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

*Ashraf Mahgoub[(a)], Karthick Shankar[(b)], Subrata Mitra[(c)],*
*Ana Klimovic[(d)], Somali Chaterji[(a)], Saurabh Bagchi[(a)]*

*(a) Purdue University, (b) Carnegie Mellon University,*
*(c) Adobe Research, (d) ETH Zurich*

## Abstract

Data analytics applications are increasingly leveraging serverless execution environments for their ease-of-use and pay-as-you-go billing. Increasingly, such applications are composed of multiple functions arranged in some workflow. However, the current approach of exchanging intermediate (ephemeral) data between functions through remote storage (such as S3) introduces significant performance overhead. We show that there are three alternative 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 different baselines: SAND [UsenixATC-18], Vanilla OpenLambda [HotCloud-16], OpenLambda integrated with Pocket [OSDI-18], and AWS Lambda (state of practice).

## 1 Introduction

Serverless computing platforms provide on-demand scalability and fine-grained allocation of resources. In this computing model, the cloud provider runs the servers and manages all the administrative tasks (e.g., scaling, capacity planning, etc.), while users focus on their application logic. Due to its significant advantages, serverless computing is becoming increasingly popular for complex workflows such as data processing pipelines [40,56], machine learning pipelines [14,59], and video analytics [6,27]. Accordingly, major cloud computing providers recently introduced their serverless workflow services such as AWS Step Functions [5], Azure Durable Functions [48], and Google Cloud Composer [29], 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) [21,56]. 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 has been identified as a major challenge in serverless workflows [39,41,50]. The reason is that, 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 that it makes for a clear separation between compute and storage resources, it adds significant performance overheads, especially for data-intensive applications [50]. For example, Pu *et al.* [56] 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 execution time (Fig. 2, fanout = 1). Previous approaches target reducing this overhead, either by replacing disk-based storage (e.g., S3) by memory-based storage (e.g., ElastiCache Redis), or by combining different storage media (e.g., DRAM, SSD, NVMe) to match application needs [15,42,56]. However, these approaches still require passing the data over the network mul-

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

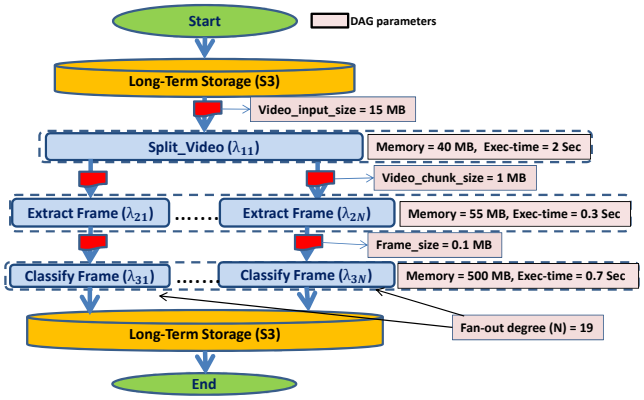


***Figure 1: DAG overview (DAG definition provided by the user and our profiled DAG parameters) for Video Analytics application.***

tiple times, adding significant latency to the job's execution time. Moreover, in-memory storage services are much more expensive than disk-based storage (e.g., ElastiCache Redis costs 700x compared to S3 per unit data volume [56]).

In our work, we show that the cloud provider can implement communication-aware placement of lambdas by optimizing data exchange between chained functions in DAGs. 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. Alternately, 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 copying intermediate data elements once, serving as a middleground 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, no single mechanism prevails across all serverless applications with different data dependencies. For example, while *Direct-Passing* does not impose strict scheduling constraints, scalability can become an issue when a large number of receiving functions copy data simultaneously saturating the VM's outgoing network bandwidth. *Hence, we need a hybrid, fine-grained data-passing approach that optimizes the data-passing for every edge of the application DAG.*

