Techniques for Storage Performance Measurement and Data Management in Container-Native and Serverless Environments

A Dissertation presented

by

Alex Merenstein

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-24-02

May 2024

Stony Brook University
The Graduate School

Alex Merenstein

We, the thesis committee for the above candidate for the
degree of Doctor of Philosophy, hereby recommend
acceptance of this dissertation

Erez Zadok – Dissertation Advisor
Professor, Computer Science Department

Anshul Gandhi – Chairperson of Dissertation Proposal
Associate Professor, Computer Science Department

Dongyoon Lee
Associate Professor, Computer Science Department

Ali Anwar
Assistant Professor, Computer Science & Engineering Department, University of Minnesota

Vasily Tarasov
Research Scientist, IBM Research

This dissertation is accepted by the Graduate School

Celia Marshik

Dean of the Graduate School

Abstract of the Dissertation

Techniques for Storage Performance Measurement and Data Management in Container-Native and Serverless Environments

by

Alex Merenstein

Doctor of Philosophy

in

Computer Science

Stony Brook University

2024

Serverless platforms have exploded in popularity in recent years. Unlike traditional "server-full" platforms that require users to manage and operate bare-metal servers, virtual machines (VMs) or containers, serverless platforms completely remove the execution environment from the responsibility of the user. The serverless platform manages creating and destroying the environment (usually based on a container or VM), freeing the user to focus on their application code.

In addition to being an easier user experience, serverless can reduce costs: serverless platforms use an on-demand cost model, meaning that users are charged only for time that their code actually runs. This is in contrast to server-full platforms, where a user must provision a VM or container in advance of when it will be needed. The user then pays for the entire time the VM or container runs, including while it is idling (*e.g.*, memory and storage space consumption). Serverless, therefore, offers a cheaper, easier alternative to deploying applications compared to traditional VMs or containers.

The move to serverless brings with it a range of new workload characteristics not seen previously. Compared to monolithic applications, serverless applications tend to be short lived, and consist of many small actions. We begin by exploring the unique challenges to measuring performance of these workloads, with an emphasis on the storage components of the workload. We developed CNSBench, which enables users to assess the performance of their application and storage infrastructure. CNSBench allows users to create storage workloads that are representative of real cloud native and serverless environments: dynamic and consisting of a diverse set of individual workloads. Since serverless platforms are typically built atop container-based orchestration platforms, CNSBench helps users to measure storage performance at the container orchestration level. This is more general and flexible than targeting serverless platforms specifically, while still

allowing for the evaluation of serverless platforms and applications with CNSBench.

Once we can measure the storage performance of serverless applications more accurately, we turn to improving performance. We introduce new techniques for managing data that are tailored to the usage patterns common in serverless environments, reducing costs to users while maintaining required levels of data durability. To this end, we built F3, a file system designed to optimize data exchange in serverless platforms. F3 introduces new methods for handling ephemeral data and modifications to a serverless scheduling algorithm so that data-locality is considered when scheduling serverless actions. These changes help to adapt existing file-based storage options to modern, cloud-native applications and use cases. F3 improves the performance of intermediate data transfer, increasing throughput by up to 6.5× and decreasing latency by as much as 2.6×.

By introducing new methods for handling ephemeral data, F3 makes a tradeoff between durability and performance. It does so by using higher performance but less-durable data stores for ephemeral data passed between application components. We further explore how lower-durability storage can be used, trading off durability for (dollar) cost. We have developed a mathematical model, called *Storage Durability Cost Model* (SDCM), that determines the durability level most appropriate for an application and its data. Additionally, we introduce an application architecture that utilizes this model to place data in cheaper storage while still meeting the data's durability requirements, thereby reducing overall costs to users. We show how SDCM can reduce storage costs by up to 3×.

In sum, serverless environments offer significant benefits over more traditional virtual-machine–based cloud environments. However, the workload characteristics of serverless environments differ in significant ways from other kinds of cloud environments. Therefore, it is our thesis that to fully realize the benefits of serverless computing, new performance measurement techniques and approaches to data management are required.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern clouds have undergone a significant evolution over the past few years. An explosion of cloud deployment options have become available (*e.g.*, edge [68], hybrid [79], multi [79]) and architectures (*e.g.*, microservice [168], function-as-a-service [84] serverless [49]), giving developers a plethora of options for how to deploy their application. Of these deployment options, serverless platforms offer a particularly intriguing set of features, promising to both eliminate infrastructure from the purview of developers and to reduce execution costs.

In serverless platforms, the execution environment is managed by the platform provider. The developer specifies a code snippet they would like to run in response to some trigger (*e.g.*, an HTTP request or an object upload). The platform then is responsible then for (1) creating the execution environment (*e.g.*, virtual machine, container, or both) where the code snippet will run; (2) running the code snippet, passing any arguments related to the trigger (*e.g.*, HTTP headers, object name); (3) monitoring for errors and restarting if needed; (4) capturing output, recording logs; and (5) deleting the execution environment when the snippet finishes. The platform may also provide additional advanced features, such as pre-loading the execution environment in anticipation of needing to respond to a trigger. By managing this complexity on behalf of the developer, the serverless platform frees the developer to focus on their application code.

Serverless platforms are also unique compared to other deployment methods in how users are billed: rather than reserving (and paying for) resources up front, serverless applications are billed only for the time spent running user code. For example, in a server-full environment, a user may deploy a webserver in a virtual machine. They then pay for the entire time that the virtual machine runs, regardless of whether the webserver is actually handling any requests. In a serverless environment, the webserver is started in response to a trigger (*e.g.*, an HTTP request) and stops running when the trigger is handled. The user is billed only for the time spent running, *i.e.*, the time spent actively handling a request. For many workloads, this can significantly reduce costs.

As serverless differs significantly from more traditional deployment platforms such as bare metal machines, virtual machines, and containers, it is not surprising to find that serverless environments exhibit significantly different workload characteristics. These new workload characteristics make it difficult to asses the storage performance of a workload or platform.

The first of these new characteristics is the increased importance of storage control operation such as volume creation and attachment. These operations have become much more frequent in

serverless environments thanks to a higher rate of workload churn and the increased ability for users—not just administrators—to manage storage volumes [113, 104]. Serverless applications also drive an increase in the diversity of workload mixes on each host [190, 168, 84], making it difficult to understand or predict the performance of each individual workload. Finally, serverless applications tend to be highly dynamic: the "on-demand" feature of serverless means that execution environments are frequently created and destroyed to closely match the current level of demand.

In addition to new operating characteristics, serverless applications have new data-access patterns as well. As applications are split into many short-lived services, an increased amount of data must now be passed between these services. This data is often short lived and can be easily re-generated if lost. Existing storage systems are designed to store data durably; they lack the optimizations and data-handling techniques that are appropriate for this kind of ephemeral data.

In this thesis, we address these new properties of serverless platforms and applications with two thrusts: (1) by developing a benchmark suite capable of assessing the storage performance of serverless workloads and (2) developing new data-management techniques tailored for access patterns common in serverless environments

We address the first challenge, of benchmarking in serverless environments, with CNSBench, a benchmark framework designed for serverless applications and environments. CNSBench enables users to create benchmarks that orchestrate the execution of multiple applications. In addition, CNSBench allows users to specify a "control workload," which consists of actions such as scaling an application deployment or snapshotting a storage volume. By orchestrating the combination of multiple applications and control workloads, CNSBench can generate workloads that are representative of real serverless workloads. This enables users to evaluate how their application will perform under serverless conditions (*e.g.*, frequent scaling or the presence of other workloads) on a particular platform. We designed CNSBench to operate at the level of the container orchestrator (*i.e.*, Kubernetes [103]), since serverless platforms are often built on top of container orchestrators. By targeting this level, we can evaluate both serverless applications and serverless platforms themselves. We show how CNSBench can assess the performance of serverless storage systems and applications.

To address the second challenge, the existence of new data access patterns in serverless applications, we developed F3. F3 is a file system that introduces new data-handling techniques tailored for the kind of ephemeral data common to serverless applications. F3 enables serverless applications to use file-based storage, and enables serverless platforms to use data locality information when scheduling the execution of serverless actions. Additionally, F3 makes a tradeoff between performance and durability, using lower-durability storage for ephemeral data that can be re-generated if lost. We show that by using F3 to transfer intermediate data in serverless applications, we can achieve up to 6.5× higher throughput and 2.6× lower latency.

Finally, we continue exploring the durability requirements in storage used for intermediate data transfer in serverless applications. In F3 we use lower-durability storage to accelerate the transfer of intermediate data. We extend this work, now focusing on the cost savings achievable through using lower-durability storage. Serverless environments offer the fairly unique property of being able to re-run a specific function in the application to re-create the outputs of that function. We

use this capability to make a tradeoff between the cost to store data in highly durable, replicated storage, and the cost to re-create that data if lost due to a storage failure. We developed a mathematical model, called *Storage Durability Cost Model* (SDCM), to help make this tradeoff. We then use SDCM to show that in many cases, it is more cost effective to use low-durability storage for intermediate serverless data. We then developed an execution system capable of automatically re-running functions to re-create lost data, thereby hiding the complexity of handling lost data from the developer. Our execution system also handles transparently placing data at the correct durability level based on the decision made by SDCM. We show with real world applications how SDCM and our execution system can be used to reduce storage costs by up to $3\times$.

In sum, it is our thesis that the characteristics of new serverless environments and applications require new tools and techniques. We first developed a benchmarking tool capable of measuring performance in a realistic manner, with all of the storage control operations, application dynamism, and diverse workloads found in these new serverless environments. We then developed new techniques for optimizing the transfer of ephemeral intermediate data, which is frequently generated by serverless applications. Finally, we developed a durability cost model to place this ephemeral intermediate data in storage with the cost-optimal level of durability.

The rest of this thesis proposal is organized as follows: In Chapter 2 we describe our motivation. In Chapter 3 we describe related works. In Chapter 4 we describe our benchmark framework CNSBench and provide an evaluation of its use. In Chapter 5 we describe F3, a storage system optimized for serverless applications. In Chapter 6 we describe SDCM and execution system for placing serverless data at different levels of storage durability. In Chapter 7 we finish with our conclusions and thoughts on possible future research directions.

Chapter 2

Motivation

In this chapter, we describe our motivation behind creating a new benchmark framework, a serverless-oriented file system, and a durability cost model for serverless storage.

2.1 Benchmarking for Serverless

Serverless platforms and applications have significantly different properties compared to previous cloud platforms and applications. For example, storage control operations are more common and applications are much more dynamic.

In our evaluation of CNSBench, we found that storage system performance varies widely across different storage systems, with some storage providers exhibiting especially low performance when executing control operations like snapshotting a volume. We also found that workloads consisting of different mixes of applications exhibited different performance, depending on the specific mix of applications. Both of these findings show the importance of being able to benchmark storage systems and applications under conditions characteristic of serverless platforms and applications. Existing benchmarks do not generate storage control operations, making it difficult to assess the performance of storage systems under realistic serverless conditions. Existing benchmarks also do not make it easy to orchestrate different mixes of applications, making it difficult to understand the performance implications when many different serverless functions are run on the same host or cluster.

2.2 File System for Serverless

Serverless and *Function as a Service* (FaaS) platforms have become popular for building and deploying applications. These platforms encourage breaking larger applications into many individual services, which makes them especially useful as a platform for deploying and running microservices. Communication between services is difficult though: many cloud platforms do not allow direct service to service network connections, and many do not offer file-based storage. The result is that applications often resort to using object stores as a means for transferring data between

components: one service writes data to an object store, and then a second service reads that data from the object store.

Using an object store for communication is often not ideal. Existing object stores often forbid or have poor performance when there are simultaneous readers and writers accessing the same data. In serverless applications, this is a common pattern. Specifically, it is common for one service to consume data as it is being written by another service.

Additionally, existing applications might require access to file-based storage. Relying solely on object storage for data transfer precludes these applications from being ported to a serverless platform.

With F3, we were able to run unmodified applications, despite these applications requiring file-based storage interfaces. This, as well as the performance improvements we demonstrate with F3, support our decision to develop a file system with data management techniques tailored for serverless applications.

2.3 Durability Requirements for Serverless Data

Storage devices are in general fairly reliable, and most users will go years or longer without ever experiencing data loss due to a failed device. At scale, however, the story is different: data centers and clouds with millions of devices must contend with frequent failures. To handle this, cloud storage systems rely on techniques such as replication and erasure coding. These are methods for increasing data's resiliency to device failures, and work by essentially spreading copies of the data across many devices. If a data copy resides on a device that fails, the original data can still be recreated with the other copies. The data's ability to withstand device failure is referred to as its *durability*. By varying the number of devices that data is copied across, a user can achieve different levels of durability.

