Appears in the proceedings of the 38th IEEE International Conference on Massive Storage Systems and Technology (MSST'24)

Balancing Costs and Durability for Serverless Data

Alex Merenstein*, Xinran Wang†, Vasily Tarasov‡, Prajjawal Agarwal*,
Scott Guthridge‡, Kapil Thakkar*, Katherine Wu*, Ali Anwar†, Erez Zadok*
*Stony Brook University, Stony Brook, NY
†University of Minnesota, Minneapolis, MN
‡IBM Research, Almaden, CA
{*mmerenstein, prajagarwal, kathakkar, kaawu, ezk*}*@cs.stonybrook.edu,*
{*wang8740, aanwar*}*@umn.edu,* {*vtarasov, guthridg*}*@us.ibm.com*

***Abstract*—Durability features such as replication or erasure coding serve an important role in storage systems, enabling users to store data without fear of loss due to device failures. However, these durability features come with a cost, in terms of storage, network traffic, and computational overheads. For most data, loss is a catastrophic event and so these overheads are acceptable. However, some data tolerates low durability and does not need the high level of durability that most storage systems provide.**

Identifying the proper level of durability for a piece of data is difficult, especially since it is often not clear how to determine the cost of loss. For some data used in serverless applications, however, this cost is relatively straightforward to calculate: serverless functions are often required to be idempotent, meaning that the data produced by them can be re-created by re-running the function. The cost of losing a piece of data then is merely the cost of re-running the function that originally created the data.

In this paper, we explore the tradeoff between the cost of storing data durably and the cost to re-create data. We focus on serverless data because its ability to be recreated makes it possible to assign a cost to its loss. We develop a mathematical model that relates compute costs, storage costs, and application-specific parameters to calculate the cost-optimal placement of data. We also develop an execution framework capable of handling lost data transparently, enabling applications to use lower-durability storage with no additional burden on the developer. Next, we show how different factors such as failure rate and compute costs affect the placement decision. We find that thanks to the relatively short lifetime of serverless data, the probability of data loss even on low-durability storage is fairly low. Finally, we use the model to place data for several applications, including a video-transcoding application and an image-assembly application. We show that our model can predict execution costs within 7% of actual execution costs, and can reduce storage costs by up to 3× while never exceeding baseline costs.

***Index Terms*—storage, serverless, durability**

I. Introduction

It is a universal truth in storage that device failures will happen: hard disks or SSDs fail, and their data is lost. On an individual basis, these failures are infrequent, with annual failure rates typically ranging from 0.1–5% [1], [2], [3], [4]. However, at large scale, they become a constant issue that must be contended with: clusters with tens or hundreds of thousands of disks are now common [1], [5], [4], [6] and data sizes are increasing [7], [8], resulting in potentially tens of thousands of failures each year.

There is a large body of work on techniques for mitigating such failures. Replication, erasure coding, and regular backups—are all different technologies or practices that accomplish the same goal: preventing data loss in the event of a device failure. Some of these, especially regular backups, can help prevent data loss in the event of other kinds of incidents like operator mistakes or cyberattack. Although the methods differ, they all accomplish this essentially by spreading multiple (full or partial) copies of the data across multiple storage devices so that loss of any one device does not result in data loss. By increasing the redundancy of the data and the number of devices the data are spread across, it is possible to withstand the loss of a large number of disks without loss of the data. The degree to which a storage system can withstand faults without data loss is referred to as its *durability* (see Section II).

In general, the degree of durability desired for data is assumed to be high. Cloud storage systems advertise how many "nines" of durability they offer, typically at least nine but sometimes even up to sixteen "nines" of durability (*i.e.*, 99.9999999% and 99.99999999999999%, respectively). At these high degrees of durability, data loss due to device failure is exceedingly rare. Note, however, that durability calculations usually consider data loss only due to device failures; loss due to operator error or large scale disaster (*e.g.*, destruction of an entire data center) are not factored in [9], [10], [11], [12].

This safety comes at a cost: duplicating data and spreading it across multiple disks (*e.g.*, racks and even regional data centers) incurs overhead in terms of both space and cost. Depending on the durability scheme being used, it may also impact performance. For most data, the additional costs are an acceptable price to pay for safety against data loss. However, some data might not necessarily require such high durability. For example, suppose a dataset is copied from a central repository to a local data center. If the local data center loses the dataset, it can simply re-download it from the central repository and incur some latency. For such data, some amount of loss may be tolerable and the additional costs of full durability may be not justified.

The challenge is that while loss of some data may be tolerable, it is hard to assign a cost to its loss. This in turn makes it difficult to calculate exactly how much loss is acceptable and therefore what level of durability is needed. Additionally, the response to data loss depends on the specifics of the data and the application. For instance, if a backup is lost, there may be no further action necessary, but if a local dataset

is lost, the appropriate response is to re-download it. The lack of standardized response to lost data makes it tempting to simply use high-durability storage and not have to worry about which actions are needed after losing it.

Serverless computing offers a potential solution to this challenge. Serverless platforms often require individual actions to be idempotent [13], [14], [15], [16], [17], meaning that the data created by these actions can be re-created by simply re-running the action. This provides a standard response to handling lost data, and makes assigning a cost to lost data straightforward: it is the cost to re-run the action. In the event that a function cannot be idempotent, for example, if it interacts with an external service, the suggestion is normally to use a workaround such as a helper library to achieve the functional equivalence of idempotency [17], [18]. The recent rise in popularity of serverless platforms provides us an opportunity to re-visit the durability assumptions that have been traditionally made.

In this paper we focus specifically on the durability of data typical to many serverless applications. When choosing the necessary level of durability for serverless data, we must make a tradeoff between the greater storage cost incurred by higher-durability storage schemes and the additional compute cost required to re-create lost data. The tradeoff is intuitively simple: if re-creating the data is costly or not possible because the function is not idempotent, then highly durable, more expensive storage is preferred. Conversely, if re-creating the data is cheap, then cheaper, less durable storage is preferable.

For example, consider the object-detection pipeline depicted in Figure 1. The first action converts the input image to grayscale, creating *Image'*. *Image'* is then processed by a de-noising step, producing data *Image"*. Finally, an object detection step reads *Image"*. If the *Grayscale* action is short and the price of compute is cheap, then it might be most cost effective to place *Image'* in cheaper, low-durability storage and re-run *Grayscale* whenever a storage failure causes *Image'* to be lost. Conversely, if compute is expensive or *Grayscale* has a long run time, it may be more cost effective to place *Image'* in costlier, highly-durable storage to avoid needing to re-run the expensive *Grayscale* action.

In practice, making this tradeoff is challenging. The optimal balance between compute and storage costs depends on multiple factors: (1) the cost to re-create the data (*i.e.*, the time to re-run the action and the cost of computation in dollars per time), (2) the lifetime of the data (how long the data will be needed), (3) the size of the data, (4) the cost of each storage option (in dollars per lifetime per size), and (5) the durability of each storage option (*i.e.*, the probability that data will be lost and need to be re-computed).

Further complicating the decision is the fact that multiple actions may need to be re-run to re-compute data. If *Image"* were to be lost before or while the object-detection action runs, then *Image"* would need to be re-created by re-running the de-noising action. However, if *Image'* was also lost, then the grayscaling action must also be re-run before the de-noising action can re-run to re-create *Image"*.

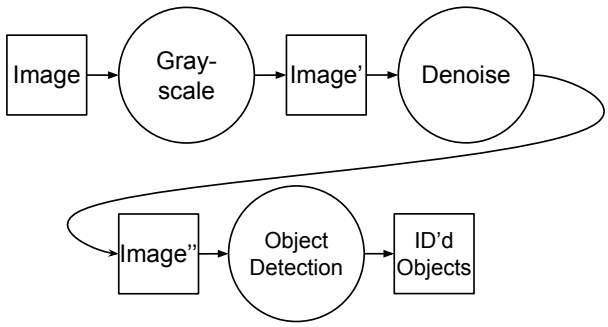


Fig. 1: Example serverless application

This complexity means that there is no single storage option that fits all applications and all environments. It is also not sufficient to simply look at one or two parameters: *e.g.*, the appropriate storage for an application may change when new storage or compute choices become available, or when the input size changes. All factors must be considered to make good storage-placement decisions. We have created a mathematical model, SDCM (Storage Durability Costs Model), that assists in making this decision. SDCM takes as input a Directed Acyclic Graph (DAG) structured serverless application, the size of the input to the application, the cost of compute at each stage, and the available storage options. SDCM then considers the parameters described above and outputs the cheapest storage choice for each stage of the application DAG. We focus on DAG structured serverless applications due to the requirement that we are able to calculate all of the dependencies of a piece of data, in order to enable re-creating that data if lost. Applications structured as a DAG satisfy this requirement.

We show that under a large number of circumstances, low-durability storage is actually more cost effective than higher cost, higher-durability storage.

To avoid placing the burden of dealing with lost data on the developer, we implement an execution system that automatically detects when data has been lost and re-runs the appropriate function(s) to re-create the lost data. It does so in a way that is transparent to the application. This allows developers to take advantage of lower-durability storage without requiring special handling for lost data in their applications. Our system additionally handles making placement decisions for an application's data, using SDCM.

In summary, our contributions are as follows:

- SDCM, a mathematical model that can predict the execution costs of a DAG-structured serverless application.
- An execution system for running DAG-structured serverless applications, capable of automatically recovering in the event of lost data.
- Evaluation of SDCM, showing that we can accurately predict DAG execution costs within 7% of actual costs and reduce storage costs by up to 3×.
- Analysis of several model parameters, showing that under many conditions low-durability storage is more cost effective than high-durability storage.

II. Background & Motivation

a) Data durability: Storage systems employ various techniques to hide low reliability of the underlying storage devices from users. These techniques involve some degree of overhead: the only current way to prevent data loss in the event of a disk failure is by ensuring that the data stored on that disk can be recreated from additional copies elsewhere in the storage cluster. This overhead is incurred in the form of additional disk utilization, slower data access times, additional network traffic, and possibly additional CPU utilization.

