contract DE-AC02-06CH11357. The funders had no role in the design or execution of the work. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

## References

[1] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)* (2020), pp. 419–434.

[2] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC)* (2018), pp. 923–935.

[3] Amazon. Aws lambda features. https://aws.amazon.com/lambda/features/, 2021.

[4] Amazon. Aws step functions: Assemble functions into business-critical applications. https://aws.amazon.com/step-functions/, Last retrieved: Jan, 2021.

[5] Ao, L., Izhikevich, L., Voelker, G. M., and Porter, G. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 263–274.

[6] AWS. Aws lambda faqs. https://aws.amazon.com/lambda/faqs/, 2021.

[7] Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Scalable atomic visibility with ramp transactions. *ACM Transactions on Database Systems (TODS) 41*, 3 (2016), 1–45.

[8] Banks, D. L., and Fienberg, S. E. Multivariate statistics.

[9] Barcelona-Pons, D., Sánchez-Artigas, M., París, G., Sutra, P., and García-López, P. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference* (2019), pp. 41–54.

[10] Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., and Appavoo, J. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.

[11] Carreira, J., Fonseca, P., Tumanov, A., Zhang, A., and Katz, R. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS* (2018), vol. 2018.

[12] Carreira, J., Fonseca, P., Tumanov, A., Zhang, A., and Katz, R. Cirrus: a serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 13–24.

[13] CloudWatch, A. Monitoring ec2 network utilization. https://cloudonaut.io/monitoring-ec2-network-utilization/, 2020.

[14] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 153–167.

[15] Deng, L. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine 29*, 6 (2012), 141–142.

[16] Donkervliet, J., Trivedi, A., and Iosup, A. Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)* (2020).

[17] Elgamal, T. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)* (2018), IEEE, pp. 300–312.

[18] Forney, G. D. The viterbi algorithm. *Proceedings of the IEEE 61*, 3 (1973), 268–278.

[19] Forney Jr, G. D. The viterbi algorithm: A personal history. *arXiv preprint cs/0504020* (2005).

[20] Forums, A. How to set memory size of azure function? https://social.msdn.microsoft.com/Forums/en-US/9a6e4728-d54a-488d-9007-5fdb80fc105e/how-to-set-memory-size-of-azure-function?forum=AzureFunctions, 2018.

[21] Fouladi, S., Romero, F., Iter, D., Li, Q., Chatterjee, S., Kozyrakis, C., Zaharia, M., and Winstein, K. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference* (2019), pp. 475–488.

[22] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. 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)* (2017), pp. 363–376.

[23] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi* (2011), vol. 11, pp. 24–24.

[24] GOOGLE. Cloud composer: A fully managed workflow orchestration service built on apache airflow. https://cloud.google.com/composer, Last retrieved: Jan, 2021.

[25] GRANDL, R., KANDULA, S., RAO, S., AKELLA, A., AND KULKARNI, J. {GRAPHENE}: Packing and dependency-aware scheduling for data-parallel clusters. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 81–97.

[26] GROTHER, P. J. Nist special database 19 handprinted forms and characters database. *National Institute of Standards and Technology* (1995).

[27] GUNASEKARAN, J. R., THINAKARAN, P., CHIDAMBARAM, N., KANDEMIR, M. T., AND DAS, C. R. Fifer: Tackling underutilization in the serverless era. *arXiv preprint arXiv:2008.12819* (2020).

[28] GUNASEKARAN, J. R., THINAKARAN, P., KANDEMIR, M. T., URGAONKAR, B., KESIDIS, G., AND DAS, C. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019), IEEE, pp. 199–208.

[29] HADARY, O., MARSHALL, L., MENACHE, I., PAN, A., GREEFF, E. E., DION, D., DORMINEY, S., JOSHI, S., CHEN, Y., RUSSINOVICH, M., ET AL. Protean:{VM} allocation service at scale. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)* (2020), pp. 845–861.

[30] HELLERSTEIN, J. M., FALEIRO, J., GONZALEZ, J. E., SCHLEIER-SMITH, J., SREEKANTI, V., TUMANOV, A., AND WU, C. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[31] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[32] HSU, C.-J., NAIR, V., MENZIES, T., AND FREEH, V. W. Scout: An experienced guide to find the best cloud configuration. *arXiv preprint arXiv:1803.01296* (2018).

[33] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 Eurosys Conference* (March 2007), Association for Computing Machinery, Inc.

[34] JONAS, E., SCHLEIER-SMITH, J., SREEKANTI, V., TSAI, C.-C., KHANDELWAL, A., PU, Q., SHANKAR, V., CARREIRA, J., KRAUTH, K., YADWADKAR, N., ET AL. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[35] KIM, Y., AND LIN, J. Serverless data analytics with flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)* (2018), pp. 451–455.

[36] KLIMOVIC, A., WANG, Y., KOZYRAKIS, C., STUEDI, P., PFEFFERLE, J., AND TRIVEDI, A. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIXATC 18)* (2018), pp. 789–794.

[37] KLIMOVIC, A., WANG, Y., STUEDI, P., TRIVEDI, A., PFEFFERLE, J., AND KOZYRAKIS, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 427–444.

[38] LE, T. N., SUN, X., CHOWDHURY, M., AND LIU, Z. Allox: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.

[39] LIGHTGBM. Lightgbm's documentation! https://lightgbm.readthedocs.io/en/latest/index.html, 2021.

[40] MAHGOUB, A., MEDOFF, A. M., KUMAR, R., MITRA, S., KLIMOVIC, A., CHATERJI, S., AND BAGCHI, S. {OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud. In *USENIX Annual Technical Conference (USENIX ATC)* (2020), pp. 189–203.