In general, all cloud storage is highly durable. This is necessary for most data that cannot be lost, but does come with a cost: additional storage space is used for the multiple copies of data, as well as the additional storage and network bandwidth needed to distribute the data copies. If data could be re-created, perhaps this additional overhead might not be needed.

Indeed, serverless enables data to be re-created. Each action within a serverless function is supposed to be able to be re-run and have the same effect as the first time it was run. If an action produced some piece of data, re-running that action will produce the same data again. This provides an alternative way to deal with storage device failures: rather than making multiple copies of data, we can simply re-run the action that created it originally. With this approach, we can reduce overheads (and cost) by storing data in lower-durability, non-replicated storage.

Alas, this approach still comes with a cost. Re-running an action uses compute resources, which incurs a cost based on the time needed to re-run the action. We therefore have a tradeoff to make: we can save money on storage by using lower-durability storage, but might pay more for compute to re-create lost data; or, we can spend more on storage, not lose any data, and not pay any extra compute to re-create data. This tradeoff must be made for each piece of intermediate data created by a serverless application, and will depend on factors such as the cost of compute and the size of the data.

Finding the cost-optimal point in this tradeoff is difficult and not practical to compute by hand. Therefore, we have developed a mathematical model, called *Storage Durability Cost Model* (SDCM). SDCM considers application and environmental parameters and chooses the appropriate level of durability for each piece of data. Since requiring developers to account for the possibility of lost data in their application would be a significant burden, we have also developed an execution framework that transparently re-creates data as needed.

Chapter 3

Related Work

In this chapter, we survey related works about storage benchmarking, storage for serverless, and storage durability. In Section 3.1 we discuss works covering classic storage benchmarks, benchmarks designed specifically for object stores, and cloud-native benchmarks; in Section 3.2 works related to using storage systems to facilitate the passing of intermediate data in serverless applications, data location-aware scheduling, and special handling for read-while-write workloads; and finally, in Section 3.3 we cover techniques for reducing storage costs in serverless, lower-durability storage options, and serverless workflow execution systems.

3.1 Storage Benchmarking

Classic storage benchmarks Storage benchmarking is an old and complex topic with many applicable techniques and intricate nuances [184]. Therefore, it is not surprising that the array of tools for benchmarking and corresponding studies is extensive. Filebench [186], fio [63], SPEC SFS [178], and IOZone [33] are just a few examples of popular file system benchmarks. For a comprehensive survey of file system and storage benchmarks we refer the reader to a study by Traeger *et al.* [194].

The majority of such benchmarks generate a single, stationary workload per run, which is not representative of cloud native environments. Few benchmarks have built-in mechanisms to dynamically increase the load, in order to discover the peak throughput where diminishing returns (*e.g.*, due to thrashing) begin to take over. For example, measuring NFS throughput via SPEC SFS [179] and process scheduling throughput using AIM7 [187].

Filebench [186, 4] comes with several canned configurations [152] and even has its own Workload Modeling Language (WML) [206]. It, however, is not distributed (cannot run in a coordinated manner across multiple containers) and, though WML is flexible for encoding stationary workloads, is still limited in creating dynamically changing workloads. In our experience, adding support for distributed and temporally varying workloads to Filebench's WML is a difficult task. Therefore, in CNSBench, we exploited the orchestration capabilities of cloud native environments and delegated these tasks to a higher level (*i.e.*, the CNSBench controller and the Kubernetes orchestrator itself). This further allowed us to support any existing benchmarks as canned I/O

generators.

RocksDB [59] is a popular key-value store with canned, preconfigured workloads using a db_bench driver to create random/sequential reads/writes and mixes thereof. One can run these workloads in any order and configure their working-set size. However, that is still a manual process with little flexibility, and no support for control operations (which is true for the previously mentioned benchmarks as well).

Object storage benchmarks In recent years the need to test the performance of cloud storage has motivated academia and industry to develop several micro-benchmarks for that task such as YCSB [44] and COSBench [211]. YCSB is an extensible workload generator that evaluates the performance of different cloud-serving key-value stores. COSBench measures the performance of cloud object storage services and comes with plugins for different cloud providers. Unlike these benchmarks, CNSBench focuses on workloads that run in containers and require a file system interface.

Cloud native benchmarks TailBench [94] provides a set of interactive macro-benchmarks: web servers, databases for speech recognition, and machine translation systems to be executed in the cloud. Similarly, DeathStarBench [67] is a benchmark suite for microservices and their hardware-software implications for cloud and edge systems. Both TailBench and DeathStarBench target cloud applications and are not explicitly storage benchmarks.

3.2 Storage for Serverless

Jonas *et al.* [91] implemented PyWren, which enables the massive parallelization of Python applications using AWS Lambda. This is one of the first cases of researchers using serverless platforms for use cases beyond web applications, and they found that existing storage solutions were lacking. In particular, they reported that the existing storage solutions are incapable of supporting large scale data operations. Following PyWren, Klimovic *et al.* [97] examined the storage use of several FaaS applications and proposed the design of a storage system suitable for these new use cases. Unlike F3, these works do not consider file-based storage for serverless.

Several papers introduce new storage systems for serverless platforms: Locus [157], Pocket [98], and Cloudburst [180]. Other frameworks for writing or running applications on serverless platforms handle storage by abstracting access to one of many possible storage backends. Examples include gg [65] and Ray [142]. In all of these cases, access to storage was exposed via a custom API interface which would require porting existing applications in order to run. Conversely, F3 allows existing applications to run unmodified. Also, F3 could be integrated into frameworks like gg or Ray as an alternative storage backend, or could be layered on top of one of the existing storage backends supported by those frameworks.

Schleier-Smith *et al.* [172] make a similar argument as we do in favor of a file interface for serverless applications. However, they assert that existing shared file systems are too slow and are incompatible with cloud environments where failures and high latencies are common; and they

propose a transactional interface. We believe that small edge data centers will have fewer random failures and lower latency than cloud data centers, and that shared file systems can achieve high performance in this setting (see Section 5.6).

Wukong [35] and SONIC [125] aim to accelerate data transfer in serverless environments by scheduling connected actions together on the same node. However, they require prior knowledge of the workload, such as the graph of which actions call other actions in order to schedule actions that share data together on the same node. F3 does not require prior knowledge about workloads in order to schedule actions close to their data.

Other work has explored transferring data using direct network connections between two serverless actions, made possible via NAT hole-punching [200]. This addresses the issue of data transfer between actions but does not address the need for file-based storage.

Our location-aware data scheduling is similar to the ideas implemented by Hadoop [6] and HDFS [7]. Hadoop and HDFS are designed for map-reduce environments and fit well for data analytics tasks. It is not possible to access HDFS data through the usual `read` and `write` system calls. F3 is created specifically for serverless computing and is suitable for running generic, unmodified applications.

Apache Crail [182] makes a similar argument to us, that some intermediate data generated by applications do not need the durability provided by most storage systems. They introduce an architecture and implementation of a system that provides fast data transfer for ephemeral data. However, unlike F3, Crail exposes a custom API that requires applications to be modified to use.

In HPC environments, burst buffers such as BurstFS [199], and GekkoFS [197] accelerate access to temporary data by adding a faster, less durable storage layer between the application and the cluster's persistent data store. Unlike F3, burst buffers treat all data as ephemeral and do not provide a shared namespace with both ephemeral and non-ephemeral data.

Using non-persistent storage such as RAM for ephemeral data is common (*e.g.*, using Redis [163] or Memcached [131]). These solutions also have no shared namespace with both ephemeral and non-ephemeral data. Additionally, popular memory-based storage systems that are accessible from multiple servers all use object interfaces, rather than file interfaces.

Like F3, the Google File System (GFS) [70] has special support for the read-while-write use case. However, GFS implements a limited number of file operations, making it potentially unsuitable for running unmodified applications. Also, the special support for reading-while-writing is exposed via a new, non-standard operation called *record append*. Unmodified applications therefore cannot benefit from this new feature. In F3, even unmodified applications can benefit from our read-while-write optimizations.

3.3 Durability Requirements for Serverless Data

Storage and data exchange for serverless There has been a large amount of recent work on data exchange for serverless [98, 157, 125, 200, 1, 35, 134, 180, 123]. Most of this work focuses on improving the performance of data transfer in a serverless environment. To do so, Pocket [98] and Locus [157] 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 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 [123] 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 predictions were needed, we could also adopt the lifecycle-prediction approach used by FuncStore.

Projects such as SONIC [125], SAND [1], Wukong [35], and Cloudburst [180] 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 [118] and de-duplication [185] 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 [98], Wukong [35], and FuncStore [123], 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.

Lower-durability storage Amazon previously offered a reduced-durability storage class for its S3 object storage service [22], 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 [159]. 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 [210] 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.

Serverless workflow execution Tools such as AWS Step Functions [17] and OpenWhisk Action Sequences [150] 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 [127], FaaSFlow [120], gg [65], Sprocket [5], Wukong [35], and SONIC [125]). 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 [137, 31, 30] 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.

Chapter 4

CNSBench: A Cloud Native Storage Benchmark

Modern hybrid cloud infrastructures require software to be easily portable between heterogeneous clusters. Application containerization is a proven technology to provide this portability for the *functionalities* of an application. However, to ensure *performance* portability, dependable verification of a cluster's performance under realistic workloads is required. Such verification is usually achieved through benchmarking the target environment and its storage in particular, as I/O is often the slowest component in an application. Alas, existing storage benchmarks are not suitable to generate cloud native workloads as they do not generate any storage control operations (*e.g.*, volume or snapshot creation), cannot easily orchestrate a high number of simultaneously running distinct workloads, and are limited in their ability to dynamically change workload characteristics during a run.

In this chapter, we present the design and prototype for the first-ever Cloud Native Storage Benchmark—CNSBench. CNSBench treats control operations as first-class citizens and allows to easily combine traditional storage benchmark workloads with user-defined control operation workloads. As CNSBench is a cloud native application itself, it natively supports orchestration of different control and I/O workload combinations at scale. We built a prototype of CNSBench for Kubernetes, leveraging several existing containerized storage benchmarks for data and metadata I/O generation. We demonstrate CNSBench's usefulness with case studies of Ceph and OpenEBS, two popular storage providers for Kubernetes, uncovering and analyzing previously unknown performance characteristics.

4.1 Introduction

The past two decades have witnessed an unprecedented growth of cloud computing [130]. By 2020, many businesses have opted to run a significant portion of their workloads in public clouds [23] while the number of cloud providers has multiplied, creating a broad and diverse marketplace [72, 81, 8, 136]. At the same time, it became evident that, in the foreseeable future, large enterprises will continue (i) running certain workloads on-premises (*e.g.*, due to security concerns), and (ii) em-

ploying multiple cloud vendors (*e.g.*, to increase cost-effectiveness or to avoid vendor lock-in). These *hybrid multicloud* deployments [79] offer the much needed flexibility to large organizations.

One of the main challenges in operating in a hybrid multicloud is workload portability—allowing applications to easily move between public and private clouds, and on-premises data centers [101]. Software containerization [83] and the larger cloud native [42] ecosystem is considered to be the enabler for providing seamless application portability [25]. For example, a container image [54] includes all user-space dependencies of an application, allowing it to be deployed on any container-enabled host while container orchestration frameworks such as Kubernetes [103] provide the necessary capabilities to manage applications across different cloud environments. Kubernetes's declarative nature [109] lets users abstract application and service requirements from the underlying site-specific resources. This allows users to move applications across different Kubernetes deployments—and therefore across clouds—without having to consider the underlying infrastructure.

An essential step for reliably moving an application from one location to another is validating its performance on the destination infrastructure. One way to perform such validation is to replicate the application on the target site and run an application-level benchmark. Though reliable, such an approach requires a custom benchmark for every application. To avoid this extra effort, organizations typically resort to using component-specific benchmarks. For instance, for storage, an administrator might run a precursory I/O benchmark on the projected storage volumes.

A fundamental requirement for such a benchmark is the ability to generate realistic workloads, so that the experimental results reflect an application's actual post-move performance. However, existing storage benchmarks are inadequate to generate workloads characteristic of modern *cloud native* environments due to three main shortcomings.