**Our solution**: We propose SONIC, a unified API and the system 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 decides the best data-passing method for every edge in the application DAG to minimize data-passing latency and cost. We show that this selection can be impacted by several parameters such as the size of input, the application's degree of parallelism, and VM network

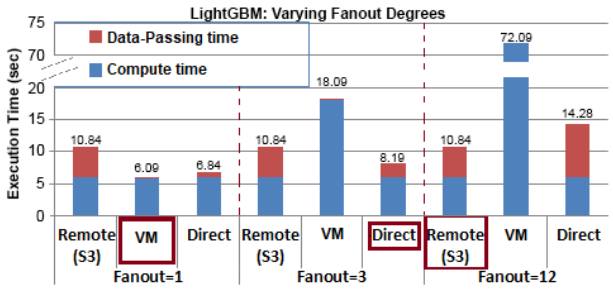


***Figure 2: Execution time comparison with Remote-Storage, VM-Storage, and Direct-Passing for the LightGBM application with Fanout = 1, 3, 12. The best data-passing method differs in every case.***

bandwidth. SONIC adapts its decision dynamically to changes in these parameters in an application-specific manner. The workflow of SONIC is shown in Fig. 3.

We note that a locally optimized data-passing decision for a given stage can become sub-optimal for the entire DAG. As an example, placing two functions together in the same VM and using VM-Storage will reduce the data passing time between these two stages but if the second function generates a large volume of intermediate data then the cost of transporting that data to a different VM may be high. Therefore, SONIC develops a Viterbi-based algorithm to avoid greedy data-passing decisions and provide optimized latency and cost across the entire DAG. SONIC abstracts its hybrid data-passing selection and provides the user with a simple file-based API. With SONIC 's API, users always read and write data as files to storage that appear local to the user. SONIC is designed to be integrated with a cluster resource manager (e.g., Protean [34]) that assigns VM requests to the physical hardware and optimizes provider-centric metrics such as resource utilization and load balancing (Fig 4).

As our approach requires integration with a serverless platform (and commercial platforms do not allow for such implementation), we integrate SONIC with OpenLambda [36]. We compare SONIC 's performance using three popular analytics applications 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 [42]. Our evaluation shows that SONIC outperforms all baselines. SONIC achieves between 34% to 158% higher performance/$ (here performance is the inverse of latency) over OpenLambda+S3, between 59% to 2.3X over OpenLambda+Pocket, and between 1.9X to 5.6X 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 provides selection of optimized data-passing methods between any two serverless functions

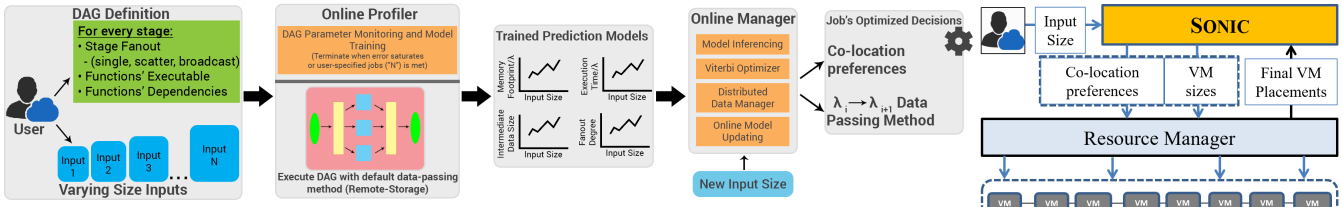


*Figure 3: Workflow of* SONIC*: Users provide a DAG definition and input data files for profiling.* SONIC*'s online profiler executes the DAG and generates regression models that map the input size to the DAG's parameters. For every new input, the online manager uses the regression models to identify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