Each of these techniques has parameters that dictate the number of device failures it can withstand before data loss occurs. For example, with 3-way replication, data is copied onto three separate disks and can therefore withstand up to two failed disks before data loss. RAID5 uses a single parity disk and can therefore withstand the loss of one device in the RAID array, whereas RAID6 uses two parity disks and can therefore withstand the loss of two disks. When a disk fails, usually a replacement disk will be added and the storage system will redistribute data to ensure the required redundancy is restored. This process is referred to as "rebuilding" the storage array, and can take several hours depending on the size of the disks, size of the array, and the bandwidth dedicated to the rebuild process. If additional failures occur during this rebuild time, then data could be lost.

We can calculate the probability of some number of disk failures during the rebuild time using the annual failure rate of the storage devices. For example, if it takes three hours to rebuild a RAID5 storage system, then we can calculate the probability that two disk failures occur within that three-hour rebuild window. This is typically calculated over some period of time (*e.g.*, the probability that within one year, two failures occur within the same rebuild window). The resulting probability is the probability that over a year, the storage system will lose some amount of data. The quantity of data loss depends on factors such as how the data is distributed across the array and how much progress the rebuild process has made before the subsequent failure occurs.

The inverse of this probability, the probability of data loss *not* occurring over a year, is referred to as the storage system's *durability*. A simple, commonly used formula for calculating durability is roughly $1-(AFR*MTTR)^{\textit{failure tolerance}}$ [19], [9], where AFR is the storage devices' annual failure rate, MTTR is the mean time to repair a failed device, and the failure tolerance is the number of devices that can fail simultaneously without losing data.

As the amount of data grows, it is increasingly important to reduce storage usage where possible. Recent surveys show that rising cloud storage costs are causing companies to search for new ways to store or reduce their data footprint [20], [21].

For most data, the overheads incurred by durability features are unavoidable: losing the data is not an option. However, any ephemeral data passed between the actions in a serverless application has two traits that enable us to avoid the overheads of durability: it is short lived and it can be re-created if lost.

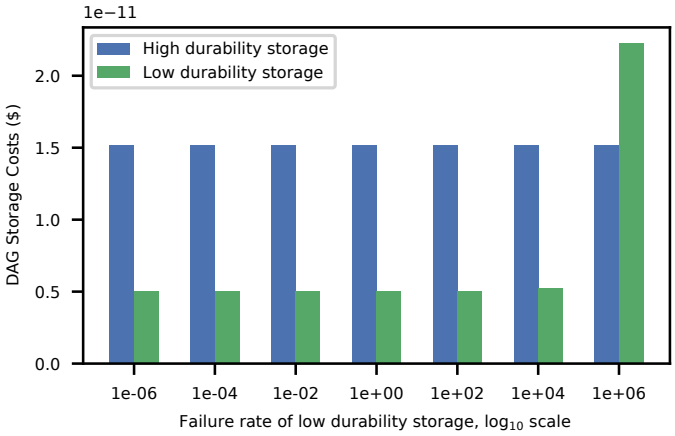


Fig. 2: Storage and re-execution costs for high and low-durability storage. The failure rate for high-durability storage is zero. The failure rate of the low-durability storage varies, increasing along the X-axis. More details about the experiment settings in Section 2.

Ephemeral serverless data is often short lived, with lifetimes on the order of seconds to minutes [22], [23]. This limits the exposure any one piece of data has to the unreliability of the storage system. In other words, the likelihood of a failure (*e.g.*, disk failure, server crash) that results in data loss, occurring during the short time period when the data is needed, is quite low. This is the case even if the storage system is unreliable, and lacks durability features such as replication.

b) Limits of low-durability storage: How much less durable can low-durability storage be while still being more cost effective than higher-durability storage? Figure 2 shows the storage plus re-compute costs of a DAG discussed in further detail in Section V-B. We compare the cost of two storage classes, one with high durability and the other with increasingly high failure rate. We price the low-durability storage at $3\times$ cheaper than the high-durability storage.

We see that the failure rate of the low-durability storage can be as high as 10,000 (bytes lost per hour) before the re-compute costs outstrip the cost savings realized by using lower-durability storage.

c) Handling lost data: Serverless actions are required in most cases to be idempotent [24], meaning that re-running an action multiple times will produce the same data each time. Therefore, in the unlikely chance that data *is* lost while it is still needed, the application can recover by re-running the action that originally created the lost data.

Although it might make economic sense to use lower-durability storage, doing so does introduce some additional complexity. In particular, developers usually operate under the assumption that if they put data into storage, the data will be there later when they need to retrieve it. Using lower-durability storage breaks this assumption, as now the data may be lost between storage and retrieval. Dealing with this possibility requires several actions: (1) identify when an action

No.	Trait	Compatible with SDCM	Suitable for SDCM
1	Data cannot be re-created	✘	✘
2	Unpredictable parameters	✘	✘
3	Latency sensitive	✔	✘
4	DAG structured	✔	✔
5	Uses high-durability storage	✔	✔
6	Large amount of data	✔	✔

TABLE I: Application traits that determine if the application is compatible with SDCM, and can indicate whether SDCM will reduce storage costs.

has failed, (2) recognize that the action has failed due to missing data, (3) determine the action that originally created the now-missing data, (4) re-run that action to re-create the missing data, and finally, (5) re-run the action that failed due to missing data.

Currently, no execution engine or framework for serverless is capable of automatically performing the above steps. However, as we show in this paper, it is possible to develop an execution system that does handle these actions. Such an execution system enables the use of lower-durability storage, without increasing the burden on the application developer.

A. Target Use Cases

SDCM aims to reduce storage costs while executing serverless DAGs. However, not all serverless applications are compatible or good fits for use with SDCM. Table I lists some criteria that help determine whether or not a serverless application can be used with SDCM, and whether or not the application can expect to benefit from using SDCM. We expand on these criteria here:

1. **Data cannot be re-created**: SDCM assumes that data can be re-created by re-running the function that originally created it. If this is not the case, the tradeoff SDCM makes between cheaper storage, lower-durability storage and re-execution costs does not make sense.
2. **Unpredictable parameters:** The inputs to SDCM include data such as the size and lifetimes of data. These inputs can be predicted using prior profiling runs of an application with various sized inputs. However, if the size or lifetime of the data is unpredictable, then knowing these model inputs will be impossible.
3. **Latency sensitive:** An application that has strict latency requirements for all requests may not be suitable for use with SDCM, since there is a chance that a request will encounter missing data that must be re-created. In this case, the request will experience increased latency as it must wait for the data to be re-created.
4. **DAG structured:** SDCM calculates the expected cost of the entire application DAG and chooses the cheapest storage option. If the application is not DAG structured, SDCM will be unable to calculate the costs associated with re-creating data, if doing so will require re-executing parent actions as well. SDCM will still be able to balance re-execution and storage costs for an individual action and the data that it produces, but may underestimate re-creation costs for this reason.
5. **Uses high-durability storage:** If the application is already using low-durability storage, then the opportunity for lowering storage costs even further may be limited. However, if there are tiers of even lower durability, cheaper storage that are cheaper than what is being used currently, then SDCM would have an opportunity to potentially reduce storage costs.
6. **Large amount of data:** The benefit SDCM provides is lower storage costs. If storage costs are a small part of an application's overall execution costs, then the total cost savings enabled by SDCM will also be small. SDCM will still work with these applications, but the benefit may be limited.

In general, applications that are not currently using low-durability storage and process large amounts of data are most likely to see storage cost reductions with SDCM. Examples of these applications might be a genomic processing pipeline [25] or a machine-learning preprocessing pipeline [26], [27]. Examples of applications that would not be suitable for use with SDCM are applications that use an in-memory key-value store (*e.g.*, Redis) for passing data [28], or applications that have strict requirements on response time (*e.g.*, applications that handle user requests [29], [30]).

III. Execution Costs Model

The tradeoff we examine is between the additional cost to store data more durably and the additional cost to re-create the data, should it be lost by lower-durability storage. We developed a mathematical model, SDCM, that balances these two costs, taking into account the probability that data is lost by a less-durable storage system. SDCM optimizes for system level costs, choosing the storage class for each piece of intermediate data created during the execution of a serverless application, that minimizes the total cost to execute the application.

We chose to develop a mathematical model, rather than using a more sophisticated machine-learning- or reinforcement-based model, for several reasons: (1) SDCM provides a closed-form solution to the tradeoff, which enables efficient decision making even in the face of large application DAGs with many storage options, (2) we found that SDCM is sufficient for making good storage-placement decisions and so using a more complicated machine-learning based model was unnecessary, and (3) the amount of data necessary to train a machine-learning model might not be readily available or easy to collect. Still, a machine learning based approach could also be feasible. We leave the investigation of such approaches to future work.

In this section we first describe the terms used by SDCM, then describe how SDCM calculates per-function and per-DAG costs, and finally describe how we use SDCM to minimize end-to-end DAG execution costs.

a) Model definitions: Consider a serverless application structured as a Directed Acyclic Graph (DAG) consisting of several nodes, denoted by $\mathcal{N}$. A node in this graph represents a function that is executed on a serverless platform. Each node, say $n_i \in \mathcal{N}$, has zero (if a root node), one, or multiple parent nodes, denoted by $\text{Pa}(n_i) = \{j : j \text{ being a parent of } n_i\}$. Also, each node n_i has zero (if a leaf node) or more children, denoted by $\text{Ch}(n_i)$. Let n_0 denote the root, and $\{n_\ell : \ell \in \mathcal{L}\}$ denote all the leaves.

Each node n_i is associated with:

- a runtime r in seconds and compute class with cost c \$ per second. The compute cost incurred by executing the node once is $c_{compute} = r \times c$.
- a list of data items produced by the node, X. Each data item has an associated lifetime x_l in seconds and size x_s in bytes.
- a set of available storage classes, denoted by $\mathcal{D}$, and the set of storage classes chosen for each of the data produced by the node, denoted by d_{n_i}. Each storage class in the set d_{n_i} has a cost $c_{\text{d}_{n_i}}$ \$ per byte-second and a probability of failure, explained below.
- i-specific cost associated with each set of storage class choices, calculated as $c_{storage} = \sum_{x \in X} x_l \times x_s \times c_{\text{d}_{n_i}}$.
- a base cost, assuming no failures, denoted by $C(n_i, \text{d})$ and calculated as $c_{compute} + c_{storage}$.
- a failure probability associated with a set of storage class choices $\text{d} \in \mathcal{D}$, denoted by p_{d}.
- an expected cost $\mathbb{E}(cost())$, calculated using the base cost and probability of failure as explained below. The expected cost for a particular node and a choice of storage classes is expressed as $\mathbb{E}(cost(n_i, \text{d}_{n_i}))$.