First, cloud native storage workloads include a *high number of control operations*, such as volume creation, snapshotting, etc. These operations have become much more frequent in cloud native environments as users, not admins [113, 104], directly control storage for their applications. As large clusters have many users and frequent deployment cycles, the number of control operations is high [133, 95, 117].

Second, a typical containerized cluster hosts a *high number of diverse, simultaneously running workloads*. Although this workload property, to some extent, was present before in VM-based environments, containerization drives it to new levels. This is partly due to higher container density per node, fueled by the cost effectiveness of co-locating multiple tenants in a shared infrastructure and the growing popularity of microservice architectures [190, 168]. To mimic such workloads, one needs to concurrently run a large number of distinct storage benchmarks across containers and coordinate their progress, which currently involves a manual and laborious process that becomes impractical in large-scale cloud native environments.

Third, applications in cloud native environments are *highly dynamic*. They frequently start, stop, scale, failover, update, rollback, and more. This leads to various changes in workload behavior over short time periods as the permutation of workloads running on each host change. Although existing benchmarks allow one to configure separate runs of a benchmark to generate different phases of workloads [52, 59], such benchmarks do not provide a versatile way to express dynamicity within a *single* run.

In this chapter we present *CNSBench*—the first open-source Cloud Native Storage Benchmark capable of (i) generating realistic control operations; (ii) orchestrating a large variety of storage workloads; and (iii) dynamically morphing the workloads as part of a benchmark run.

CNSBench incorporates a library of existing data and metadata benchmarks (*e.g.*, fio [63], Filebench [186], YCSB [44]) and allows users to extend the library with new containerized I/O generators. To create realistic control operation patterns, a user can configure CNSBench to generate different control operations following variable (over time) operation rates. CNSBench can therefore be seen as both (i) a framework used for coordinating the execution of large number of containerized I/O benchmarks and (ii) a benchmark that generates control operations. Crucially, CNSBench bridges these two roles by generating the control operations to act on the storage used by the applications, thereby enabling the realistic benchmarking of cloud native storage.

As an example, consider an administrator evaluating storage provider performance under a load that includes frequent snapshotting. Conducting an evaluation manually requires the administrator to create multiple storage volumes, run a complex workload that will use that volumes (*e.g.*, a MongoDB database with queries generated by YCSB), and then take snapshots of the volumes while the workload runs. The same evaluation with CNSBench requires just that the administrator specify which workload to run, which storage provider to use, and the rate with which snapshots should be taken. CNSBench handles instantiating each component of the workload (*i.e.*, the storage volume, the MongoDB database, and the YCSB client) and then executing the control operations to snapshot the volume as the workload runs.

While developing CNSBench, we have also been building out a library of pre-defined workloads. The previous example uses one such workload, which consists of YCSB running against a MongoDB instance. If the administrator instead wanted to instantiate a workload not found in our library, it is easy to package an existing application into a workload that can be used by CNSBench. In that case, we would also encourage the administrator to contribute their new workload back to our library so that it could be used by a broader community.

To demonstrate CNSBench's versatility, we conducted a study comparing cloud native storage providers. We pose three questions in our evaluation: (A) How fast are different cloud storage solutions under common control operations? (B) How do control operations impact the performance of user applications? (C) How do different workloads perform when run alongside other workloads? We use Ceph [36] and OpenEBS [146] in our case study as sample storage providers. Our results show that control operations can vary significantly between storage providers (*e.g.*, up to 8.5× higher Pod creation rates) and that they can slow down I/O workloads by up to 38%.

In summary, this chapter makes the following contributions:

1. We identify the need and unique requirements for cloud native storage benchmarking.
2. We present the design and implementation of CNSBench, a benchmark that fulfills the above requirements and allows users to conveniently benchmark cloud native storage solutions with realistic workloads at scale.
3. We use CNSBench to study the performance of two storage solutions for Kubernetes (Ceph and OpenEBS) under previously not studied workloads.

CNSBench is open-source and available for download from *https://github.com/CNSBench*.

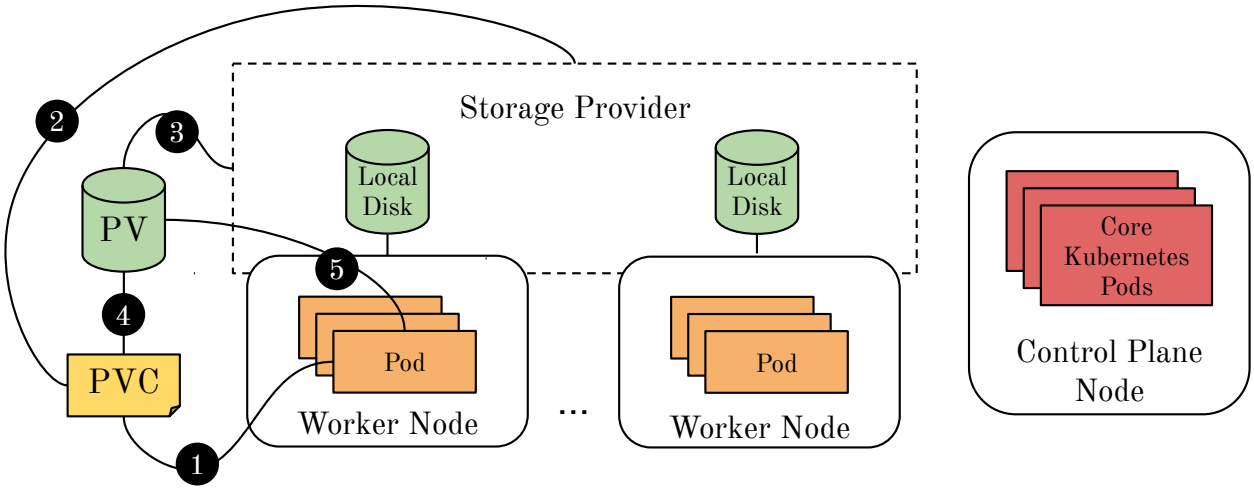


Figure 4.1: Basic topology of a Kubernetes cluster, with a single control plane node, multiple worker nodes, and a storage provider which aggregates local storage attached to each worker node. Also shows the operations and resources involved in providing a Pod with storage.

4.2 Kubernetes Background

We implemented our benchmark for Kubernetes and so in the following sections we use many Kubernetes concepts to contextualize CNSBench's design and use cases. Therefore, we begin with a brief background on how Kubernetes operates.

Overview A basic Kubernetes cluster is shown in Figure 4.1. It consists of control plane nodes, worker nodes, and a storage provider (among other components). Worker and control plane nodes run *Pods*, the smallest unit of workload in Kubernetes that consist of one or more *containers*. User workloads run on the worker nodes, whereas core Kubernetes components run on the control plane nodes. Core components include (1) the API server, which manages the state of the Kubernetes cluster and exposes HTTP endpoints for accessing the state, and (2) the scheduler, which assigns Pods to nodes. Typically, a Kubernetes cluster has multiple worker nodes and may also have multiple control plane nodes for high availability.

The storage provider is responsible for provisioning persistent storage in the form of *volumes* as required by individual Pods. There are many architectures, but the "hyperconverged" model is common in cloud environments. In this model, the storage provider aggregates the storage attached to each worker node into a single storage pool.

The state of a Kubernetes cluster, such as what workloads are running on what hosts, is tracked using different kinds of *resources*. A resource consists of a *desired state* (also referred to as its specification) and a *current state*. It is the job of that resource's *controllers* to reconcile a resource's current and desired states, for example, starting a Pod on node X if its desired state is "running on Node X". Pods and Nodes are examples of resources.

Persistent Storage Persistent storage in Kubernetes is represented by resources called *Persistent Volumes* (PVs). Access to a PV is requested by attaching the Pod to a resource called a *Persistent Volume Claim* (PVC). Figure 4.1 depicts this process: ❶ A Pod that requires storage creates a PVC, specifying how much storage space it requires and which storage provider the PV should be provisioned from. ❷ If there is an existing PV that will satisfy the storage request then it is used. Otherwise, ❸ a new PV is provisioned from the storage provider specified in the PVC. A PVC specifies what storage provider to use by referring to a particular *Storage Class*. This class is a Kubernetes resource that combines a storage provider with a set of configuration options. Examples of common configuration options are what file system to format the PV with and whether the PV should be replicated across different nodes.

Once the PV has been provisioned, ❹ it is bound to the PVC, and ❺ the volume is mounted into the Pod's file system.

Kubernetes typically communicates with the storage provider using the *Container Storage Interface* (CSI) specification [209], which defines a standard set of functions for actions such as provisioning a volume and attaching a volume to a Pod. Before CSI, Kubernetes had to be modified to add support for individual storage providers. By standardizing this interface, a new storage provider needs only to write a CSI driver according to a well-defined API, to be used in any container orchetrator supporting CSI (*e.g.*, Kubernetes, Mesos, Cloud Foundry).

Although Kubernetes has good support for provisioning and attaching file and block storage to pods via PVs and PVCs, no such support exists for object storage. Therefore, CNSBench currently supports benchmarking only file and block storage.

4.3 Need for Cloud Native Storage Benchmarking

In this section we begin with describing the properties of cloud native workloads, which current storage benchmarks cannot recreate. We then present the design requirements for a cloud native storage benchmark.

4.3.1 New Workload Properties

The rise of containerized cloud native applications has created a shift in workload patterns, which makes today's environments different from previous generations. This is particularly true for storage workloads due to three main reasons: (i) the increased frequency of control operations; (ii) the high diversity of individual workloads; and (iii) the dynamicity of these workloads.

Control Operations Previously infrequent, control operations became significantly more common in self-service cloud native environments. As an example, consider the frequent creations and deletions of containers in a cloud native environment. In many cases, these containers require persistent storage in the form of a storage volume and hence, several control operations need to be executed: the volume needs to be created, prepared for use (*e.g.*, formatted with a file system), attached to the host where the container will run (*e.g.*, via iSCSI), and finally mounted in the container's file system namespace. Even if a container only needs to access a volume that already

exists, there are still at least two operations that must be executed to attach the volume to the node where the container will run and mount the volume into the container.

To get a better idea of how many control operations can be executed in a cloud native environment, consider these statistics from one container cluster vendor: in 2019 they observed that over half of the containers running on their platform had a lifetime of no more than five minutes [34]. In addition, they found that each of their hosts were running a median of 30 containers. Given these numbers, a modestly sized cluster of 20 nodes would have a new container being created every second on average. We are not aware of any public datasets that provide insight into what ratio of these containers require storage volumes. However, anecdotal evidence and recent development efforts [105] indicate that many containers do in fact attach to storage volumes.

In addition to being abundant, control operations, depending on the underlying storage technology, can also be data intensive. This makes them slow and increases their impact on the I/O path of running applications. For example, volume creation often requires (i) time-consuming file system formatting; (ii) snapshot creation or deletion, which, depending on storage design, may consume a significant amount of I/O traffic; (iii) volume resizing, which may require data migration and updates to many metadata structures; and (iv) volume reattachment, which causes cache flushes and warmups.

Now that data-intensive control operations are more common, there is a new importance to understanding their performance characteristics. In particular, there are two categories of performance characteristic that are important to understand: (1) How long does it take a storage provider to execute a particular control operation? This is important because in many cases, control operations sit on the critical path of the container startup. (2) What impact does the execution have on I/O workloads? This impact can be significant either due to the increased load on the storage provider or the particular design of the storage provider. For example, some storage providers freeze I/O operations during a volume snapshot, which can lead to a spike in latency for I/O operations [154].

Existing storage benchmarks and traces focus solely on data and metadata operations, turning a blind eye to control operations.

Diversity and Specialization The lightweight nature of containers allows many different workloads to share a single server or a cluster [34]. Workload diversity is fueled by a variety of factors. First, projects such as Docker [53] and Kubernetes [103] have made containerization and cloud native computing more accessible to a wide range of users and organizations, which is apparent in the diversity of applications present in public repositories. For example, on Docker Hub [55] there are container images for fields such as bioinformatics, data science, high-performance computing, and machine learning—in addition to the more traditional cloud applications such as web servers and databases. Additionally, the popularity of microservice architectures has caused traditionally monolithic applications to be split up into many small, specialized components [190]. Finally, the increasingly popular serverless architecture [9], where functions run in dynamically created containers, takes workload specialization even further through an even finer-grained split of application components, each with their own workload characteristics.