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

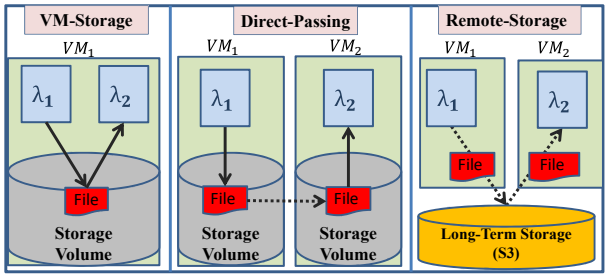


*Figure 5: The three data-passing options between two lambdas (*$\lambda_1$ *and* $\lambda_2$*). VM-Storage forces* $\lambda_2$ *to run on the same VM as* $\lambda_1$ *and avoids copying data. Direct-Passing stores the output file of* $\lambda_1$ *in the source VM's storage, and then copies the file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

in a serverless workflow. SONIC adapts its decisions dynamically to changes in the input, degree of parallelism, etc.

**(3)** We evaluate SONIC's sensitivity to serverless-specific challenges such as the cold-start problem, lack of point-to-point communication, and provider's desired operating point of not using the input data's content for any decision. SONIC shows its benefit over all baselines with three distinct popular serverless applications, with different input sizes and user-specified parameters.

## 2 Rationale and Overview

Here we discuss in detail the trade-offs of the three data passing methods used by SONIC (Fig. 5), and hence motivate our hybrid and dynamic approach.

### 2.1 Data-passing Methods

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

***VM-Storage:*** This method minimizes latency by leveraging data locality, which is achieved by preserving the local state of the sending λ and scheduling the receiving λ(s) to execute on the same VM with the preserved state. However, when the sending VM's memory cannot fit all receiving λ(s), this method forces the scheduler to run the receiving λs *serially*, either individually or in batches. This sacrifices parallelism in favor of data locality. Also notice that data locality is infeasible if the memory requirement of a *single* receiving λ exceeds the sending VM's capacity, or when the receiving λ collects data from multiple sending 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*, while it copies the data over the network directly between source and destination VMs.

***Remote-Storage:*** This data passing method is the state-of-practice in commercial serverless platforms. This is also the model that is optimized in several recent papers [15, 42, 56]. Uploading output files to a remote storage system and downloading them at destination λ(s) provides high scalability with no restrictions on λ placements. It also has the advantage of almost flat data-passing time with increasing fanout degrees as shown in Fig. 6 due to the highly over-provisioned 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 (copies to multiple destinations occur in parallel)

### 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). To demonstrate these trade-offs, 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 se-

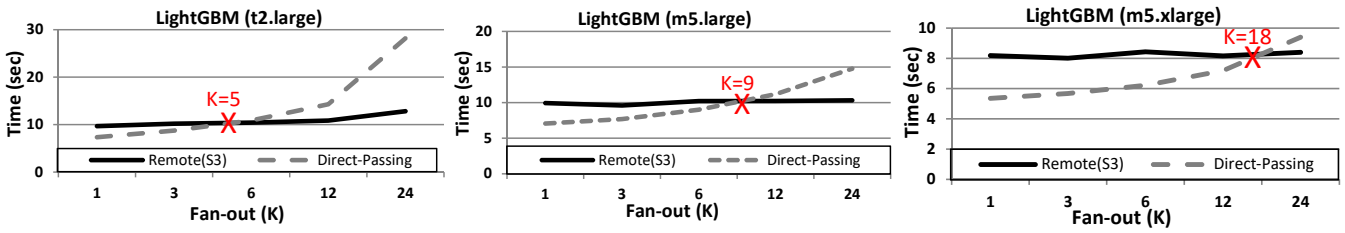


***Figure 6: Comparison between Remote-Storage and Direct-Passing for LightGBM workload with varying fanout degrees. Typically, beyond a certain fanout, Remote has lower execution time than Direct-Passing. A more well-provisioned VM (m5.xlarge) will shift the crossover point to the right.***

lects 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 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[2]. 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 [62]. *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. 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.