Here, a failure means that some data produced by the node has been lost as a result of a failure of the storage system. Since the data has been lost, subsequent functions that rely on the lost data cannot execute, and so the node must be re-executed to re-create the lost data. We assume that the re-execution recurs until it succeeds. We refer to the probability failure (*i.e.*, the probability that data is lost as a result of a storage system failure at some point during the lifetime of the data) as $\mathbb{P}_f$ and the probability of success as $\mathbb{P}_s$.

We define the system-level cost to be the expected total costs of all nodes to successfully perform one execution.

b) Calculating DAG execution costs: We use a top-to-bottom procedure to recursively calculate the system-level cost. We first evaluate the root node n_0 and calculate the expected cost for executing this node:

$$\begin{aligned}&\mathbb{E}(cost(n_0, \text{d}_{n_0})) \\ &= \mathbb{P}_s \times C(n_0, \text{d}_{n_0}) + \mathbb{P}_f \times \Big\{ C(n_0, \text{d}_{n_0}) + \mathbb{E}(cost(n_0, \text{d}_{n_0})) \Big\}\end{aligned} \tag{1}$$

Solving the above equation (1) leads to

$$\mathbb{E}(cost(n_0, \text{d}_{n_0})) = \frac{C(n_0, \text{d}_{n_0})}{1 - \mathbb{P}_f} = \frac{C(n_0, \text{d}_{n_0})}{1 - p_{\text{d}_0}} \tag{2}$$

If we assume that each node keeps its data until the end of the DAG's execution, then Equation 2 can also be used for calculating the expected cost of a generic node.

c) Correlated failures: If data for one node has been lost, it might be likely that data for other nodes has been lost as well. In the worst case, we might assume that if data for one node has been lost, then the entire DAG's data has been lost. In this worst-case scenario, when calculating the expected cost of a node, we must consider the cost of re-running the entire DAG up until that node, in order to re-create the node's data.

For a generic node, denoted by n, recall that its parents are $\text{Pa}(n) = \{j : j \text{ being a parent of } n\}$. We define $\text{Pa}^{(r)}(n) = \text{Pa}(\cdots \text{Pa}(n))$ (r times composition) as the ancestor of n up to r levels. For example, if n is a child of n_0, we have $\text{Pa}^{(1)}(n) = n_0$. By a similar argument as in Equation 2, we have

$$\begin{aligned}\mathbb{E}(cost(n, \text{d}_n)) &= \mathbb{P}_s \times C(n, \text{d}_n) \\ &\quad + \mathbb{P}_f \times \Big\{ C(n, \text{d}_n) + \mathbb{E}(cost(n, \text{d}_n)) \\ &\quad + \mathbb{E}\big(cost(\text{Pa}(n), \text{d}_{\text{Pa}^{(1)}(n)})\big) \Big\}\end{aligned} \tag{3}$$

Solving the above leads to

$$\begin{aligned}&\mathbb{E}(cost(n, \text{d}_n) \\ &= \frac{\Big\{ C(n, \text{d}_n) + \mathbb{P}_f \mathbb{E}\big(cost(\text{Pa}(n), \text{d}_{\text{Pa}^{(1)}(n)})\big) \Big\}}{1 - \mathbb{P}_f} \\ &= \frac{\Big\{ C(n, \text{d}_n) + p_{\text{d}_n} \mathbb{E}\big(cost(\text{Pa}(n), \text{d}_{\text{Pa}^{(1)}(n)})\big) \Big\}}{1 - p_{\text{d}_n}}\end{aligned} \tag{4}$$

where $\mathbb{E}\big(cost(\text{Pa}(n), \text{d}_{\text{Pa}^{(1)}(n)})\big)$ is the sum of the expected costs of all the immediate parents of node n, namely

$$\mathbb{E}\big(cost(\text{Pa}(n), \text{d}_{\text{Pa}^{(1)}(n)})\big) = \mathbb{E}\Big(\sum_{j \in \text{Pa}_j(n)} \text{cost}(j, \text{d}_j) \Big). \tag{5}$$

This establishes a *recursion* between $\mathbb{E}(cost(n, \text{d}_n))$ and $\mathbb{E}(cost(\text{Pa}(n), \text{d}_{\text{Pa}(n)}))$. Using the recursion in Equation 4 and the expected cost of the root node (Equation 1), we can calculate $\mathbb{E}(cost(n, \text{d}_n))$ for any particular node n.

Our implementation of SDCM allows a user to choose between using Equations 2 and 4.

d) Minimizing DAG execution costs: The expected system-level cost is

$$\text{System-cost}(\text{d}_n, n \in \mathcal{N}) = \sum_{n \in \mathcal{N}} \mathbb{E}(cost(n, \text{d}_n)). \tag{6}$$

Based on this formula for system-level cost, we can evaluate the cost associated with any set of storage placement decisions, namely d_n for all $n \in \mathcal{N}$. We can then make storage-placement decisions to minimize the cost to execute the serverless application. In other words, we want to solve

$$\min_{\text{d}_n \in \mathcal{D}, n \in \mathcal{N}} \text{System-cost}(\text{d}_n, n \in \mathcal{N}). \tag{7}$$

This is in general a challenging problem because (i) the decisions made by different nodes are interconnected, and (ii) a node may not be able to infer the optimal decision by excluding other nodes (*e.g.*, parent or children nodes). For the non-correlated case (Equation 2) this is $\mathcal{O}(cn)$, and for the correlated case (Equation 4) it is $\mathcal{O}(c^n)$, where c is the cost to calculate a single node cost and n is the number of nodes in a DAG. In our experience both c and n tend to be low, and this value takes under a minute to calculate.

We provide a greedy algorithm that will iteratively reduce the system cost and converge to a local minimum. We use a node-wise coordinate-decent procedure to optimize each node's decision, fixing other nodes' decisions at each iteration. More specifically, we solve

$$\min_{\mathrm{d}_{n'} \in \mathcal{D}} \text{System-cost}(\mathrm{d}_n, n \in \mathcal{N}) \tag{8}$$

by fixing $\mathrm{d}_n, n \in \mathcal{N} - \{n'\}$, and iterating over all $n' \in \mathcal{N}$. To solve Equation 8, we note that the node n' will only be invoked by its descendants instead of ancestors, and that the number of invoking n' is irrelevant with a node's own decision. So, solving Equation 8 is equivalent to minimizing the cost of a single successful execution of n', namely

$$\min_{\mathrm{d}_{n'} \in \mathcal{D}} \mathbb{E}(cost(n', \mathrm{d}_{n'})) \tag{9}$$

which can be easily solved by evaluating $\mathbb{E}(cost(n', \mathrm{d}))$ for each $\mathrm{d} \in \mathcal{D}$ and choosing the one that minimizes it.

A. Future Model Extensions

There are several ways in which our model could be enhanced or extended, which we hope to address in future work. We describe the two most significant extensions here. First, our model currently does not consider the performance differences that might exist between storage classes. Lower-durability storage is likely to have higher performance [25] as there are no overheads from replication or erasure coding. Depending on the data usage of an application, these performance differences can result in significant differences in runtime. Lower runtime results in lower compute costs, as well as lower storage costs and a lower likelihood of data loss. All of these factors can influence the model's placement decisions.

Second, currently our model simply selects the storage placements that result in the lowest DAG execution costs. However, in some cases this might not be acceptable. For example, applications might have real-time constraints that require a response within a certain deadline. For these applications, occasionally re-running functions due to lost data might add an unacceptable amount of execution time. These constraints would work in tandem with the aforementioned performance extensions. Higher performance, lower-durability storage may be able to meet runtime constraints even with periodic re-executions. Conversely, lower-durability storage that is not higher performance may not meet these constraints.

Finally, there are other ways in which SDCM could be used besides for placing data. For instance, it could be used to decide when to delete data: if the predicted cost to store the data is more than the predicted cost to re-create the data, then the data should be deleted and re-created when needed. SDCM can help make this decision.

IV. Design & Implementation

SDCM relies on tracking several application metrics, such as the runtime of each action for a given input size and the size and lifetime of data generated by each action. In addition, we assume that it is possible to re-create data lost by a storage system by re-running the action that created it. Existing serverless compute platforms do not support tracking all of the metrics we require, and have no way of tracking which data is created by which specific invocation of an action. This makes it impossible to automatically re-generate data that has been lost, a key capability necessary to using lower-durability storage.

We therefore developed an execution system that handles both collecting the metrics that SDCM requires as well as handling re-running actions to re-create lost data. We designed our execution system to run serverless applications on OpenWhisk, an open-source serverless platform. We installed OpenWhisk on a Kubernetes cluster. For storage we use a Ceph Object Store, which we configured to have multiple storage classes with different degrees of replication. Figure 3 depicts the components that are typical of a serverless cluster in blue. We extended these with several additional components and scripts, colored in green, in Figure 3. These components include mitmproxy [31] ❻, MongoDB [32] ❸, and additional scripts. In total, our execution system consists of around 400 lines of Go and 1,535 lines of Python.

a) Running applications: The DAG structured applications run on our execution system consist of a series of OpenWhisk actions, with one action per DAG step—although a step may contain several instances of the same action that run in parallel. Users start by defining their DAG in YAML ❶, specifying each step. The specification for each step in the DAG includes the OpenWhisk action and arguments used by that step, as well as the input and output data consumed and produced by that step.

Our script then converts this YAML specification into a runnable shell script ❷. Based on the inputs and outputs of each DAG step, the script orders the OpenWhisk action invocations and runs actions in parallel when possible. In addition to invoking actions, the shell script creates the storage bucket used by the application. After the application finishes, it will log the lifetime and size of all objects in the bucket in our metrics and tracking database ❸.

OpenWhisk actions are not invoked directly, but are instead run via a Python runner ❹. This runner is responsible for logging metrics such as the runtime of each action, as well as for passing additional arguments to each OpenWhisk action. These arguments include the size of the input to the DAG and a unique identifier for the specific action invocation.