[41] MAHGOUB, A., WOOD, P., GANESH, S., MITRA, S., GERLACH, W., HARRISON, T., MEYER, F., GRAMA, A., BAGCHI, S., AND CHATERJI, S. Rafiki: a middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (2017), pp. 28–40.

[42] MAHGOUB, A., WOOD, P., MEDOFF, A., MITRA, S., MEYER, F., CHATERJI, S., AND BAGCHI, S. {SOPHIA}: Online reconfiguration of clustered nosql databases for time-varying workloads. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)* (2019), pp. 223–240.

[43] MAO, H., SCHWARZKOPF, M., VENKATAKRISHNAN, S. B., MENG, Z., AND ALIZADEH, M. Learning scheduling algorithms for data processing clusters. In

*Proceedings of the ACM SIGCOMM* (2019), ACM, pp. 270–288.

[44] MARANDI, P. J., PRIMI, M., AND PEDONE, F. High performance state-machine replication. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)* (2011), IEEE, pp. 454–465.

[45] MICROSOFT. Azure durable functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/durable/, Last retrieved: Jan, 2021.

[46] MORITZ, P., NISHIHARA, R., WANG, S., TUMANOV, A., LIAW, R., LIANG, E., ELIBOL, M., YANG, Z., PAUL, W., JORDAN, M. I., ET AL. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)* (2018), pp. 561–577.

[47] MÜLLER, I., MARROQUÍN, R., AND ALONSO, G. Lambada: Interactive data analytics on cold data using serverless cloud infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 115–130.

[48] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), NSDI'11, p. 113–126.

[49] MXNET. Using pre-trained deep learning models in mxnet. https://mxnet.apache.org/api/python/docs/tutorials/packages/gluon/image/pretrained_models.html, 2021.

[50] OAKES, E., YANG, L., ZHOU, D., HOUCK, K., HARTER, T., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. {SOCK}: Rapid task provisioning with serverless-optimized containers. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)* (2018), pp. 57–70.

[51] OPENLAMBDA. An open source serverless computing platform. https://github.com/open-lambda/open-lambda, 2021.

[52] PERRON, M., CASTRO FERNANDEZ, R., DEWITT, D., AND MADDEN, S. Starling: A scalable query engine on cloud functions. In *SIGMOD* (2020).

[53] PU, Q., VENKATARAMAN, S., AND STOICA, I. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 193–206.

[54] RAUPACH, B. Choosing the right amount of memory for your aws lambda function. https://medium.com/@raupach/choosing-the-right-amount-of-memory-for-your-aws-lambda-function-99615ddf75dd, 2018.

[55] RAUSCH, T., HUMMER, W., MUTHUSAMY, V., RASHED, A., AND DUSTDAR, S. Towards a serverless platform for edge {AI}. In *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)* (2019).

[56] RIBEIRO, V. J., RIEDI, R. H., BARANIUK, R. G., NAVRATIL, J., AND COTTRELL, L. pathchirp: Efficient available bandwidth estimation for network paths. In *Passive and active measurement workshop* (2003).

[57] RIBENZAFT, R. How to make aws lambda faster: Memory performance. https://epsagon.com/observability/how-to-make-aws-lambda-faster-memory-performance/, 2018.

[58] SHAHRAD, M., FONSECA, R., GOIRI, I., CHAUDHRY, G., BATUM, P., COOKE, J., LAUREANO, E., TRESNESS, C., RUSSINOVICH, M., AND BIANCHINI, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 205–218.

[59] SHANKAR, V., KRAUTH, K., PU, Q., JONAS, E., VENKATARAMAN, S., STOICA, I., RECHT, B., AND RAGAN-KELLEY, J. Numpywren: Serverless linear algebra. In *ACM Symposium on Cloud Computing (SoCC)* (2020), pp. 1–14.

[60] SREEKANTI, V., WU, C., CHHATRAPATI, S., GONZALEZ, J. E., HELLERSTEIN, J. M., AND FALEIRO, J. M. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–15.

[61] SREEKANTI, V., WU, C., LIN, X. C., SCHLEIER-SMITH, J., FALEIRO, J. M., GONZALEZ, J. E., HELLERSTEIN, J. M., AND TUMANOV, A. Cloudburst: Stateful functions-as-a-service. https://arxiv.org/pdf/2001.04592.pdf, 2020.

[62] TOOTAGHAJ, D. Z., FARHAT, F., ARJOMAND, M., FARABOSCHI, P., KANDEMIR, M. T., SIVASUBRAMANIAM, A., AND DAS, C. R. Evaluating the combined impact of node architecture and cloud workload characteristics on network traffic and performance/cost. In *2015 IEEE International Symposium on Workload Characterization* (2015), IEEE, pp. 203–212.

[63] Wang, H., Lee, K. S., Li, E., Lim, C. L., Tang, A., and Weatherspoon, H. Timing is everything: Accurate, minimum overhead, available bandwidth estimation in high-speed wired networks. In *Proceedings of the 2014 Conference on Internet Measurement Conference* (2014), pp. 407–420.

[64] Xu, R., Zhang, C.-l., Wang, P., Lee, J., Mitra, S., Chaterji, S., Li, Y., and Bagchi, S. Approxdet: content and contention-aware approximate object detection for mobiles. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems* (2020), pp. 449–462.

[65] Yi, X., Park, D., Chen, Y., and Caramanis, C. Fast algorithms for robust pca via gradient descent. In *Advances in neural information processing systems* (2016), pp. 4152–4160.

[66] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. Spark: Cluster computing with working sets. *HotCloud 10*, 10-10 (2010).

[67] Zhang, T., Xie, D., Li, F., and Stutsman, R. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing* (2019), pp. 1–12.