[2]Some broadcast tree-based protocols like [8, 22] may enable higher scalability for Direct-Passing but eventually it will run into this bottleneck. Further, these solutions do not support Scatter Fanout.

## 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

**Developers:** SONIC serves as a data-exchange management layer that is deployed and managed by the cloud (FaaS) provider. As shown in Fig. 3, users provide SONIC with an application represented as a DAG. We assume users would execute these application-specific DAG definitions multiple times with potentially different input data. Each such execution instance is referred to as a *job*. Our approach 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) executable for every λ; (2) dependency between the λs; (3) 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 [25, 58, 61]). 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.

**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 [30], Protean [34], AlloX [43] and DRF [28] 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 triggers the execution of 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* because the scheduling constraint that it imposes may be unacceptable to the manager. Therefore, for that selection, SONIC always proposes the second-best option. We show in the Supplement (§ 8.6.1) that such override does not hurt SONIC's gain significantly.

**Data-passing Interface**: SONIC abstracts the selection of data-passing methods from users who do not need to write any special code to benefit from the optimal selection. 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.

## 3.2 Online Profiling and Model Training

SONIC profiles jobs *online* in order to understand the impact of changes in input size to the following parameters: (1) Each constituent λ's memory footprint. (2) Each λ's compute time. (3) Intermediate data volume between any two communicating λs. (4) Fanout type and degree in every stage. The word "online" denotes that our system is serving production workloads, while the models are being trained. Notice that the above parameters are *not* dependent on the data-passing method. Hence, initially SONIC uses a default data-passing policy (*Remote-Storage*) while it collects these DAG parameters for the first $N$ runs of the job to train the 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 estimates the values of these DAG parameters 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 [11]. SONIC continuously monitors the error rate for the prediction models during execution and if the error rate exceeds a user-defined threshold (which can happen when the workload evolves), SONIC falls back to the default policy and resumes online training.

We believe this form of online training, while having no conceptual novelty, is a crucial factor for SONIC's adoption in production environments. In discussion 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.

Further, 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 (cold execution).

## 3.3 Minimizing End-to-End Execution Time

At the highest level, 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:

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

Notice that we assume data is balanced among the receiving functions. In case of significant data skew, SONIC can optimize the lambda-placement and data-passing options based on the predicted *max* intermediate data size and *max* execution time instead of the average. We give the equivalent equations

for *Remote-Storage* in the Supplement (§ 8.3). Often cloud providers overprovision network bandwidth [17], 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. This is challenging because of the dependency between decisions in consecutive stages. 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 (the VM hosting `Split_Video`). Accordingly, 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 this issue, we apply the Viterbi algorithm [23] 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. Afterward, 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 $ 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 describe the Viterbi algorithm in detail in the Supplement (§ 8.1).

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 [56]. This reduction in complexity happens because SONIC applies the *same* data-passing method to *all* the functions in a given stage. We choose the Viterbi algorithm as it is guaranteed to find the true maximum a posteriori (MAP) solution [23, 24] unlike heuristic-based searching algorithms such as Genetic Algorithms or Simulated Annealing.

### 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 [10] and AFT [66], idempotence in itself is not sufficient to achieve fault tolerance in serverless, 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*). We show, in Figure 20 in the Supplement, an example of this atomic visibility mechanism applied to a sample DAG. 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 the evaluation are with atomicity enabled. Optionally, users can disable the atomicity of SONIC in cases where high fault tolerance is not required.

**Scalability of** SONIC. For purposes of scalability, SONIC is made distributed through a simple design. It is essentially a shared-nothing system where no state is shared across application jobs. Therefore, 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 [47].

## 4 Implementation

We implement SONIC as a data-passing management layer and we use OpenLambda as our serverless platform [36].