These arguments are consumed by a library included by each OpenWhisk action ❺. This library also contains a function for uploading data to an object store, which adds

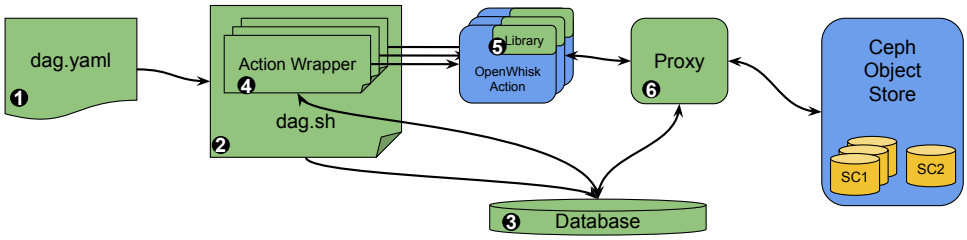


Fig. 3: Design of execution system

Name	Durability scheme	Cost ($ / GB-hour)	NOMDL$_{1h}$
SC1	3-way replication	3.19×10^{-5}	0
SC2	None	1.06×10^{-5}	9.8×10^{-9}

TABLE II: Hypothetical storage classes

additional headers that specify information such as the action associated with the object being uploaded.

We use a proxy ❻ between the application and the Ceph Object Store to track data accesses. We track when an object is created and when it is last accessed, what action created an object, and when an action tries to access a missing object (*i.e.*, when an action's request receives a 404 response code from the Ceph Object Store).

The proxy also has the role of adding storage class choice to the object PUT request. It looks up the storage class chosen for the object in the tracking database, and then specifies this storage class using the `X-Amz-Storage-Class` HTTP header. This header is the standard way of specifying storage class placement for an object, used by both Amazon S3 and by Ceph Object Store [33], [34].

b) Storage class selection: The application is first run in a profiling mode to capture runtime and data lifetime, as well as size metrics. A script then takes these metrics and selects a storage class for each of the objects produced by the DAG, using Equation 9 described in Section III. The script saves these selections in the database ❸.

c) Failure handling: If an action submits a GET request for an object which returns a 404 code (*i.e.*, the object is missing), the proxy ❻ will record in the database the action and the object that was missing. The Python runner ❹ will see that the action failed, and will check the database if the proxy recorded any missing objects for the failed action. Upon finding a missing object, the Python runner looks up the action that created the object (recorded by the proxy) and re-runs that action to re-create the data. Finally, the Python runner will re-run any failed actions that depended on the missing data.

A. Hypothetical storage classes

Storage systems that are available in the cloud currently are highly-durable and come with the associated cost premiums. Therefore, using SDCM to select between existing storage options would have limited utility. Instead, we create hypothetical storage classes that have lower durability and costs. We then use SDCM to select between these lower-durability storage classes and more traditional, high-durability storage classes. Our hypothetical storage classes are listed in Table II and our methodology for calculating their costs and failure rates are described below. We define two hypothetical storage classes with two levels of durability: one that uses 3-way replication, and another that uses no durability features (*i.e.*, no replication or erasure coding). Note that other classes can be easily created and supported by SDCM.

a) Cost: We base the prices of our hypothetical storage classes on actual cloud storage pricing. Our durable storage class is priced at the price of Amazon's S3 storage (as of January 2024). Our low-durability storage class is priced at a third of this price, based on the cost savings achieved by using unreplicated storage versus using triply replicated storage.

b) Failure rate: Calculating the failure rate of a storage configuration is a complex task. Markov models are frequently used to model the state of the storage system as disks fail and are repaired [35], [36]. These Markov models are then used calculate statistics such as the mean time to data loss. However, these models often suffer from invalid assumptions and other inaccuracies [37], [38]. We instead use a simulator developed by Greenan [37] that aims to address the issues common to Markovian mean time to data loss analyses.

The simulator calculates the metric NOMDL, or Normalized Magnitude of Data Loss. This metric is expressed as bytes lost per storage system capacity for a certain time period, which for us was one hour. NOMDL is inversely related to durability: a storage system with a high NOMDL has low durability, and therefore, a higher likelihood of losing data.

SDCM uses the calculated NOMDL values when considering where to place a piece of data created by an application. SDCM calculates the data's anticipated size and lifetime; then, for each storage class's pre-computed NOMDL value, SDCM calculates the probability that the data will be lost during its lifetime. The data's size is normalized to the size of the cluster, just like the NOMDL value was. The data's lifetime is used to scale the NOMDL value: for example, if the expected lifetime is 30 minutes, then the expected data lost during that time is equal to half of the pre-computed NOMDL, since that was pre-computed based on a one hour mission time.

V. Evaluation

In this section, we demonstrate using SDCM to make storage placement decisions and predict DAG execution costs.

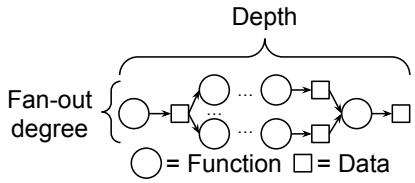


Fig. 4: DAG Used for Accuracy Evaluation

For all experiments, we use our execution system that handles automatically re-creating lost data.

In Section V-A we evaluate SDCM's ability to make accurate predictions of DAG execution costs. In Section V-B we explore how changing model and environmental parameters impacts SDCM's decisions. In Section V-C we provide an analysis of the impact that re-executing actions has on overall DAG runtimes. Finally, in Section V-D, we demonstrate using SDCM and an execution system for running two real world applications.

A. Model Accuracy

We evaluated SDCM by executing the DAG depicted in Figure 4. Each function in the DAG simply makes a copy of its input data, then exits. We evaluate several factors, including DAG depth, fan-out degree, and input size: three different fan-out degrees, five depths, and three input sizes for a total of 45 different DAGs. We measured the actual runtime of each function in the application, as well as the size and lifetime of each piece of data produced by the application. From these measurements and by choosing compute and storage costs, we can calculate the cost to execute the DAG. We compare this actual value with the cost predicted by SDCM.

Since SDCM accounts for failures and considers the cost to recompute data, our evaluation must also include failures. We simulate data failures by randomly deleting data between each stage of the DAG. This forces subsequent stages that rely on the deleted data to fail. Our execution system (see Section IV) identifies that a function failed due to missing data, identifies the function responsible for originally creating that data, and re-runs it. After successfully re-creating the data, our execution system re-runs the original function that failed due to the missing data. We delete data based on the NOMDL value of the storage where the data has been placed. Since even our non-durable storage class has a fairly low rate of data loss, for evaluation purposes we artificially inflated the data loss rate to 10,000 bytes-per-hour for a 100-disk cluster. We do this only in this subsection, for the purpose of demonstrating the accuracy of SDCM.

We ran each of the 45 DAGs at least ten times to ensure that the variation of runtimes across each DAG execution is low. In all but two cases the runtime variation was less than 10%. In those two cases, it was 12.8% and 11.1%. The high variation in these cases is due to data loss resulting in functions needing to be re-run. This produces a bi-modal distribution of runtimes, with some DAGs executing entirely with no data loss and others with re-executions adding to their overall runtime. The two most extreme cases are shown in Figure 5. On top is the DAG with a depth of three and fan out of one, with a 1024MB input. The variation across runtimes for this DAG was 12.8%, and we see that this is due to a small number of runs encountering errors requiring re-runs, resulting in a much higher runtime. The situation is reversed for the lower plot: in this case the DAG depth was 4, fan-out was ten, and the input was 1024MB. The runtimes for this DAG had a variation of 11.1%, but now this high variance is due to most runs encountering some errors, and a small few runs not encountering any errors.

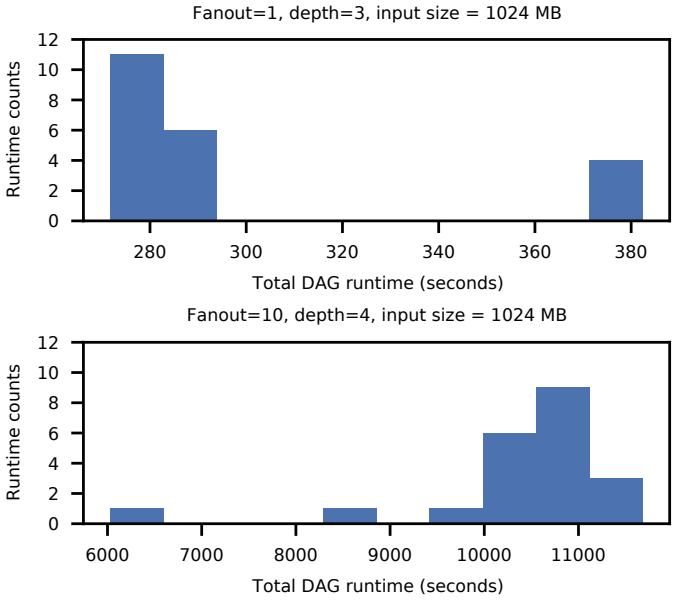


Fig. 5: Histograms of DAG runtimes for the two DAGs with highest runtime variance. The bimodal distribution of runtimes is a result of some DAG runs encountering errors and other runs encountering no errors.

We ran SDCM with the assumption that all data is placed in our high failure rate (NOMDL_{1h} = 10,000) storage. We use the price of SC2 from Section IV-A, $\$1.06 \times 10^{-5}$ per GB-hour, as the price for this storage. For compute costs, we use $\$1.67 \times 10^{-5}$ per second, roughly equivalent to the per-second cost of a 1GB AWS Lambda instance. We found, however, that storage cost and compute costs do not impact SDCM's accuracy. Higher storage and compute costs would amplify errors in predictions of runtime, data size, data lifetime, and data loss. These errors are small enough that changes in the underlying compute and storage costs do not significantly impact the accuracy of the whole-DAG cost prediction.

Figure 6 shows results for 1MB, 128MB, and 1024MB sized inputs. Each figure shows the accuracy as a percentage on the Y-axis. The figures plot the accuracy as a function of the total number of DAG stages, DAG depth (3 through 7 stages) and fan out degree (1, 5, and 10).