The result of these factors is that the workloads running in a typical shared cluster (and on each of its individual hosts) have a highly diverse set of characteristics in terms of runtime, I/O

patterns, and resource usage. Understanding system performance in such an environment requires benchmarks that recreate the properties of cloud native workloads. Currently, such benchmarks do not exist. Hence, realistic workload generation is possible only by manual selection, creation, and deployment of several appropriate containers (*e.g.*, running multiple individual storage benchmarks that each mimic the characteristics of a single workload). As more applications of all kinds adopt containerization and are broken into sets of specialized microservices, the number of containers that must be selected to make up a realistic workload continues to increase. Making this selection manually has become infeasible in today's cloud native environments.

Elasticity and Dynamicity Cloud native applications are usually designed to be elastic and agile. They automatically scale to meet user demands, gracefully handle failed components, and are frequently updated. Although some degree of elasticity and dynamicity has always been a trait of cloud applications, the cloud native approach takes it to another level.

In one example, when a company adopted cloud native practices for building and operating their applications, their deployment rate increased from rolling out a new version 2–3 times per week to over 150 times in a single day [201]. Other examples include companies utilizing cloud native architectures to achieve rapid scalability in order to meet spikes in demand, for example in response to breaking news [47] or the opening of markets [28].

Currently, benchmarks lack the capability to easily evaluate application performance under these highly dynamic conditions. In some cases, benchmark users resort to creating these conditions manually to evaluate how applications will respond—for example manually scaling the number of database instances [44]. However, the high degree of dynamicity and diversity found in cloud native environments makes recreating these conditions manually nearly impossible.

4.3.2 Design Requirements

The fundamental functionality gap in current storage benchmarks is their inability to generate control-operation workloads representative of cloud native environments. At the same time, the I/O workload (data and metadata, not control operations) remains an important component of cloud native workloads, and is more diverse and dynamic than before. Therefore, the primary goal for a cloud native storage benchmark is to enable combining control-operation workloads and I/O workloads—to better evaluate application and cluster performance. This goal led us to define the following five core requirements:

1. I/O workloads should be specified and created independently from control workloads, to allow benchmarking (i) an I/O workload's performance under different control workloads and (ii) a control workload's performance with different I/O workloads.
2. It should be possible to orchestrate I/O and control workloads to emulate a dynamic environment that is representative of clouds today. In addition, it should be possible to generate control workloads that serve as microbenchmarks for evaluating the performance of individual control operations.

3. I/O workloads should be generated by running existing tools or applications, either synthetic workload generators like Filebench or real applications such as a web server with a traffic generator.
4. It should be possible for users to quickly configure and run benchmarks, without sacrificing the customizability offered to more advanced users.
5. The benchmark should be able to aggregate unstructured output from diverse benchmarks in a single, convenient location for further analyses.

A benchmark which meets these requirements will allow a user to understand the performance characteristics of their application and their cluster under realistic cloud native conditions.

4.4 CNSBench Design and Implementation

To address the current gap in benchmarking capabilities in cloud native storage, we have implemented the *Cloud Native Storage Benchmark*—CNSBench. Next, we describe CNSBench's design and implementation. We first overview its architecture and then describe the new Kubernetes *Benchmark* custom resource and its corresponding controller in more detail.

Overview In Kubernetes, a user creates Pods (one of Kubernetes' core resources) by specifying the Pod's configuration in a YAML file and passing that file to the `kubectl` command line utility. Similarly, we want CNSBench users to launch new instances by specifying CNSBench's configuration in a YAML file and passing that file to `kubectl`. To achieve that, our CNSBench implementation follows the *operator design pattern*, which is a standard mechanism for introducing new functionality into a Kubernetes cluster [110]. In this pattern, a developer defines an *Operator* that comprises a custom resource and a controller for that resource. For our implementation of CNSBench, we defined a custom *Benchmark* resource and implemented a corresponding *Benchmark Controller*. Together, these two components form the *CNSBench Operator*. The Benchmark resource specifies the I/O and control workloads, which the controller is then responsible for running.

Figure 4.2 shows the Kubernetes cluster depicted in Figure 4.1 with added CNSBench components shown in blue. The overall control flow is as follow: Ⓐ The Benchmark controller watches the API server for the creation of new Benchmark resources. Ⓑ When a new Benchmark resource is created, the controller creates the resources described in the Benchmark's I/O workload: the *I/O Workload Pods* for running the workloads and the Persistent Volume Claims (PVCs) for the Persistent Volumes (PVs) against which the workloads are run. Ⓒ For running the control operation workload, the Benchmark includes a *Rate Generator*, which triggers an *Action Executor* in user-specified intervals to invoke the desired control operations (*actions*).

4.4.1 Benchmark Custom Resource

The Benchmark custom resource lets users specify three main benchmark properties: (1) the control operation workload; (2) the I/O workloads to run; and (3) where the output should be collected.

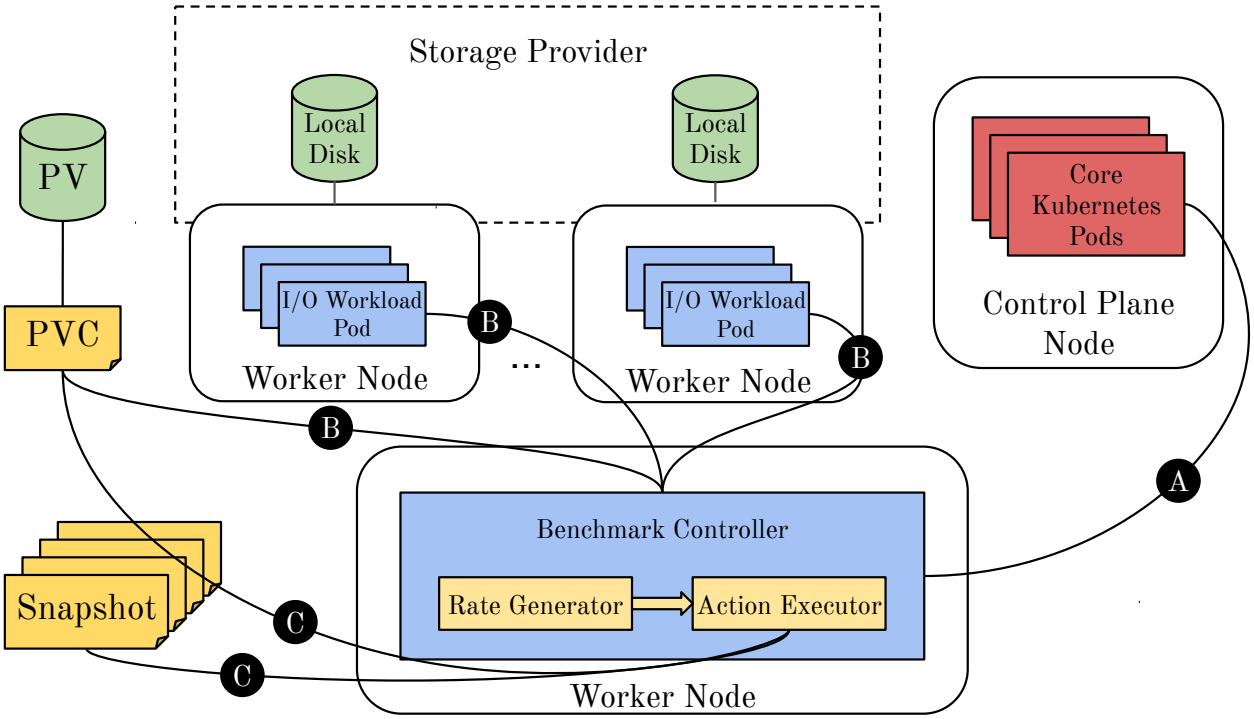


Figure 4.2: CNSBench overview with its components in blue

Control operation workload One of CNSBench's primary requirements is the ability to create realistic control workloads. However, microbenchmarks that purposefully stress only one component or operation of a system are also valuable (*e.g.*, for an in-depth analysis and point optimization of system performance). Useful insights can be derived, for instance, from a benchmark that executes some control operation at a regular interval. Our control workload specification satisfies both use cases, by making it easy to create simple control workloads without sacrificing the ability to define realistic ones.

In CNSBench, control workloads are specified using a combination of *actions* and *rates*. Actions execute operations, for instance *create resource* (*e.g.*, *create Pod or Volume*), *delete resource* (*e.g.*, *delete snapshot*), *snapshot volume*, and *scale resource* (*e.g.*, *scale database deployment*). Rates trigger associated actions at some interval. For our evaluations we used a simple rate which runs actions every T seconds, but more sophisticated rates could be implemented to enable the creation of more realistic control workloads. For example, given a set of cluster traces that logged when different operations were executed, a rate could be implemented that reads those traces and generates a control workload mimicking their specific operating conditions. Actions and rates are deliberately decoupled, so that these more sophisticated rates can be developed independently from CNSBench and then plugged in later.

I/O workload Often, a benchmark's goal is to understand how a particular workload or set of workloads will perform under various conditions. The role of CNSBench's I/O workload component is to either instantiate those workloads or to instantiate a synthetic workload with the same

I/O characteristics of a real workload. Specifying these I/O workloads requires defining all of the different resources (*e.g.*, Pods and PVCs) that must be created in order to run the I/O workload. This can be difficult and make benchmark specifications long and complex.

To ease the burden on users and to help them focus on the overall benchmark specification, rather than the specific details of the I/O workload, CNSBench separates the I/O workload specification from the rest of the benchmark specification. The I/O workload specification is defined using a *ConfigMap*—a core Kubernetes resource for storing configuration files and other free-form text. These files contain the specifications for the Pods that will run the I/O workloads, as well as specifications for supporting resources such as PVCs. In addition, they use metadata annotations to specify information such as what output files should be collected and what parsers should be used to process them. Since the specification uses a core Kubernetes resource, it can be accessed using standard Kubernetes tools from anywhere in the cluster.

Users specify which I/O workloads to run in a Benchmark custom resource using a *create resource* action that references (by name) the I/O workload to create. To enable reuse across various use cases and benchmarks, fields in an I/O workload specification can be parameterized and given a value when the workload is instantiated by a specific benchmark.

We are building out an open source Workload Library, available at *https://github.com/CNSBench/workload-library*, which offers pre-packaged I/O workloads including fio [63], Filebench [186], pgbench [153], YCSB [44], and RocksDB's db_bench [59].Ideally, most users will be able to find a suitable I/O workload in the library and hence, do not need to define their own. We hope that community members will contribute the I/O workloads that they develop to this library as well.

Control and data operations In some cases control and I/O operations can be intertwined. For example, an increase in I/O operations can cause a workload to scale out, which in turn can execute more control operations. Reproducing such events with CNSBench would require a feedback mechanism that conveys to CNSBench information about the I/O operations executed by the I/O workloads. CNSBench's design and implementation do not preclude such mechanism but we leave its implementation to future work.

Benchmark output Many of the results of a CNSBench benchmark will be generated by the I/O workload Pods. Collecting this output presents three challenges. First, Kubernetes currently lacks the ability to extract files from Pods in a clean and generic manner [106]. Second, the output produced by some tools can be large, especially for long-running processes that produce output throughout the run. Third, in our experience, many I/O workloads produce output as unstructured text. This can make it difficult to analyze the results using tools such as Kibana [96], especially if the benchmark consists of multiple I/O workloads that all report results in a different unstructured output formats.

To address these issues, we allow I/O workload authors to specify which files should be collected from the workload Pods and to provide a parser script to process the output. Parsing the output allows large files to be reduced to a more succinct size and to output results in a standard fashion. The output files are collected and parsed using a helper container, described in more detail in Section 4.4.2. Parsers for common I/O benchmarking tools can be included in the Workload Li-

```
1  kind: Benchmark
2  metadata:
3    name: fio-benchmark
4  spec:
5    actions:
6      - name: fio (D)
7        createObjSpec:
8          workload: fio (A)
9          count: 3
10         vars:
11           storageClass: obs-r (C)
12       outputs:
13         outputName: es
14     - name: snapshots
15       rateName: minuteRate
16       snapshotSpec:
17         actionName: fio (D)
18         snapshotClass: obs-csi
19   rates:
20     - name: minuteRate
21       constantRateSpec:
22         interval: 60s
23   outputs:
24     - name: es
25       httpPostSpec:
26         url: http://es:9200/fio/_doc/
```

Listing 4.1: Sample Benchmark Custom Resource Specification

brary, either packaged with the tool's workload specification or as a standalone entry. For instance, we include parsers for fio and YCSB in the Workload Library.