OpenLambda is an open-source platform that relies on Linux containers for isolation and orchestration [55]. 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 [9]. 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 [16] (default of every 5 min). We use a weighted average of historic measures (as is common [60, 69]) 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

We define our primary performance metric to be Perf/$. Since we are minimizing end-to-end (E2E) execution time (synonymous with 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 $-cost normalization effect. For the first metric, higher is better, and for the second, lower is better. If we just refer to the metric as performance, we mean Perf/$. When we want to 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. For example, one can make *VM-Storage* faster by running the application on a VM with a larger memory size and make *Direct-Passing* faster using VMs with a higher network bandwidth. One can also make *Remote-Storage* faster by using an in-memory storage system such as Amazon ElastiCache Redis. *Hence, our solution should consider both latency* ***and*** *cost when selecting the best data passing method.* Consequently, we use the Perf/$ metric. One concern may be that this degrades the raw performance to an unacceptable level. This is handled in our design by providing a lower acceptable bound on the performance metric and SONIC will never make a selection that violates that bound. 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, 11, 12, 21). Finally, there is precedence of prior work in cloud optimization using performance normalized by price [37, 45, 68]. 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** [36]: 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** [42]: 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 and vary the number of nodes in Pocket for every application until we achieve the best Perf/$. The DRAM storage strikes the balance between performance and cost and provides the best Perf/$ (Table 4 in [42]). We include the price of the storage nodes only, excluding master and metadata nodes. We use EC2's per-second level pricing to calculate its $ cost.
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 [54], 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. We do not include the third application from [42], distributed compilation (`cmake`), since we determined (with authors' input) that significant engineering will be required to port it to OpenLambda. **Video Analytics:** Fig. 1 shows the DAG for the Video Analytics application. It performs object classifica-

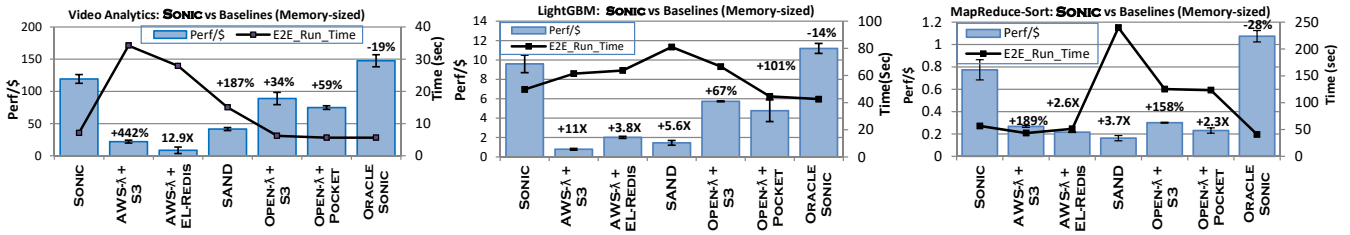


*Figure 7: Performance of* SONIC *and the baselines for our three applications (memory-sized). Two performance metrics are shown: Perf/$ (bars; left axis) and end-to-end execution time (lines; right axis). Relative improvements in Perf/$ due to* SONIC *are at the top of each bar for the corresponding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

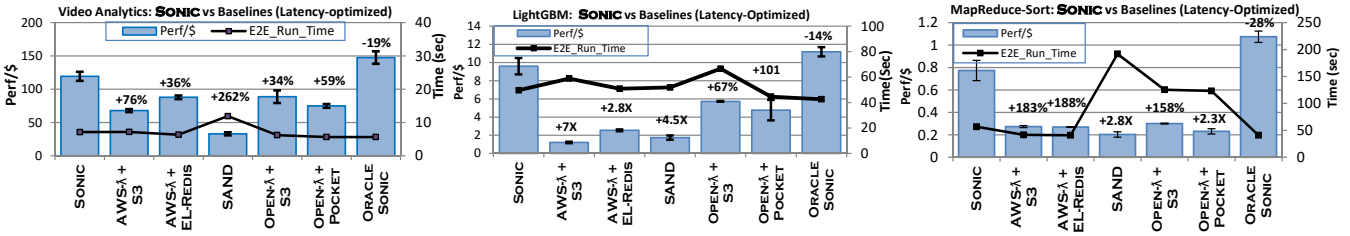


*Figure 8: Performance of* SONIC *and the baselines for our three applications with the latency-optimized configuration.*

tion 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 [52]) 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. (DAG for LightGBM in Supplement (Fig. 14); `LightGBM` Python library used for decision tree training [44]). 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 [19, 31].