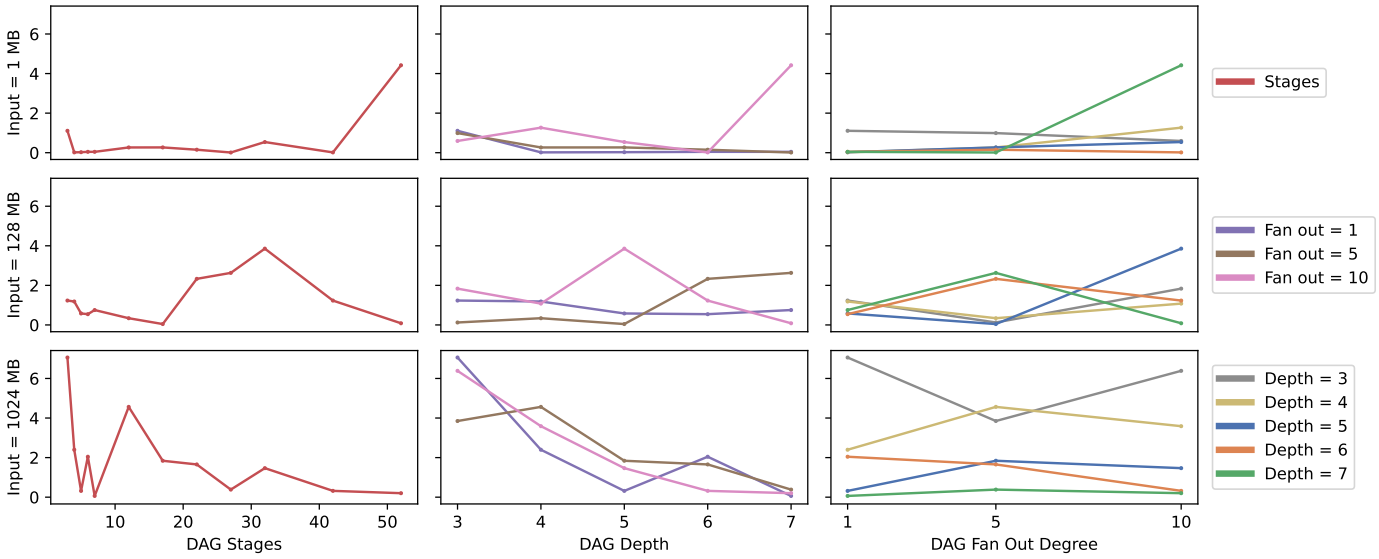


Fig. 6: Accuracy of SDCM for three different input sizes.

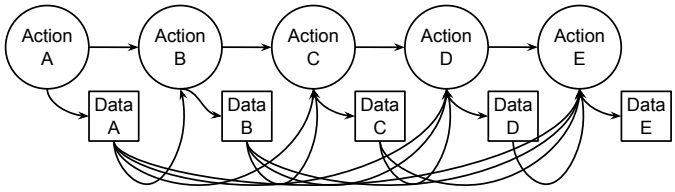


Fig. 7: DAG used for model parameters exploration. The data produced by each step of the DAG was used by each subsequent step.

In all cases, we find that SDCM was accurate within 7% of the actual DAG execution costs. For 128MB and 1MB input sizes we find that SDCM is within ±5% of actual DAG execution costs. This is better than similar studies, which predicted DAG execution properties to within 15% [24], [39].

Note that this accuracy is with respect to the *average* DAG execution costs, and *includes the cost of re-executions due to lost data amortized across all DAG executions*. Therefore, when comparing SDCM's prediction to any individual DAG execution, SDCM's prediction will be either slightly higher than the actual cost (if there were no data losses encountered during the DAG's execution) or lower than the actual cost (if the DAG's execution included re-executions due to data loss).

B. Model Parameters

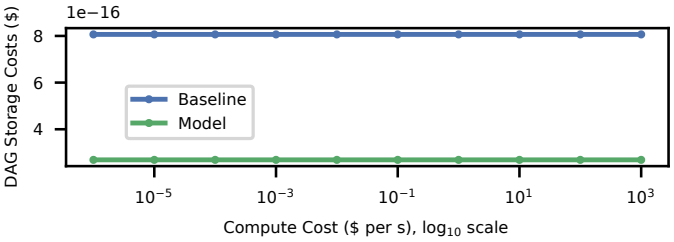
We now describe the impact that different parameters have on SDCM's storage class decisions. We use a simple five-stage DAG for exploring these parameters, shown in Figure 7. Each step produced 1MB of output, and each step's output was used by each subsequent step (so the lifetime of data A is highest, and the lifetime of data E is lowest). The DAG's actions simply make a copy of the input data, and ran for five seconds. SDCM chooses between the two storage classes described in Section IV-A.

a) Compute cost: We first look at how the cost of compute impacts storage class placement decisions. Figure 8a shows the storage component of the total DAG execution cost (Y-axis), versus the cost of compute (X-axis). The storage component includes both the cost of storage for data produced by the DAG, as well as the additional cost required to re-run data in the event of data loss. "Model" is this value as calculated by SDCM, using storage classes that SDCM determined given the lowest total DAG execution cost. It includes the additional cost predicted by SDCM to be incurred as a result of losing data and needing to re-run part(s) of the DAG.

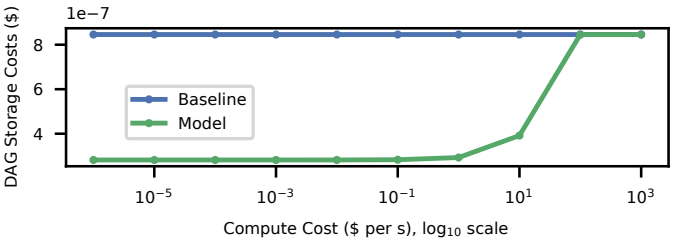
"Baseline" is the storage component of the total cost to execute the DAG if the most-durable storage class is used for all the DAG's data.

On the low end of the X axis is cheap compute, roughly equivalent to the dollar-per-second cost of an EC2 Spot Instance virtual machine. The high end is more expensive than any available cloud compute. Intuitively, at these higher compute costs, we might expect SDCM to choose more-durable storage, as the cost to re-compute data becomes more prohibitive. However, for all compute costs seen here, SDCM chooses the cheaper, less-durable storage for the DAG's data (SC2). We find that this is because the high compute cost is offset by the extremely low probability of losing data: for the data produced by the DAG, the probability of loss is calculated to be between 1.9×10^{-21} and 7.4×10^{-21}.

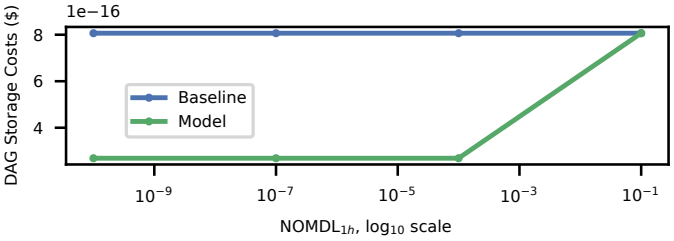
Figure 8b shows what happens if we instead have 1TB of data produced at each stage, the data's lifetime was 1,000× longer, and SC2's NOMDL_{1h} was increased by 1,000×. The probability of data loss now ranges from 1.7×10^{-10} to 7.8×10^{-10}. This makes SDCM more sensitive to increases in compute cost, and we do indeed see that as the compute



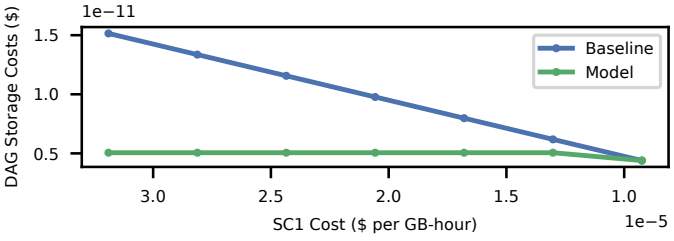
(a) Storage costs during DAG execution vs cost of compute. Note the log scale on the X-axis.



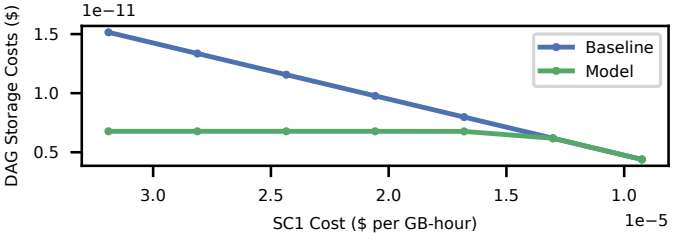
(b) Storage costs during DAG execution vs. cost of compute, if intermediate data is larger, longer lived, and storage is less reliable. Note the log scale on the X-axis.



(c) Storage costs during DAG execution vs NOMDL$_{1h}$. Note the log scale on the X-axis.



(d) Storage costs during DAG execution vs price of SC1. Note the reversed (decreasing) X-axis.



(e) Storage costs during DAG execution vs price of SC1, with high NOMDL$_{1h}$ (10,000). Note the reversed (decreasing) X-axis.

Fig. 8: Impact of different parameters on SDCM storage class choice.

cost increases, SDCM eventually selects more-durable storage to avoid the cost of re-compute.

b) NOMDL: Next, we look at the impact a storage class's NOMDL value has on SDCM's decisions. We keep the NOMDL of the more-durable storage class (SC1) the same, but increase the NOMDL value of the cheaper but less-durable storage class (SC2). Figure 8c shows the storage component of the DAG execution cost on the Y axis with the NOMDL value of SC2 on the X axis.

We expect that as NOMDL value increases, eventually the cost of needing to frequently re-create lost data will outweigh the cost savings from using cheaper storage. Indeed, once the NOMDL value reaches 10^{-1}, SDCM switches from using SC2 to SC1 for all of the DAG's data.

c) Storage price: Finally, we evaluate what it would take for a more-durable storage class to be cost-competitive with lower-durable storage. Figure 8d shows SDCM and baseline costs as we decrease the cost of SC1 (*i.e.*, our more-durable storage class). The baseline cost always uses SC1, whereas SDCM cost chooses between SC1 and SC2 to minimize DAG execution costs. We see therefore the baseline cost decrease as the cost of SC1 decreases. However, only once the cost of SC1 reduces lower than the cost of SC2 does SDCM choose to use SC1. In other words, the cost of SC1 needs to be reduced to that of SC2 in order for it to be cost-competitive with SC2. This is due to the very low probability of failure, between 1.9×10^{-21} and 7.4×10^{-21}.