The user specifies where the final, parsed results should be sent to in the *output* section of the Benchmark custom resource. Results do not all need to be sent to the same output. For instance, a benchmark with both fio and YCSB I/O workloads could send the fio results to one location and the YCSB results to another. The benchmark metadata, including the Benchmark resource specification and the start and end times, can be sent to an output as well.Currently CNSBench supports sending the results to a collection server via an HTTP POST request to a user-specified URL. Support for additional kinds of output, such as simply writing the output to a file, can be easily added.

In addition to workload output, it is also important to collect metrics such as Pod or Node resource utilization during a benchmark run. We defer the collection of these metrics to any of the many tools that are commonly used to collect such metrics in a Kubernetes cluster [114].

Example An example Benchmark custom resource is shown in Listing 4.1 and an example of an I/O workload specification is shown in Listing 4.2. Due to space constraints, many of the details of the I/O workload specification are omitted. Figure 4.3 shows the Kubernetes resources that are

```
1  kind: ConfigMap
2  metadata:
3    name: fio (A)
4  spec:
5    data:
6      pod.yaml: | (B)
7        ...
8      pvc.yaml: | (B)
9        ...
10       storageClass: {{storageClass}} (C)
11       ...
```

Listing 4.2: Sample I/O workload specification

created as a result of this Benchmark specification.

Lines 6–13 of Listing 4.1 specify the benchmark's I/O workload. Line 8 references the name of the I/O workload that should be run, labeled (A) in both listings. Lines 6–11 of Listing 4.2 specify the resources that make up the I/O workload. These correspond to the Pods and PVCs in Figure 4.3 labeled (B).

I/O workload specifications can be parameterized to enable their reuse across different use cases and benchmarks. An example of this is on line 10 of Listing 4.2, where the PVC's Storage Class field is parameterized. Label (C) in the two listings and in Figure 4.3 shows how this parameter is set in the Benchmark custom resource specification (line 11 in Listing 4.1), and then how that value is used in the workload's PVCs.

Lines 14-18 of Listing 4.1 specify a *snapshot volume* action. In Kubernetes, volume snapshots are created using a *Snapshot* resource which references a PVC to use as the source of the snapshot. The user indicates which action's PVCs should be snapshotted by referencing the target action by name (line 17 of Listing 4.1). Since all resources created by an action are labeled with that action's name, the controller can map an action name to a set of PVCs (label (D)). These PVCs are then used as the source in the Snapshot resource (label (E)). Additional examples can be found at *https://github.com/CNSBench/CNSBench*.

4.4.2 Benchmark Controller

The Benchmark Controller watches for newly created Benchmark objects and runs their specified actions. The controller has three main responsibilities: (1) triggering control operations; (2) synchronizing the individual benchmark workloads; and (3) collecting the output of the individual workloads.

Triggering control operations When a new Benchmark resource is created, the Controller starts two *goroutines* (Go's equivalent of a thread) for each of the specified rates: one is responsible for generating the rate, and the other is responsible for running all of the actions using that rate. The rate goroutine uses a shared channel to tell the executor goroutine when it is time to run an action.

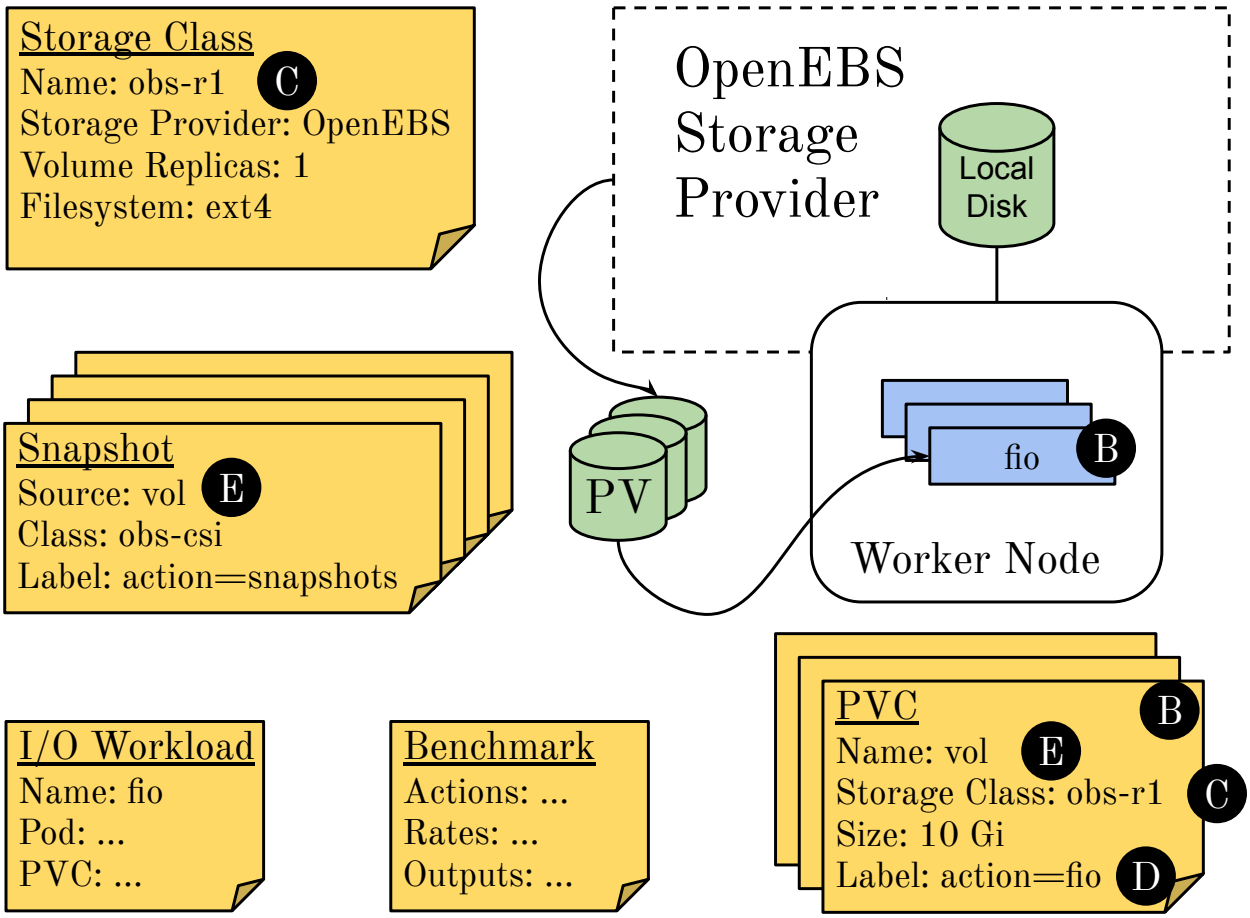


Figure 4.3: Subset of a Kubernetes cluster with a single worker node and a PV. Shows the CNSBench resources that are involved (the I/O Workload and Benchmark), as well as the core Kubernetes resources created by the CNSBench controller according to the Benchmark specification (the Snapshots, PVCs, PV, and workload Pods).

As described in Section 4.4.1, decoupling the rates from the actions simplifies adding new kinds of rates or actions later.

Actions not tied to any rate are run by the controller as soon as the Benchmark resource is created. This is often how I/O workloads are instantiated, since they often use a long running process that generates I/O throughout the benchmark's duration.

Synchronizing workloads In many cases, I/O workloads require an initialization step such as loading data into a database or creating a working set of files. When there are multiple I/O workloads being run, some workloads can finish their initialization step faster than others and begin running their main workload earlier. This can cause misleading and inconsistent results. If the purpose of the benchmark is to evaluate a storage provider's performance under the concurrent load of ten read-heavy I/O workloads, then all ten should start at the same time.

To synchronize the I/O workloads, CNSBench leverages Kubernetes' *initialization containers*

feature. Pods have a list of initialization containers which are executed in order, each one running to completion before the next one starts. The Pod's main containers do not run until all of the initialization containers have completed. CNSBench assumes that a workload's initialization step has been put into an initialization container, which is the responsibility of the I/O workload's author. Although this is usually a straightforward task, it is an example of why separating the I/O workload specifications from the rest of the Benchmark specifications is useful: it allows users to select existing workloads from the Workload Library and not worry about how their workload's initialization is implemented.

When the Benchmark controller instantiates the I/O workloads, it adds an additional *synchronization container* at the end of the list of initialization containers. This container runs a script that queries the Kubernetes API server for the status of each instance of the I/O workload and checks to see if all of their initialization containers have completed (all except for the other synchronization containers). Once all of the non-synchronization initialization containers have completed, the script exits and the synchronization containers stop successfully, allowing Kubernetes to run each Pods' main container. Since all instances of the I/O workload have this synchronization container added, all instances begin running their main containers simultaneously. Many workloads support running for a set amount of time, so synchronizing the finish of each workload is generally not an issue.

Output and metrics collection As described in Section 4.4.1, I/O workload authors can specify which files to extract from a workload's Pods and provide a script to parse those files. Extracting these files from the workload Pods is difficult since there is no standard interface for doing so [106]. The approach used by the official Kubernetes command-line client `kubectl` involves running the `tar` utility inside the target container, and does not work after the container has finished running [107].

To work around these difficulties, the controller modifies the workload Pod to add both a helper container responsible for running the parser script, and also a volume mounted by both the helper and workload containers. The I/O workload author must ensure that the workload's output is written to this volume, which will be mounted at `/output`. Similar to how the synchronization container works, the helper container queries the Kubernetes API server to find out when the workload container has finished; thereafter, the output is ready to be parsed.

4.5 Evaluation

To demonstrate both the need for and the utility of CNSBench, we ran several benchmarks to look at different aspects of cloud native storage performance. We examine the performance of individual control operations, the impact that control operations have on I/O workloads, and the impact that different combinations of I/O workloads can have on overall performance.

4.5.1 Methodology

To evaluate our benchmark, we instantiated an 11-node Kubernetes v1.18.6cluster in an on-premises OpenStack environment: one control plane node and 10 workers. Each worker node is a virtual machine with 4 vCPUs, 8GB of RAM, and 384GB of locally attached storage. The control plane node is a VM with 4 vCPUs, 12GB of RAM, and 100GB of local storage. The VM hosts were located in multiple racks, with racks connected via a 10Gbps network and individual hosts connected to the top of rack switch via 1Gbps links.

We used two storage providers: OpenEBS and Ceph. Our requirements for the storage systems were that they be open-source, free, and not based on cloud-as-a-service model—so we could install and test them locally, and to enable more repeatable results. Additionally, they had to have a CSI driver. These requirements eliminated many existing storage systems. Out of the remaining options, we selected Ceph and OpenEBS due to their popularity.

OpenEBS [146] is a new storage provider built specifically to be cloud native. OpenEBS uses the *Container Attached Storage* paradigm [156], where controllers that provision volumes and manage features such as data replication, themselves run in containers. This provides storage with all of the advantages of the cloud native methodology, such as agility and flexibility. It also enables the storage to be managed like any other resource in a cloud native cluster. We used OpenEBS's cStor storage engine version 2.0.0.

Ceph [202] is a widely used file storage system that is built on top of the RADOS object store [203]. We used the Rook operator for Ceph [167], which handles the deployment and management of a Ceph cluster. The Rook management layer allows Ceph to be managed in a cloud native fashion, using Kubernetes objects and standard Kubernetes management tools. We used Rook version 1.4.1 and Ceph version 15.2.4, with Ceph's BlueStore storage backend.

Both Ceph and OpenEBS provide storage by aggregating the local storage attached to each cluster node. Volumes are provisioned from this combined storage pool and are formatted with Ext4 prior to being attached to a Pod. Ceph and OpenEBS both come with CSI drivers that interface with Kubernetes.

Both OpenEBS and Ceph also offer volume replication for high availability use cases. With volume replication, data written to a volume by a Pod is transparently copied across several volume replicas, which are ideally situated in different availability zones. This enables the cluster to tolerate the loss of one or more hosts—depending on the replication factor—without suffering any data loss. The trade-off is that volume replication often comes at a cost of increased I/O latencies and an increase in network and disk utilization.

Ceph has an additional high availability mechanism using erasure coding, which encodes data into chunks using a forward error-correction code and then replicates those chunks. The use of a forward error-correction code means that fewer replicas are needed to provide the same availability guarantees, and hence less disk space is needed overall. However, erasure coding uses more CPU and RAM than basic data replication.