**MapReduce Sort:** This application implements MapReduce sort with serverless functions. (DAG for MapReduce Sort in Supplement (Fig. 15)). In the first stage, $K$ parallel lambdas (i.e., mappers; $K$ = user parameter) fetch the input file from a remote data store (e.g., S3) and then generate the intermediate files. Next, $K$ parallel lambdas (i.e., reducers) 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 § 5.6.1). 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 35 jobs, we reach convergence for all predictions for all applications with a low average Mean Absolute Percentage Error (MAPE) $\leq$ 15% (result plots in the Supplement, Figures 16 and 17).

**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 from these end-to-end experiments. 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 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. SONIC's gains over OpenLambda are higher for LightGBM and MapReduce Sort as these two have higher intermediate data volumes and SONIC significantly reduces data-passing times. 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

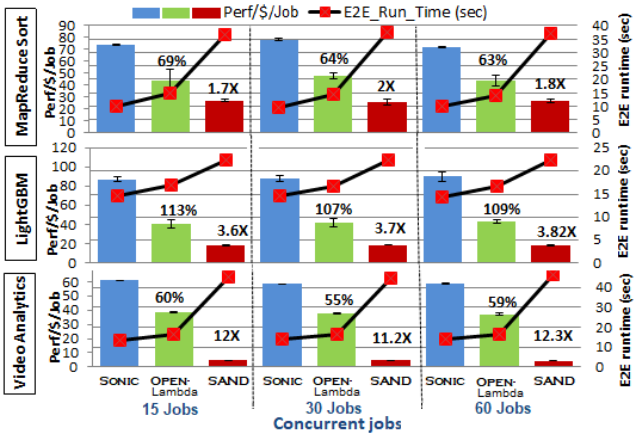


***Figure 9: Scalability of SONIC, OpenLambda+S3, and SAND with equal portions of our three apps running concurrently. SONIC maintains its improvement over the entire scale, in both Perf/$/job and raw latency.***

fully utilized (except for SAND). We show the Perf/$/job and E2E runtime for every app in Figure 9[3]. We notice that the gain of SONIC over both baselines is consistent across the different degrees of concurrency, which shows SONIC's ability to seamlessly scale with the number of concurrent invocations. It is of course not surprising that the VM infrastructure scales up — the question was would SONIC scale up as well. This experiment answers that question in the affirmative (within the scales of the experiment), for SONIC as well as for the two baselines. This is expected from the design of SONIC where most of its components are stateless and different instances are spun up to handle more input jobs (Section 3.5). Since SAND schedules the entire job to execute on a single VM, it cannot utilize all the compute cores and hence suffers from very high latency. In contrast, OpenLambda uses *Remote-Storage* passing between functions, which allows scheduling functions on different VMs and utilizing the available compute resources. SONIC's hybrid approach achieves both lower E2E execution time along with high resource utilization, and therefore, achieves better Perf/$ and raw latency than all baselines.

### 5.5.2 Varying Intermediate Data Size

In Fig. 10, 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 [70] and it dominates the total computation time). Recall that the Oracle

[3]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.

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 sensitivity to input content dependency, prediction noise, and varying cold-start times. We also show SONIC's gains over all baselines under network bandwidth fluctuations and co-location constraints in the Supplement (§ 8.6.2 & § 8.6.1 respectively).