If we increase the NOMDL$_{1h}$ of SC2 to 10,000, then the probability of failure (and therefore, the cost due to re-executions) increases. This works to increase the price where SC1 becomes more cost-effective than SC2, so that SC1 can be slightly more expensive than SC2 (1.3×10^{-5} vs 1.06×10^{-5} per GB-hour) and still be more cost effective overall. Given that cloud storage prices have been increasing, not decreasing [40], [41], it seems unlikely that highly-durable cloud storage will be able to compete on price with lower-durability storage any time soon.

C. Latency Analysis

Re-running actions to re-create data incurs not only an additional compute cost, but also adds additional latency. This additional latency is amortized across many application runs, since even with low-durability storage, failures (and therefore re-runs) are infrequent. Still, even if the costs make sense (*i.e.*, the additional compute cost is smaller than the cost savings from using lower-durability storage), for some applications the additional latency will be unacceptable. This is captured in Trait 3 in Table I, which says that SDCM may not be suitable for use with applications that are highly latency sensitive.

Here, we conduct a short analysis of that additional latency. We use the DAG depicted in Figure 7 with fixed runtime for each action and fixed sizes for the data created by each action. We evaluate with two configurations: in the first ("Configuration 1"), the actions each execute for 60 seconds and produce 100MB of output. In the second ("Configuration 2"), each action runs for one hour and produces 1TB of data.

Config	App	NOMDL$_{1h}$	Input size (MB)	Correlated failures?
A	Video transcoding	10,000	99	No
B	Video transcoding	100,000	99	No
C	Montage 0.25	10,000	16	No
D	Montage 0.25	10,000	16	Yes
E	Montage 0.25	100,000	16	Yes
F	Montage 1.0	10,000	153	No
G	Montage 1.0	100,000	153	No
H	Montage 1.0	100,000	153	Yes

TABLE III: Parameters used for applications in case study

We evaluate each configuration with and without correlated failures. We increase the NOMDL$_{1h}$ of the storage system used for storing the data, and calculate the probability of losing data (and therefore, the probability of needing to re-run a stage of the DAG).

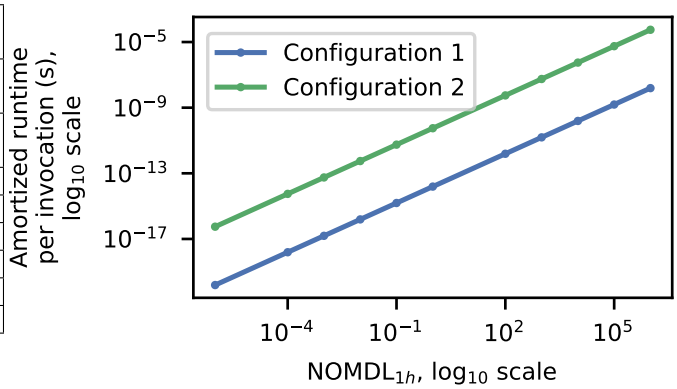
If an action needs to be re-run, the overall runtime of the DAG will increase by up to 2× the runtime of a single action. This is because in the worst case, an action will run until just before completion before the data it is consuming is lost by the storage system. One action will need to be re-run to re-create the data, and then the consumer action will need to be re-run since it failed to complete on its initial run. In the case of correlated failures, the entire DAG will need to be re-run. Again, the worse case is that data is lost right before completion, meaning that the entire DAG essentially needs to be run twice. For Configuration 1, in the correlated case, this means that a DAG invocation that encounters an error will run for up to $2 \times 60 \times 5 = 600$ seconds, rather than 300 seconds in the usual case. For Configuration 2, an invocation encountering an error can run for up to 10 hours, compared to 5 in the non-error case.

If the application encounters lost data due to a storage system failure, that specific run will take significantly longer to complete. However, this case is fairly rare, and amortized across all runs the additional runtime is quite small. Figure 9 shows the additional time spent re-running actions as a result of failure, amortized across the predicted number of DAG invocations between failures. Figure 9a shows this additional time in seconds, and Figure 9b shows this time as a percentage of the total runtime for a single invocation of the DAG. We show here the correlated failure case; the non-correlated case will have an even lower amortized time cost.

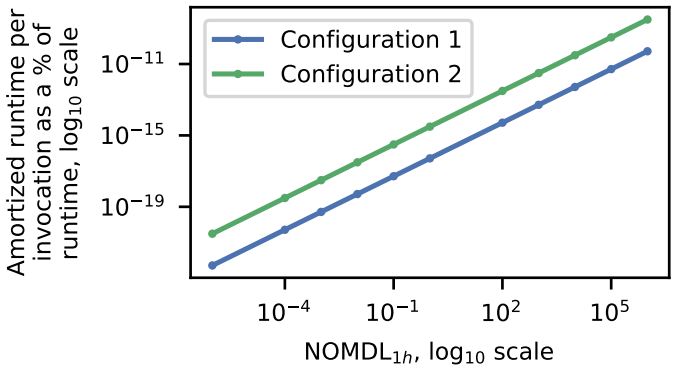
Even in the case where the storage system's failure rate is unrealistically high (NOMDL$_{1h}$ = 10^5), the additional time is only 1.6×10^{-8} seconds and 5.6×10^{-5} seconds per DAG invocation for Configurations 1 and 2, respectively. This corresponds to just $5.1 \times 10^{-11}\%$ and $3.1 \times 10^{-9}\%$ of the runtime for a single DAG invocation, for Configurations 1 and 2, respectively.

D. Case Studies

In this section we demonstrate using SDCM and execution system to run real world applications. We chose applications that satisfied the criteria described in Table I: 1) the intermediate data could be reproduced by re-executing the creating action, 2) the model parameters for each application (*e.g.*, runtime of each action, the size and lifetime of data produced by each action) are predictable after profiling, 3) the applications are not latency sensitive, 4) are DAG structured, 5) were written originally using durable storage, and 6) process a non-trivial amount of data.



(a) Additional time, amortized across the expected number of DAG invocations between failures. Note the $\log_{10}$ scale on both X- and Y-axes.



(b) Additional time, amortized across the expected number of DAG invocations between failures, as a percentage of a single DAG's runtime. Note the $\log_{10}$ scale on both X- and Y-axes.

Fig. 9: Additional time spent re-executing to re-create lost data. In Configuration 1 each action runs for 60 seconds and produces 100 MB of data. In Configuration 2, each action runs for one hour and produces 1TB of data.

Table III contains the parameters used for each of the experiments. We used the storage classes defined in Section IV-A, and $0.0068 per second as our compute cost. Figure 12 graphs the storage costs (including re-execution costs) for SDCM's storage class selections, compared with the baseline of using highly-durable storage for all data.

a) Video Transcoding: We implement a three stage workflow, using FFmpeg [42] to transcode a video to a new resolution. Our workflow is depicted in Figure 10 and consists of a split stage, a parallel transcode stage, and finally a combine stage. In total, the workflow contains 12 functions

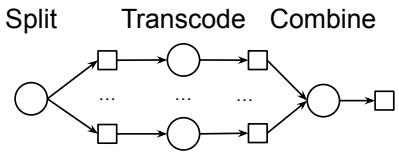
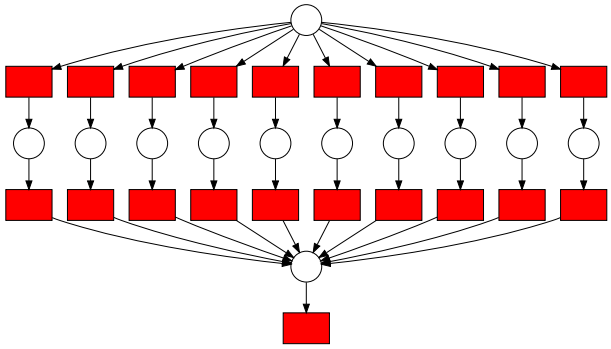
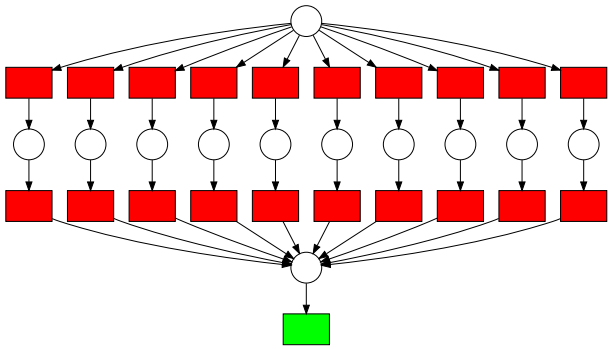


Fig. 10: Video transcoding DAG



(a) Configuration A



(b) Configuration B

Fig. 11: Video transcoding storage class selections. Green is SC1, red is SC2. See Table III for configuration details.

and generates 284 MB of intermediate data.

We run with two configurations (A and B), differing only in that configuration B has a higher NOMDL$_{1h}$ than configuration A. Figure 11 depicts the storage class decisions made by SDCM, with red boxes indicating data stored in SC2 (low-durability, cheap storage) and green boxes indicating data stored in SC1 (high-durability, more expensive storage).