In our experiments, we use Ceph and OpenEBS in three ways: without replication, in triple-replication mode, and Ceph (only) in erasure-coded mode (ec). In addition to Ceph and OpenEBS, in some evaluations we used a null storage provider that implements the CSI functions involved in provisioning and attaching volumes. The null driver simply returns success to most CSI functions

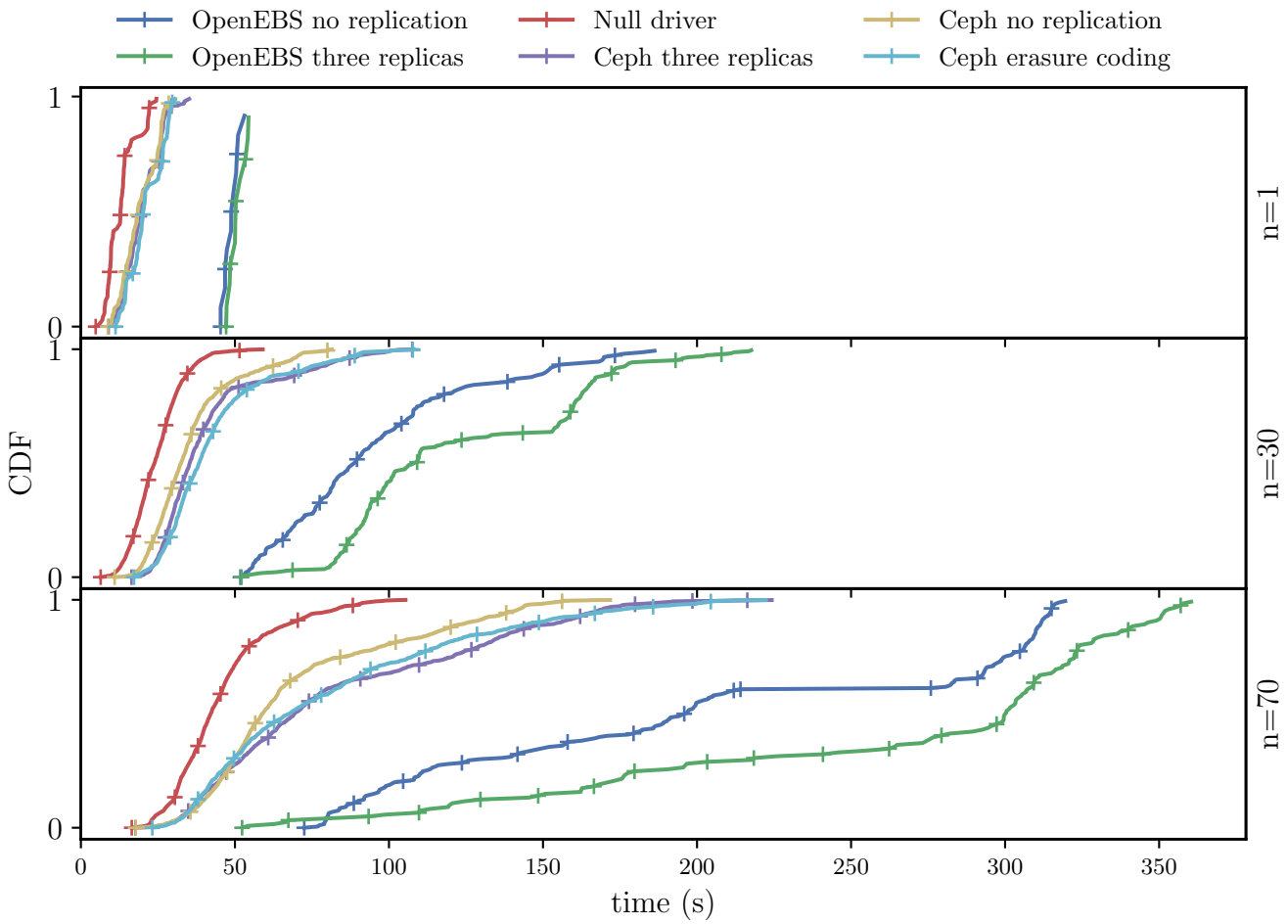


Figure 4.4: CDFs of time required to create and attach volumes for different storage provider configurations. n is the number of simultaneous volume creations. For all storage configurations, increasing the number of simultaneous volume operations increased the average time to create and attach an individual volume.

without performing actual work. The null driver does, however, maintain a list of provisioned volumes so the *ListVolumes* CSI function returns an accurate result. We use the null driver as a baseline to show the maximum possible performance of the underlying Kubernetes cluster.

Each evaluation was conducted five times and unless otherwise noted has a standard deviation of less than 20%.

4.5.2 Performance of Control Operations

In Section 4.3.1 we described the importance of control operations in cloud native workflows. In this section, we demonstrate how the performance of these operations can vary across different storage providers and configurations. We looked at two common storage control operations:

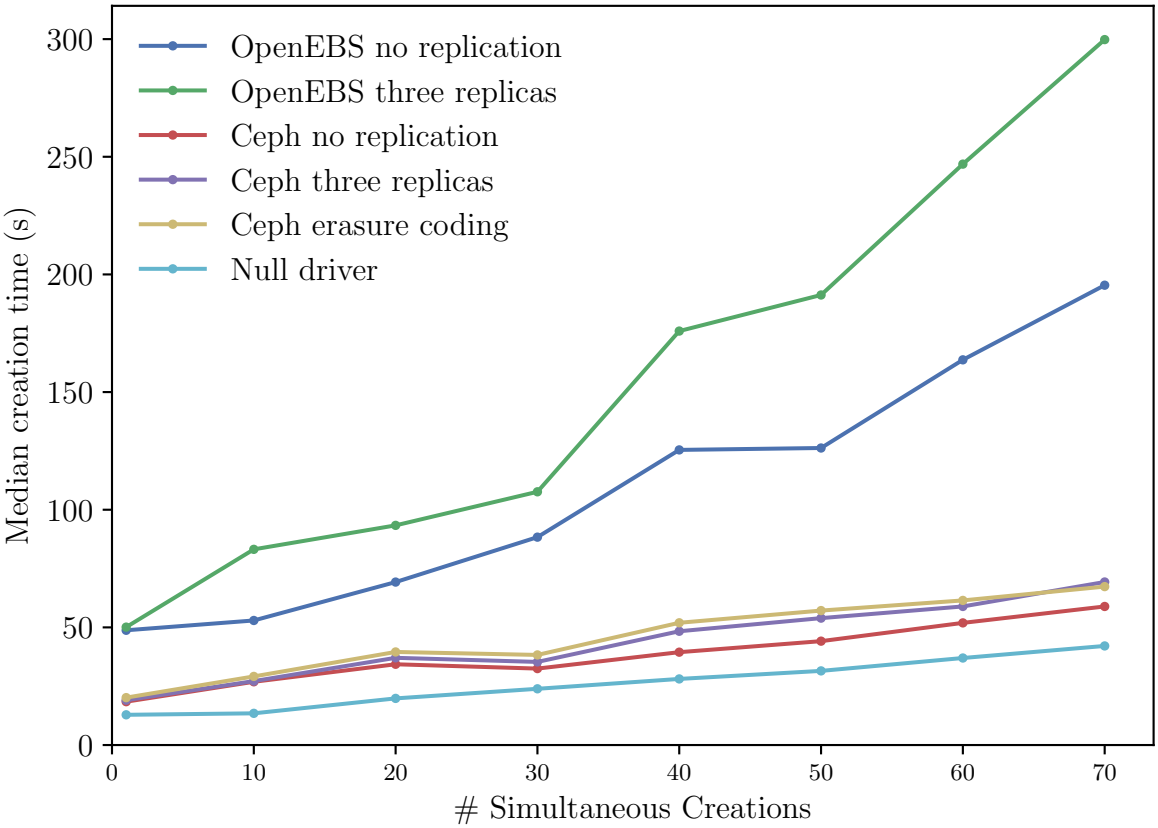


Figure 4.5: Median creation times versus degree of Pod creation parallelism

volume provisioning and attaching.

Our goal was to time how long it took each storage provider configuration to provision a volume and attach that volume to a Pod. To do so, we timed how long it took to create and run new Pods that were attached to volumes. The time to create and run a Pod with an attached volume includes the time taken by the storage provider to provision and then attach that volume. Any additional overhead related to running the Pod is constant across storage configurations.

We ran this test with 1, 10, 20, 30, 40, 50, 60, and 70 parallel Pod creations. Each test ran for five minutes, where we maintained a fixed parallelism level N by starting a new Pod whenever one Pod was created; there were always N Pods in the process of being created. The workload run by each Pod simply exited immediately, so Pods finished running as soon as they started.

We repeated each run five times. Figure 4.4 shows CDFs for Pod start time across all of the Pods created during each of the five runs, for six storage provider configurations. We show CDFs only for three degrees of parallelism (1, 30, and 70) because the CDFs for the intermediate parallelism values follow the trends that are visible from these three. Figure 4.5 shows the median Pod start time for all degrees of parallelism. Figure 4.6 shows the overall volume creation and attachment rate per minute for different parallelism levels. These rates are averaged across each of the five runs and had a standard deviation under 11% of the mean, except for OpenEBS which had

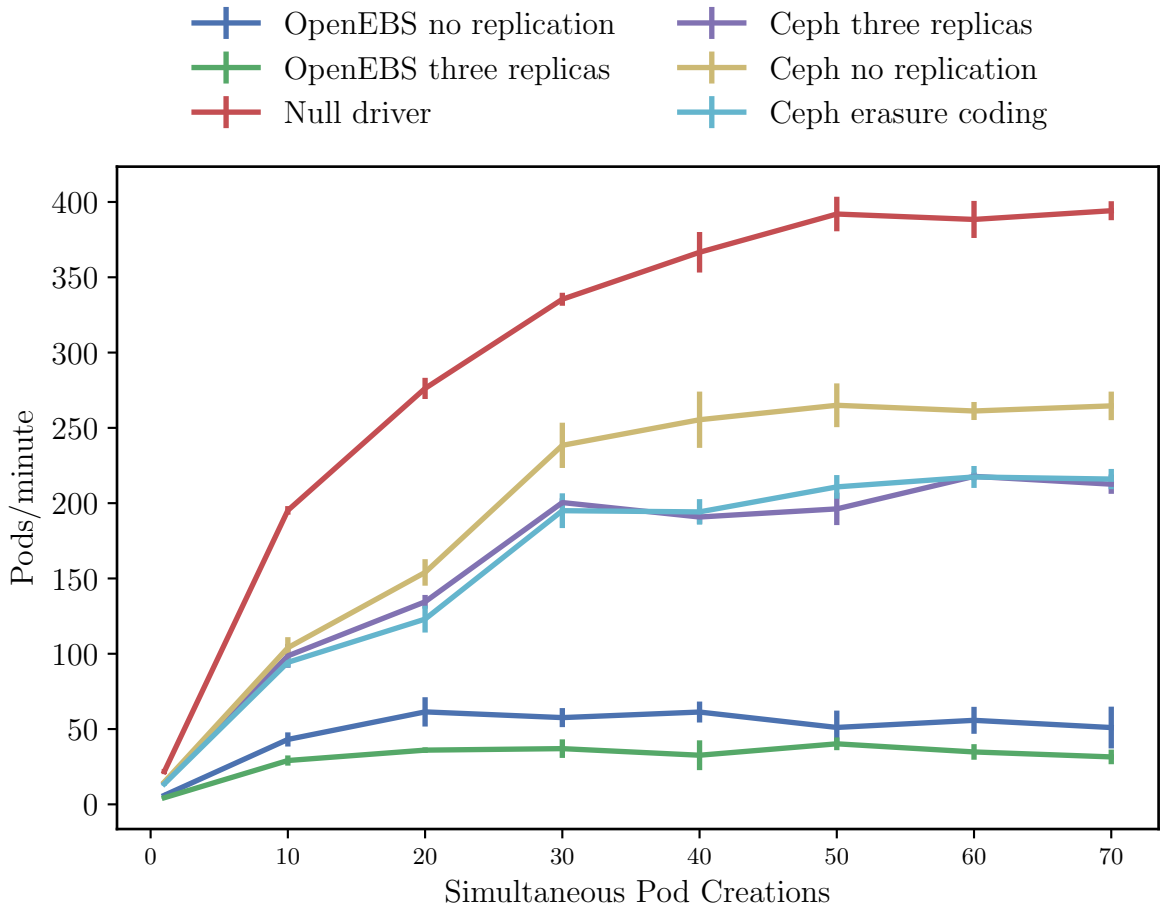


Figure 4.6: Volume creations and attachments per minute, for different numbers of simultaneous operations. The vertical lines at each point shows the standard deviation for volume creation and attachment rate at that point.

standard deviations of up to 30% of the mean. This higher standard deviation can be attributed to the polling architecture which is used throughout Kubernetes and OpenEBS [147], which causes some actions to take sometimes significantly different amounts of time depending on which side of the poll the resource becomes available.