### 5.6.1 Sensitivity to Input Content

SONIC uses only the input size information, as opposed to content awareness, to predict DAG parameters for a new job[4]. 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. 11, 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.

### 5.6.2 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. 12. 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.

[4]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.

### 5.6.3 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. 13. 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 [54]). We also evaluate SONIC with varying fanout and ratios of compute time to total execution time (=compute time+data-passing time). We show SONIC's gains over SAND and OpenLambda+S3 in Supplement (§ 8.6.3). 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 [14, 15, 35, 36, 72]. Pocket [42] and Locus [56] 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*

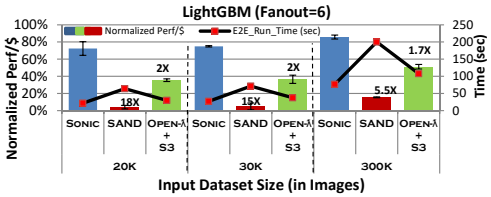


*Figure 10: Normalized Perf/$ of* SONIC *vs SAND and OpenLambda+S3 with varying input sizes. We fix the Fanout-degree=6 and change the number of images used in training the Random-Forest model.*

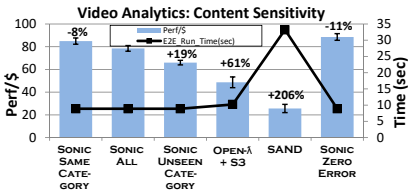


*Figure 11: Effect of training* SONIC *on YouTube video categories similar or dissimilar to test. % over the bars represent the* SONIC*'s (second bar from left) gain over that baseline.*

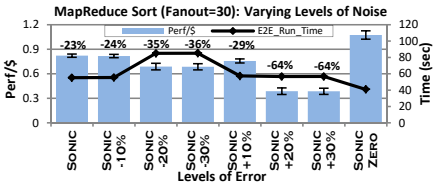


*Figure 12: Impact of noise in* SONIC*'s memory footprint predictions. Errors of less than ± 10% have small effect and over-prediction has higher effect than under-prediction. The values over the bars are w.r.t.* SONIC *with zero error (rightmost).*

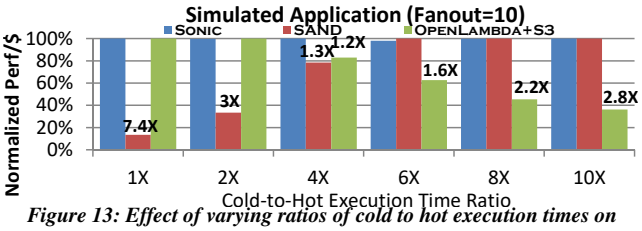


*Figure 13: Effect of varying ratios of cold to hot execution times on* SONIC*, SAND, and OpenLambda+S3. Normalized Perf/$ is calculated by dividing the Perf/$ by the max across the three techniques.*

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 [27] uses a rendezvous server and a fleet of long-lived ephemeral workers to enable direct communication. However, it needs both endpoint lambdas to be executing at the time of the data transfer, which SONIC does not. gg [26] 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 [12, 49, 56, 63], 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. Cloudburst proposes using a cache on each lambda-hosting VM for fast retrieval of frequently accessed data in a remote key-value store [67], 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 § 5.6.3 (e.g., SOCK [54], SEUSS [13], and Firecracker [1].) Another strategy optimizes in the isolation *vs*. agility spectrum (e.g., Firecracker [1], MVE [20] (supporting hugely concurrent services as in popular games), Spock [33], and Fifer [32] (hybrids of serverless and other cloud technologies for microservices)). The data-passing selection in these works can benefit from SONIC. Costless [21] 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 [71], Dryad [38], CIEL [51], and Decima [46] 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. In § 8.6 [Supplement], we contrast with Spark's data-execution co-location decision.