We see that as NOMDL$_{1h}$ increases from configuration A to configuration B, SDCM chooses more-durable storage for the final output. For configuration A, the baseline storage costs are 2.98× SDCM's, and 2.94× more for configuration B.

b) Montage: Montage [43] is a toolkit for processing and stitching together astronomical images. We port the workflow implemented for HyperFlow [44], [45] to run on our execution

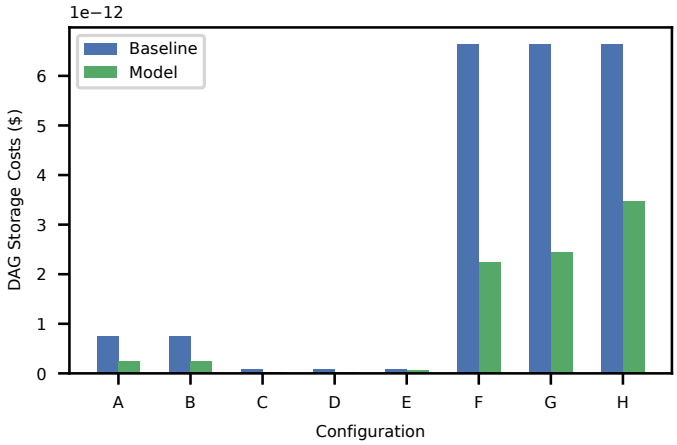


Fig. 12: Storage component of DAG costs for applications used in case study. See Table III for configuration specifics.

system. We run two sizes of Montage workflow, 0.25 and 1.0, with configurations listed in Table III. Figure 13 shows the structure of the workflow. The Montage 0.25 workflow contains 43 total functions and generates 185MB of intermediate data, and the Montage 1.0 workflow contains 469 functions and generates 1,828MB of intermediate data.

Figure 14 shows the storage class selections for the Montage 0.25 workflow. The results for the Montage 1.0 workflow are too large to be included.

As described in Section III, correlated failures makes the assumption that any data loss will require re-execution of the entire DAG. Therefore, It has the effect of increasing the cost of recovering lost data, especially as we progress further in the DAG: the further along we are, the more work will need to be re-run. Indeed, we see this as we enable correlated failures in configuration D vs. configuration C: SDCM selects the more-durable storage class SC1 for data produced towards the end of the DAG.

This effect is amplified by increasing NOMDL$_{1h}$, as seen in configuration E vs. D. Here SDCM selects SC1 for even more data, again all data towards the end of the DAG.

For the Montage 0.25 workflow, the baseline storage costs are 2.99×, 2.65×, and 1.58× higher than SDCM costs for configurations C, D, and E, respectively. For the Montage 1.0 workflow, the baseline costs are 2.97×, 2.72×, and 1.92× higher than SDCM costs for configurations F, G, and H, respectively.

VI. Related Work

a) Storage and data exchange for serverless: There has been a large amount of recent work on data exchange for serverless [46], [47], [24], [48], [49], [50], [25], [51], [52]. Most of this work focuses on improving the performance of data transfer in a serverless environment. To do so, Pocket [46] and Locus [47] utilize a mix of slower, cheaper storage and faster, more expensive storage. The faster, more expensive storage is memory based and does not utilize durability

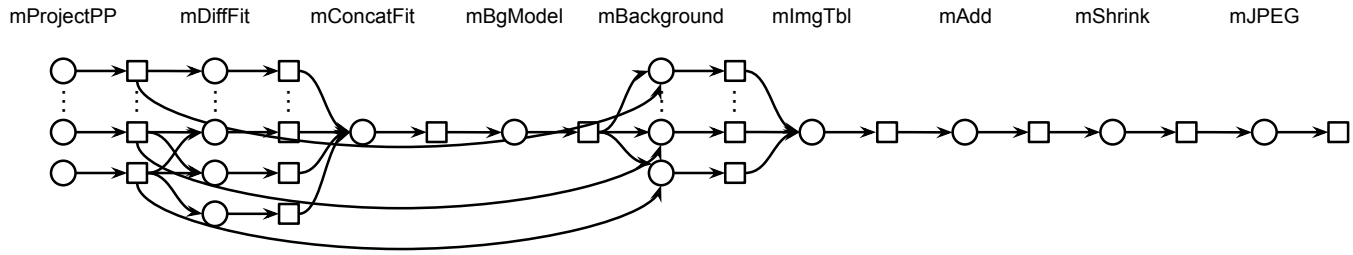
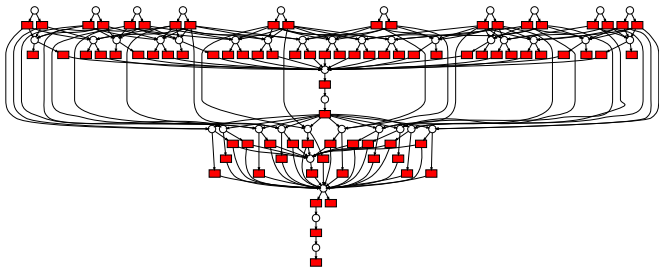
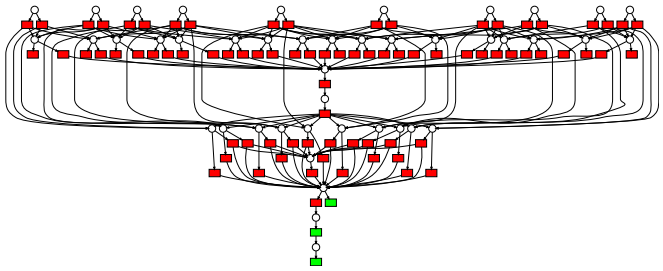


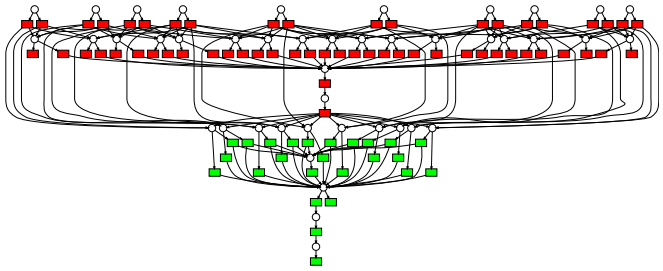
Fig. 13: Montage DAG



(a) Configuration C



(b) Configuration D



(c) Configuration E

Fig. 14: Montage 0.25 storage class selections. Green is SC1, red is SC2. See Table III for configuration details.

features such as replication. These works make the argument that *all* serverless data is short lived and can be re-created, and therefore such volatile storage is suitable for serverless data. However, they simply assume that lower-durability storage is acceptable, and do not consider the cases where such storage may be less cost effective compared to durable storage. Also, they do not explore how data loss might be handled by the application.

FuncStore [52] aims to reduce resource waste by deleting objects when they are no longer needed. To do so, they analyze an application's DAG and use a machine-learning model to determine the anticipated lifetime of each object. We also predict the lifecycle of objects created by applications, but we were able to achieve accurate results using profiling and linear interpolation. Note that because SDCM inherently supports the regeneration of data, mis-predicting the lifecycle of data is not much of a concern as it is with FuncStore, which—like many other related projects—have no mechanism for re-creating lost data. If more accurate lifecycle prediction were needed, we could also adopt the lifecycle-prediction approach used by FuncStore.

Projects such as SONIC [24], SAND [49], Wukong [50], and Cloudburst [51] accelerate data transfer by passing data directly among functions running on the same host. Not all data is capable of being transferred in this way: for instance scheduling constraints may force functions to be run on different hosts, making this data transfer method not possible.

Techniques such as compression [53] and de-duplication [54] can be used to reduce the size of data, and therefore, storage costs. We note that SDCM is not incompatible with these techniques, and the methods SDCM uses to reduce storage costs can be used in conjunction with these other techniques. In fact, many of the data-transfer and data-reduction techniques discussed here, such as as Pocket [46], Wukong [50], and FuncStore [52], could be used together with SDCM to further reduce storage costs. Any technique that uses an intermediate data store to transfer data can be used with SDCM to place data in that intermediate data store at an appropriate, cost-optimal durability level.

b) Lower-durability storage: Amazon previously offered a reduced-durability storage class for its S3 object storage service [55], called Reduced Redundancy Storage (RRS). That storage class provided just four 9's of durability compared to S3's usual eleven 9's and was initially priced at 33% cheaper than other storage classes. However, in 2017, this storage class was deprecated with no reduced durability replacement [56]. We have been unable to find out why Amazon RRS was deprecated. One possible explanation is that it was difficult to use: when data was lost, S3 would return a specific HTTP error code (405, "Method Not Allowed"). Developers would need to add special handling to their application to check for this code, and then to respond accordingly when data was lost. Both aspects of this (detecting and responding) presents

a burden that developers may not have been willing to bear. Additionally, the appropriate response to lost data was often specific to each application, making it difficult to generalize and handle by a library or framework.

Nowadays, the rise of serverless has greatly simplified the task of handling lost data. Lost data can now be generically handled by re-running a function to replace the lost data. This enables libraries or frameworks, such as we present in this paper, to take care of responding to lost data.

To the best of our knowledge, no other cloud provider has offered any kind of reduced-durability storage.

Spark's [57] Resilient Distributed Datasets (RDDs) supports storing data in non-durable storage such as memory. In the event of data loss, the RDD recomputes the lost data. This is similar to our approach, except it does not use a model for identifying the most cost effective storage class for the data.

c) Serverless workflow execution: Tools such as AWS Step Functions [58] and OpenWhisk Action Sequences [59] allow users to combine multiple actions in a sequence, passing data from one stage to the next. However, they do not track the provenance of data produced by the actions. This makes it impossible to transparently handle data loss, as our work does. Similarly, there has been a lot of research on serverless execution systems (*e.g.*, Hyperflow [44], FaaSFlow [60], gg [61], Sprocket [62], Wukong [50], and SONIC [24]). These projects focus on various aspects of writing applications that run on serverless platforms, but do not address the problem of re-creating data lost by a non-durable storage system.

Microsoft's Durable Function framework [63], [15], [64] allows users to build complex applications that are executed in a serverless context. Results from individual stages are saved, and if the stage needs to be re-run, the saved results are used instead of re-computing. This is similar in concept to our work, in that they address the possibility of needing to re-run functions using have special support. However, they do not address the possibility of data loss. Therefore, our work is complimentary in that our work could be used to guide the placement of the intermediate data saved by the Durable Functions framework.

VII. Conclusion

Storage systems have been built and operated with the assumption that high durability is necessary and desired for all kinds of data. In this paper, we revisit this assumption and find that it no longer holds for all data. Specifically, serverless data has unique traits that make it tolerant of loss: it can be re-created and it is short lived. In other words, it is unlikely to be lost even if stored on low-durability storage. Moreover, in the event that it is lost, there is a clear recovery mechanism.

We presented a mathematical model, SDCM, that identifies the most cost effective storage class for serverless data, given application and environmental parameters. We also built an execution system using SDCM to place data, which transparently re-runs functions in response to lost data. Finally, we demonstrate how the placement decisions made by SDCM can lower storage costs when executing serverless DAGs, by up to 3×, while never exceeding baseline costs.