As expected, Pod creation is fastest with the null storage provider. The storage provider configurations with no replication are slightly faster than their replicated counterparts. This is also expected, since volumes with replication require additional resources to be allocated during provisioning.

As the number of simultaneous Pod creations increases, we noticed that subsets of Pods took an increasingly long time to start (see Figure 4.6). Eventually, each of the six storage configurations reached a point where its Pod creation rate plateaus. Note that Pod creation goes through three states: initially it is in a “Pending” state before it can be assigned to a Node. Once the Kubernetes scheduler has assigned the Pod a Node to run on, it moves it to a “Creating” state where container images are downloaded and volumes are mounted. Then, the Pod enters the “Running” state.

As an initial investigation, we counted how many Pods were in each state to identify the bottleneck. We observed that for the null storage provider and the three Ceph configurations, the rate that Pods moved from "Pending" to "Creating" and then from "Creating" to "Running" equalizes when the number of simultaneous Pod creations reaches around 50. At this point, increasing the number of simultaneous Pod creations only increased the number of Pods in the "Pending" state, and did not increase the overall Pod creation rate.

The situation is different for the two OpenEBS configurations. As shown in Figure 4.6, these configurations plateau at a lower rate of around 30 simultaneous Pod creations. When observing the Pod transitions for these configurations, we saw that the rate at which Pods moved from "Creating" to "Running" was low compared to the rate that Pods moved from "Pending" to "Creating" resulting in all Pods being in either "Creating" or "Running" states throughout the test. The Pods in the "Creating" state were all waiting for OpenEBS to finish provisioning and attaching a volume for the Pod. So, increasing the number of simultaneous Pod creations did not increase the overall Pod creation rate, since that rate was limited by how fast OpenEBS was able to provision and attach volumes.

From these experiments we see that although all three storage providers have scalability limits in terms of how many simultaneous Pod creations they support, the source of their limits appear to be different. Whereas the null storage provider and Ceph are limited by the scheduling stage of Pod creation, OpenEBS is limited by its own volume creation and attachment rate.

Overall, the experiment shows that there can be significant differences in the performance of control operations across different storage providers and configurations. This highlights the need to systematically benchmark these kinds of operations to understand their bottlenecks and improve upon them. Conducting this experiment without CNSBench would require starting different numbers of Pods using a tool such as `kubectl`. Whenever a Pod finishes being created, a new one needs to be started, which would be cumbersome to coordinate manually.

4.5.3 Impacts on I/O Workloads

In this section, we demonstrate the impact that control operations, in particular snapshotting a volume, can have on the I/O workload that uses the volume. As described in Section 4.3.1, control operations are executed far more often in cloud native environments than they are elsewhere. Snapshotting is especially common and users take frequent snapshots of their volumes for a number of reasons: periodically, during a long running task to checkpoint progress, prior to making some significant change so rollback to a known good point is possible, or to protect themselves against attacks such as ransomware.

Although previously these operations were executed too infrequently to have a noticeable effect on an I/O workload, this is no longer guaranteed to be the case in cloud native environments. Due to differences in the design and architecture of different storage providers, the degree to which these control operations impact an I/O workload can vary significantly.

To evaluate the impact of snapshotting operations, we used CNSBench to run three instances of MongoDB [164] with ten clients each. The clients ran YCSB Workload A [44] (consisting of a mix of reads and updates) for twenty minutes to reach steady state; the volumes holding the MongoDB databases were snapshotted every thirty seconds.

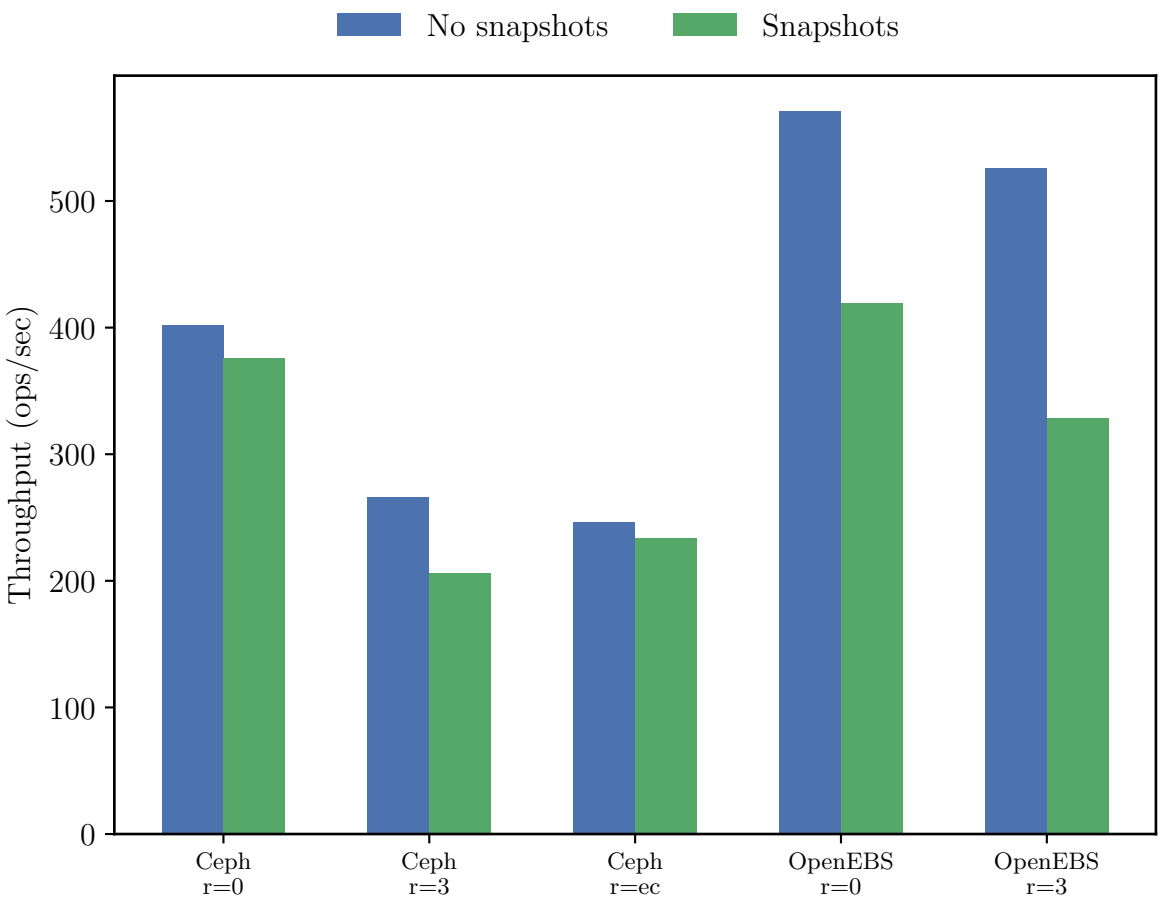


Figure 4.7: Effect of snapshotting on I/O workload. r=0 indicates zero volume replicas, r=3 indicates three volume replicas, and r=ec indicates erasure coding.

Figure 4.7 shows the per-client throughput in terms of operations per second for five storage provider configurations, with and without snapshotting. The throughput values are averaged across all thirty YCSB clients.

Overall the results show that snapshotting reduces the throughput across all configurations. The decrease in throughput is more noticeable for OpenEBS (27% and 38% for zero and three volume replica configurations, respectively) than for Ceph (up to 22% for three volume replicas but as low as 5% and 6% for erasure coding and zero replication configurations, respectively). We found that although the average throughput decreased with snapshotting across all OpenEBS YCSB clients, the decrease was more pronounced for some clients than others. For those clients, we observed that the maximum latency reported by YCSB was much higher than the average maximum latency. In addition, these clients reported extended periods (30+ seconds) when zero operations were executed. During these periods with zero operations, the Mongo database reported that some queries were taking a long time to be processed.

One possible explanation is the fact that OpenEBS quiesces and suspends I/O while a snapshot operation is in progress [148]. During that time, any writes issued by Mongo cannot complete.

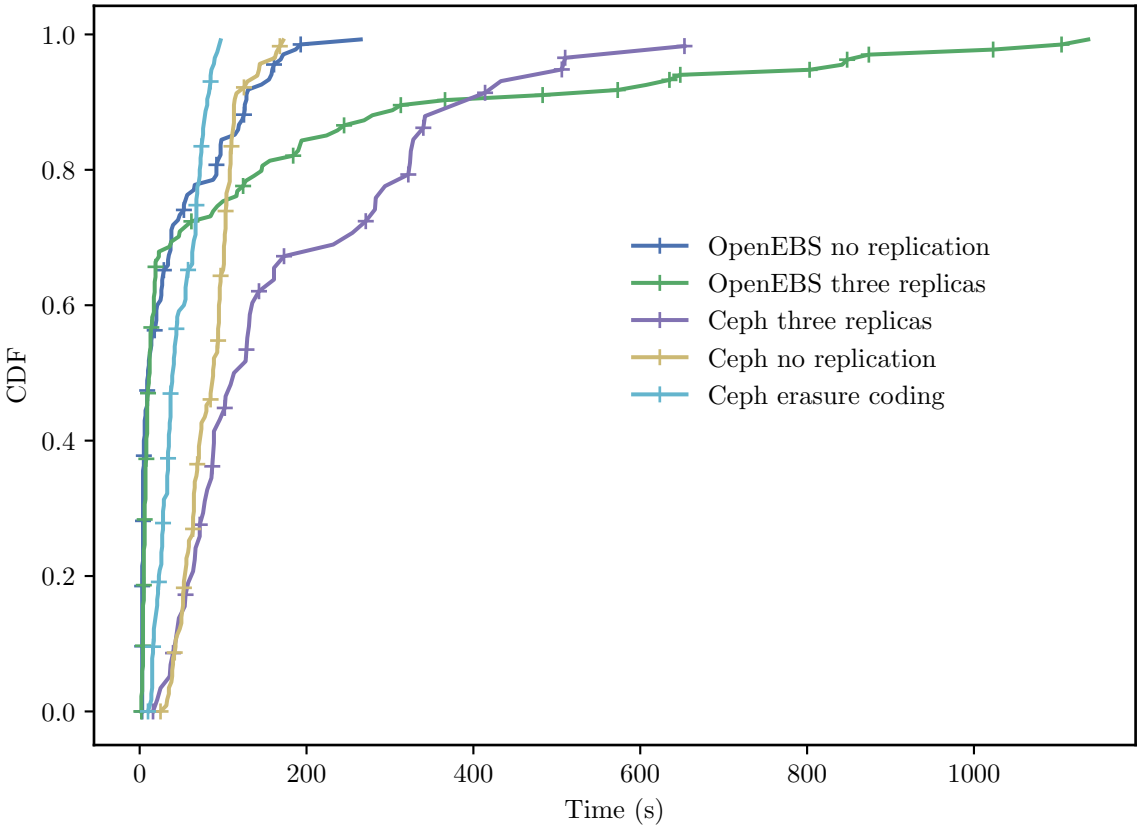


Figure 4.8: CDF of snapshot creation times for different storage provider configurations.

Some of these periods of suspended I/O lasted several seconds, which could explain the periods when no operations could be executed by the clients and the reduction in overall throughput. We analyzed the distribution of throughputs for all clients and found a long tail with many clients timing out after several quiescing periods, then retrying.

Ceph does not quiesce [37] I/O during a snapshot and we did not observe the same spikes in maximum latency that we observed with OpenEBS. We did observe some of the same periods with zero completed operations that we saw with OpenEBS, and also observed the same complaints of slow queries from the Mongo logs. One possible explanation is that there was an increased load on Ceph: with snapshots, around four times as many objects were created in the underlying RADOS object pools compared to no snapshotting.

To create a new snapshot in Kubernetes, users create a Snapshot resource. This resource is created immediately. However, the underlying snapshot is not necessarily ready right away. Figure 4.8 shows a CDF of how long it took after creating a new Snapshot resource until the storage provider reported that the snapshot was actually ready to be used.