## 7 Conclusion

For cost-optimal performance of analytics jobs on serverless platforms, it is crucial to minimize the data-passing time between chained lambdas. The selection of the best data-passing method depends on parameters mapped to input sizes and user requirements, such as degree of parallelism and latency requirements. We present SONIC that solves this issue by *jointly* optimizing the lambda placement and the inter-lambda data exchange. SONIC performs online profiling to determine the relation between the application's input size and its DAG parameters. This is followed by a global optimization performed using an online Viterbi algorithm, to minimize the application's end-to-end latency normalized by cost. SONIC achieves lower ($ cost-normalized) latency against all baselines for three popular serverless applications, both for memory-sized and latency-optimized configurations. We draw some key insights about serverless data passing from our work. **First,** SONIC's ability to *dynamically* select the data-passing method for each edge in the application DAG significantly improves the performance vis-à-vis prior work, which relies on static data-passing methods—either always *local* (e.g., SAND) (VM-Storage in SONIC's parlance) or always remote storage (e.g., AWS-λ or OpenLambda, with S3 or Pocket). **Second,** SONIC's ability to use the DAG's input size to predict the execution parameters is crucial to dynamically select the optimal data-exchange method and lambda placement. **Third,** one needs to perform *joint* optimization of the lambda placement and the data-passing method by considering multiple factors, of which the following turn out to be crucial—data locality *vs*. parallelism, cold *vs*. hot executions, and identifying fanout type (scatter or broadcast) and degree. In ongoing work, we are designing SONIC to handle conditional control flows in the application DAG through content-aware prediction.

## 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. How do i benchmark network throughput between amazon ec2 linux instances in the same amazon vpc? https://www.liquidweb.com/blog/why-aws-is-bad-for-small-organizations-and-users/, 2021.

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

[6] 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.

[7] APACHE. Spark: Data locality. https://spark.apache.org/docs/latest/tuning.html#data-locality, 2020.

[8] AWAN, A. A., CHU, C.-H., SUBRAMONI, H., AND PANDA, D. K. Optimized broadcast for deep learning workloads on dense-gpu infiniband clusters: Mpi or nccl? In *Proceedings of the 25th European MPI Users' Group Meeting* (2018), pp. 1–9.

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

[10] 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.

[11] BANKS, D. L., AND FIENBERG, S. E. Multivariate statistics.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] DATAFLAIR. Directed acyclic graph dag in apache spark. https://data-flair.training/blogs/dag-in-apache-spark/, 2018.

[19] 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.

[20] 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).

[21] 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.

[22] FIGUEIRA, S. M. Improving binomial trees for broadcasting in local networks of workstations. *VECPAR'02* (2002).

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

[24] FORNEY JR, G. D. The viterbi algorithm: A personal history. *arXiv preprint cs/0504020* (2005).

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

[32] 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).

[33] 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.

[34] 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.

[35] 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).

[36] 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).

[37] 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).

[38] 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.

[39] 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).

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

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

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

[53] NGUYEN, N., KHAN, M. M. H., AND WANG, K. Towards automatic tuning of apache spark configuration. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)* (2018), IEEE, pp. 417–425.

[54] 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.

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

[56] 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.

[57] QUBOLE. Apache spark on qubole. https://www.qubole.com/developers/apache-spark-on-qubole/, 2020.

[58] 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.

[59] 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).

[60] 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).

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

[62] 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.

[63] 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.

[64] SHEA, R., WANG, F., WANG, H., AND LIU, J. A deep investigation into network performance in virtual machine based cloud environments. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications* (2014), IEEE, pp. 1285–1293.

[65] SPARK. pyspark package. https://spark.apache.org/docs/latest/api/python/pyspark.html, 2020.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., STOICA, I., ET AL. Spark: Cluster computing with working sets. *HotCloud 10*, 10-10 (2010).

[72] 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.

[73] ZHU, Y., AND LIU, J. Classytune: A performance autotuner for systems in the cloud. *IEEE Transactions on Cloud Computing* (2019).