VIII. Acknowledgments

We thank the anonymous reviewers and our shepherd, Raju Rangaswami, for their constructive feedback. This work was made possible in part thanks to Dell-EMC, NetApp, Facebook, and IBM support; a SUNY/IBM Alliance award; and NSF awards CNS-1900706, CCF-1918225, CNS-1951880, CNS-2106263, CNS-2106434, and CNS-2214980. Xinran Wang is supported by the 3M Science and Technology Graduate Fellowship.

References

[1] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers," in *Proceedings of the Scientific Discovery through Advanced Computing*, ser. SciDAC'07, 2007.

[2] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*. Santa Clara, CA: USENIX Association, February 2016, pp. 67–80.

[3] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you?" in *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*. San Jose, CA: USENIX Association, February 2007, pp. 1–16.

[4] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. Khessib, and K. Vaid, "SSD failures in datacenters: What? when? and why?" in *Proceedings of the Ninth ACM Israeli Experimental Systems Conference (SYSTOR '16)*. Haifa, Israel: ACM, May 2016, pp. 7:1–7:11.

[5] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," in *Proceedings of the 2015 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2015)*. Portland, OR: ACM, June 2015, pp. 177–190.

[6] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu, "Lessons and actions: What we learned from 10k SSD-Related storage system failures," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 961–976. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/xu

[7] P. Villalobos and A. Ho, "mitmproxy," September 2022, https://epochai.org/blog/trends-in-training-dataset-sizes.

[8] D. Fediuk, "Increasing dataset sizes," May 2019, https://dmitry.ai/t/topic/198.

[9] B. Wilson, "Backblaze durability calculates at 99.999999999% — and why it doesn't matter," July 2018, https://www.backblaze.com/blog/cloud-storage-durability/.

[10] C. Quinn, "S3's durability guarantees aren't what you think," April 2021, https://www.lastweekinaws.com/blog/s3s-durability-guarantees-arent-what-you-think/.

[11] "Amazon s3 faqs," 2024, https://aws.amazon.com/s3/faqs/.

[12] "Azure storage redundancy," Jan 2024, https://learn.microsoft.com/en-us/azure/storage/common/storage-redundancy.

[13] "Developing for retries and failures," 2024, https://docs.aws.amazon.com/lambda/latest/operatorguide/retries-failures.html.

[14] "Designing azure functions for identical input," June 2022, https://learn.microsoft.com/en-us/azure/azure-functions/functions-idempotent.

[15] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, "Durable functions: Semantics for stateful serverless," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485510

[16] H. Ding, Z. Wang, Z. Shen, R. Chen, and H. Chen, "Automated verification of idempotence for stateful serverless applications," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 887–910. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/ding

[17] S. Ford and M. Skoviera, "Avoiding gcf anti-patterns part 1: How to write event-driven cloud functions properly by coding with idempotency in mind," October 2021, https://cloud.google.com/blog/topics/developers-practitioners/avoiding-gcf-anti-patterns-part-1-how-write-event-driven-cloud-functions-properly-coding-idempotency-mind.

[18] J. Beswick, J. Van Der Linden, and D. Osiennik, "Handling lambda functions idempotency with aws lambda powertools," April 2022, https://aws.amazon.com/blogs/compute/handling-lambda-functions-idempotency-with-aws-lambda-powertools/.

[19] M. Wu, "Maybe you shouldn't be that concerned about data durability," June 2019, https://blog.synology.com/data-durability.

[20] "The data explosion and hidden data storage costs in the cloud – could object storage be the answer?" September 2023, https://www.lightedge.com/blog/the-data-explosion-and-hidden-data-storage-costs-in-the-cloud-could-object-storage-be-the-answer/.

[21] T. Savvas, "Increased cloud costs pulls focus of some smes to on-site storage," September 2023, https://blocksandfiles.com/2023/09/11/increased-cloud-costs-pulls-focus-of-some-smes-to-on-site-storage/.

[22] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[23] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding ephemeral storage for serverless analytics," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 789–794. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[24] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 285–301. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/mahgoub

[25] A. Merenstein, V. Tarasov, A. Anwar, S. Guthridge, and E. Zadok, "F3: Serving files efficiently in serverless computing," in *Proceedings of the 16th ACM International Systems and Storage Conference (SYSTOR '23)*. Haifa, Israel: ACM, Jun. 2023, won Best Paper Award.

[26] "Data preprocessing for ml: options and recommendations," August 2023, https://www.tensorflow.org/tfx/guide/tft_bestpractices.

[27] S. Ghosh, "A comprehensive guide to data preprocessing," August 2023, https://neptune.ai/blog/data-preprocessing-guide.

[28] A. Raina, "Redis use case examples for developers," July 2022, https://redis.io/blog/5-industry-use-cases-for-redis-developers/.

[29] J. Beswick, "Replacing web server functionality with serverless services," July 2020, https://aws.amazon.com/blogs/compute/replacing-web-server-functionality-with-serverless-services/.

[30] "Serverless web application," https://learn.microsoft.com/en-us/azure/architecture/web-apps/serverless/architectures/web-app.

[31] "mitmproxy," https://mitmproxy.org/.

[32] MongoDB, Inc., "MongoDB: The database for modern applications," Sep. 2019, https://www.mongodb.com/.

[33] "Using amazon s3 storage classes," https://docs.aws.amazon.com/AmazonS3/latest/userguide/storage-class-intro.html.

[34] "Using storage classes," https://docs.ceph.com/en/latest/radosgw/placement/#using-storage-classes.

[35] I. Iliadis and V. Venkatesan, "Rebuttal to "beyond mttdl: A closed-form raid-6 reliability equation"," *ACM Trans. Storage*, vol. 11, no. 2, mar 2015.

[36] J. L. Hafner and K. Rao, "Notes on reliability models for non-mds erasure codes," 2006.

[37] K. M. Greenan, J. S. Plank, and J. J. Wylie, "Mean time to meaningless: MTTDL, Markov models, and storage system reliability," in *HotStorage '10: Proceedings of the 2nd USENIX Workshop on Hot Topics in Storage*, 2010.

[38] J. G. Elerath and J. Schindler, "Beyond mttdl: A closed-form raid 6 reliability equation," *ACM Trans. Storage*, vol. 10, no. 2, mar 2014. [Online]. Available: https://doi.org/10.1145/2577386

[39] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, "ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 303–320. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/mahgoub

[40] A. Smith, "Cloud storage economics: List price stagnation and fee inflation challenge traditional expectations," September 2022, https://wasabi.com/industry/cloud-storage-fee-inflation/.

[41] L. Kuperman, "Why your cloud expenses are rising: Blame cloud-flation," July 2022, https://tdwi.org/articles/2022/07/13/ppm-all-why-cloud-expenses-are-rising-cloud-flation.aspx.

[42] "Ffmpeg," https://ffmpeg.org/.

[43] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. C. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. A. Prince, and R. Williams, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *Int. J. Comput. Sci. Eng.*, vol. 4, pp. 73–87, July 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1568665.1568666

[44] M. Malawski, "Towards serverless execution of scientific workflows-hyperflow case study." in *Works@ Sc*, 2016, pp. 25–33.

[45] "Ffmpeg hyperflow wms," https://github.com/hyperflow-wms.

[46] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[47] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, fast and slow: Scalable analytics on serverless infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pu

[48] M. Wawrzoniak, I. Müller, G. Alonso, and R. Bruno, "Boxer: Data analytics on network-enabled serverless platforms," in *Conference on Innovative Data Systems Research*, 2021.

[49] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 923–935.

[50] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng, "Wukong: A scalable and locality-enhanced framework for serverless parallel computing," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–15. [Online]. Available: https://doi.org/10.1145/3419111.3421286

[51] V. Sreekanti, C. Wu, X. Charles Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: stateful functions-as-a-service," in *Proceedings of the VLDB Endowment, Volume 13, Issue 12*, 2020, pp. 2438–2452.

[52] Y. Liu, Z. Huang, J. Yue, H. Huang, S. Wu, and J. Hai, "Funcstore: Resource efficient ephemeral storage for serverless data sharing," in *Proceedings of the 38th Symposium on Mass Storage Systems and Technologies (MSST)*, 2024.

[53] D. Lelewer and D. Hirschberg, "Data compression," in *ACM Computing Surveys (CSUR)*. ACM, 1987, pp. 261–296.

[54] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok, "Dmdedup: Device mapper target for data deduplication," in *Proceedings of the Linux Symposium*, Ottawa, Canada, Jul. 2014, pp. 83–95.

[55] J. Barr, "New: Amazon s3 reduced redundancy storage (rrs)," May 2010, https://aws.amazon.com/blogs/aws/new-amazon-s3-reduced-redundancy-storage-rrs/.

[56] C. Quinn, "S3 reduced redundancy storage is dead," April 2017, https://www.lastweekinaws.com/blog/s3-reduced-redundancy-storage-is-dead/.

[57] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica *et al.*, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[58] "Aws step functions," https://aws.amazon.com/step-functions/.

[59] "Creating action sequences," July 2023, https://github.com/apache/openwhisk/blob/master/docs/actions.md#creating-action-sequences.

[60] Z. Li, Y. Liu, L. Guo, Q. Chen, J. Cheng, W. Zheng, and M. Guo, "Faasflow: Enable efficient workflow execution for function-as-a-service," in *ASPLOS '22: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 782–796. [Online]. Available: https://doi-org.proxy.library.stonybrook.edu/10.1145/3503222.3507717

[61] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton,

WA: USENIX Association, Jul. 2019, pp. 475–488. [Online]. Available: http://www.usenix.org/conference/atc19/presentation/fouladi

[62] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *SoCC '18: Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 263–274. [Online]. Available: https://doi.org/10.1145/3267809.3267815

[63] "What are durable functions?" August 2023, https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.

[64] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C. S. Meiklejohn, and X. Zhu, "Netherite: Efficient execution of serverless workflows," *Proc. VLDB Endow.*, vol. 15, no. 8, p. 1591–1604, apr 2022. [Online]. Available: https://doi.org/10.14778/3529337.3529344