Both Ceph and OpenEBS implement copy-on-write snapshots, so it is expected that for most storage configurations, snapshots became available nearly as fast as the Snapshot resources were created. However, some configurations exhibited a long tail where snapshots took several minutes to become ready. For example, although the median time to become ready for snapshots on

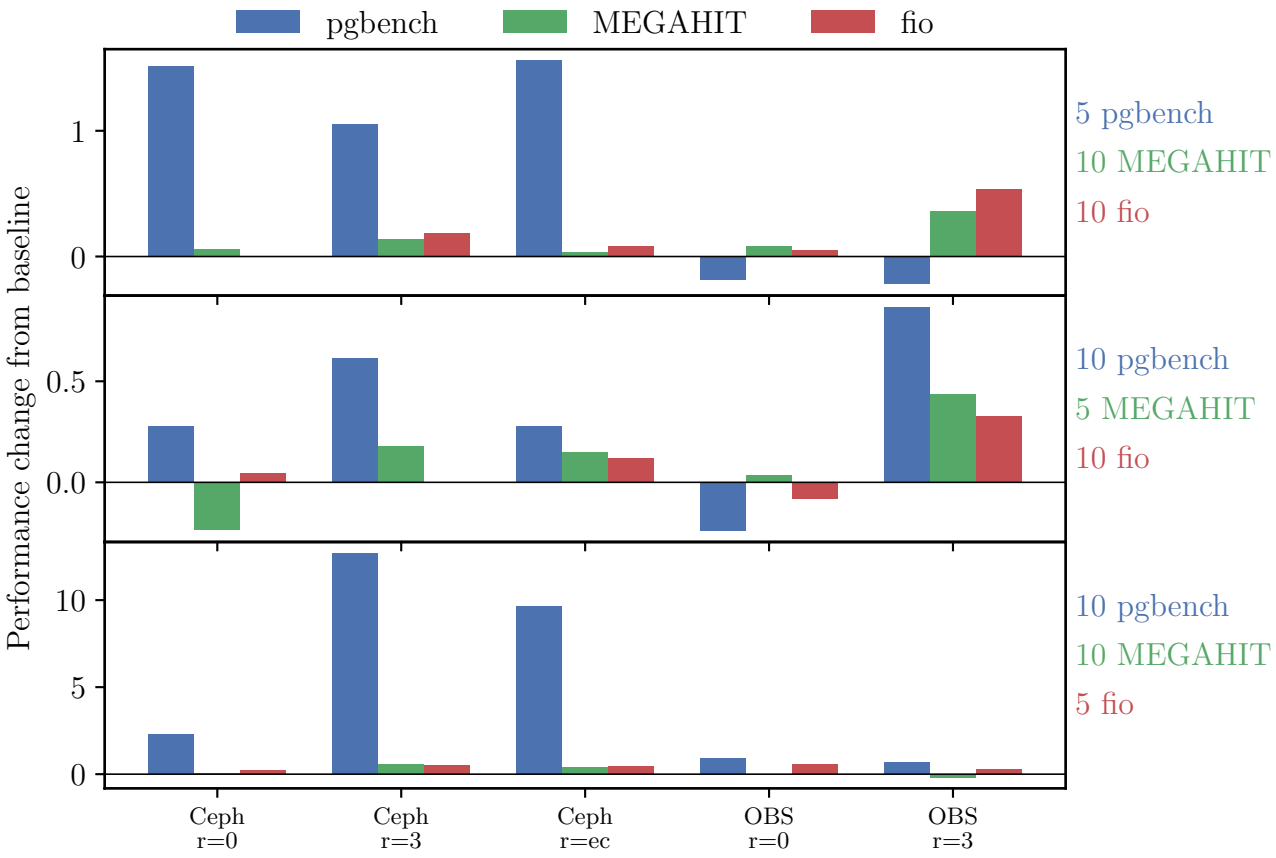


Figure 4.9: Change in performance compared to baseline, for three different ratios of I/O workload on five different storage configurations.

OpenEBS with three volume replicas was 12 seconds, 10% took longer than 310 seconds and 5% took longer than 702 seconds. The interface between Kubernetes and the storage provider's CSI driver is the Kubernetes Snapshot Controller [112]. When we analyzed the logs for this container, we found that the CreateSnapshot CSI calls for some snapshots were timing out due to slow I/O on the underlying disks used by the storage provider. For some unlucky snapshot instances the CreateSnapshot call would repeatedly timeout, resulting in snapshot creation times of several *minutes*. One interesting observation was that even when the Snapshot Controller aborted its CreateSnapshot call (due to the timeout), the storage provider would still finish creating a snapshot. However, the Kubernetes Snapshot Controller had already timed out, thus missing the successful response from the storage provider.

Running this experiment without CNSBench would require specifying and creating each of the resources required to run MongoDB and YCSB (Pods, PVCs, Services, etc.). Then, while YCSB ran, the user would need to create snapshots of each of the volumes being used by specifying the snapshot resources in YAML and instantiating the resources with a tool such as `kubectl`.

4.5.4 Orchestration

One of the core CNSBench capabilities is to make it easy to run various mixes of I/O workloads. This is needed since the alternative, to manually choose and assemble workloads together to form a representative combined workload, is infeasible due to the diversity of workloads in cloud native environments.

One potential use case for this task is to determine which storage configuration is best suited for a particular set of workloads. Another might be to help influence scheduling decisions, such as which workloads to run simultaneously.

To demonstrate CNSBench's orchestration capabilities, we ran multiple instances of three different workloads: (1) MEGAHIT [119], a bioinformatics tool that processes genetic data; (2) fio [63] for generating an intense I/O workload of mixed random reads and writes; and (3) the PostgreSQL [155] database with a workload generated using its benchmark tool pgbench [153]. Each instance of the PostgreSQL workload ran a distinct pair of database and client. Out of each of the workloads, fio was the most I/O intensive, followed by pgbench. Both fio and pgbench spent most of their time waiting for I/O, whereas MEGAHIT was mostly CPU bound.

We tested four different workload mixes: a baseline with ten independent instances of each workload, and then three additional mixes with ten instances of two of the workloads and five of the third. For MEGAHIT and fio we measured the total time to run a fixed load; for pgbench we measured the average throughput after running for ten minutes. This was necessary since the different storage provider configurations performed significantly different, so it would be impractical to evaluate using a fixed amount of work.

Figure 4.9 shows the changes in runtime and throughput, normalized to the baseline values, for different workload mixes and storage providers. The baseline throughputs for pgbench are 5.2, 0.37, 0.38, 85, and 16 operations per second for Ceph (no volume replication), Ceph (three volume replicas), Ceph (erasure coding), OpenEBS (no volume replication), and OpenEBS (three volume replicas), respectively. MEGAHIT had baseline runtimes of 309, 910, 609, 185, and 324 seconds, and fio had baseline runtimes of 427, 816, 699, 923, and 2478 seconds, respectively.

The largest increase in performance of 3.2× is for pgbench when the number of fio instances is reduced. This makes sense: the Ceph storage configurations shows the largest increase in pgbench performance, since pgbench's baseline performance on Ceph is much worse than on OpenEBS so there is a larger potential for improvement. Also, pgbench and fio are both I/O-intense workloads, *i.e.*, reducing the number of fio instances would help pgbench, but not MEGAHIT.

The workload that had the overall smallest impact on performance is MEGAHIT. This is also expected as fio and pgbench are mainly I/O bound while MEGAHIT is mainly CPU bound and hence reducing the number of MEGAHIT instances does not free up significant I/O resources.

These results demonstrate the variability in storage provider performance, and the utility of being able to easily compose and run diverse sets of workloads at various mixes. Conducting this experiment on Kubernetes without CNSBench would require creating all of the resources required for a workload (PVCs, Pods, Services, etc.) manually, for example by specifying them in YAML and passing the YAML to a tool such as `kubectl`. To run multiple instances of a workload, the user would need to specify multiple copies of each resource, making sure to give each copy a unique name and updating references to resources accordingly. This would need to be repeated

Benchmark	Lines
Volume creation and attachment § 4.5.2	19
YCSB and MongoDB, no snapshots § 4.5.3	35
YCSB and MongoDB, with snapshots § 4.5.3	54
Multiple workloads § 4.5.4	72–117

Table 4.1: Number of lines needed to specify CNSBench benchmarks used during evaluation.

for each workloads mix being evaluated. Synchronizing the start of each workload would need to be done manually. For example, to synchronize the start of multiple MongoDB+YCSB workloads the user would need to first start each MongoDB database pod, then wait for the databases to be initialized, and then run each instance of their YCSB benchmark.

4.5.5 Benchmark Usability

Requirement 4 in Section 4.3.2 states that CNSBench should be easy for users to configure and run. Although usability is often subjective, one metric that can be used to estimate ease of use is the number of lines necessary for specifying a workload. Table 4.1 shows the number of lines needed to specify each of the benchmarks used in this evaluation section.

Overall a user can specify the complex, distributed, and diverse workloads in just 19–117 lines of configuration. The workloads used in Section 4.5.4 require slightly longer specifications as they contain multiple instances of the same sub-workload, which currently results in duplication in the CNSBench's benchmark specification. We plan to eliminate such repetitions in the future to make using CNSBench even simpler.

4.6 Conclusion

Although measuring storage performance was always an important topic, its relevance has escalated in recent years due to the increased demand to reliably move containerized applications across clouds. Furthermore, I/O patterns of applications have evolved, exhibiting higher density, diversity, dynamicity, and specialization than before. Perhaps most importantly, storage services now experience a high rate of *control operations* (*e.g.*, volume creation, formatting, snapshotting), which directly impact the performance of applications that call them and indirectly influence the I/O of other applications in a cluster. Existing storage benchmarks, however, are not able to model these new cloud native scenarios and workloads holistically and faithfully.

In this chapter we presented the design of CNSBench—a storage benchmarking framework that containerizes legacy I/O benchmarks, orchestrates their concurrent runs, and concurrently generates a stream of control operations. CNSBench is easy to configure and run, while still being versatile enough to express a high variety of real-world cloud native workloads. We used CNSBench to evaluate two cloud native storage backends—OpenEBS and Ceph—and found several differences. For example, our evaluation shows that the maximum rate of control operations varies significantly across storage technologies and configurations by a factor of up to 8.5×.

Future work We plan to work on extending the library of I/O workloads with I/O "kernels" that represent microservices, and also improve the benchmark specification language to make the syntax more concise and avoid having to duplicate sub-workloads. Further, we will work on collecting I/O and control operation traces from production environments, analyze them, and create corresponding profiles for CNSBench. Our longer term plans including finding and fixing performance bugs using CNSBench, and even developing our own efficient storage solution.

We hope our benchmark will be adopted by storage and cloud native communities, and look forward to contributions.

Chapter 5

F3: Serving Files Efficiently in Serverless Computing

Serverless platforms offer on-demand computation and represent a significant shift from previous platforms that typically required resources to be pre-allocated (*e.g.*, virtual machines). As serverless platforms have evolved, they have become suitable for a much wider range of applications than their original use cases. However, storage access remains a pain point that holds serverless back from becoming a completely generic computation platform.

Existing storage for serverless typically uses an object interface. Although object APIs are simple to use, they lack the richness, versatility, and performance of file based APIs. Additionally, there is a large body of existing applications that relies on file-based interfaces. The lack of file based storage options prevents these applications from being ported to serverless environments.

In this chapter, we present F3, a file system that offers features to improve file access in serverless platforms: (1) efficient handling of ephemeral data, by placing ephemeral and non-ephemeral data on storage that exists at a different points along the durability-performance tradeoff continuum, (2) locality-aware data scheduling, and (3) efficient reading while writing. We modified OpenWhisk to support attaching file-based storage and to leverage F3's features using hints. Our prototype evaluation of F3 shows improved performance of up to 1.5–6.5$\times$ compared to existing storage systems.

5.1 Introduction

Serverless platforms have already proven their utility in running small web-oriented tasks. They are approaching a turning point, however, where their on-demand computation is expanding to a wider range of applications [91, 145, 46]—possibly any application. To this end, serverless platforms have been relaxing constraints and adding features, for instance, allowing users to run arbitrary containers and increasing execution time limits to support longer-running actions. Here, an "action" is a snippet of code or a standalone executable, and a serverless application is made up of one or more actions [10, 24, 14].

Storage access, however, remains a pain point for generic applications in serverless environ-

ments. Serverless platforms typically support only object-based storage. Object is a natural choice for the short, stateless, web-oriented tasks for which serverless platforms were originally designed and used; but more generic applications frequently need functionality not supported by traditional object storage—for example file-based access to data, the ability to perform in-place modifications, support for concurrent writers, and the ability to read data as it is being written. The lack of support for these features has held serverless computing back from becoming a generic computational platform.

Although most serverless platforms still do not offer a way to connect file based storage to serverless applications (*e.g.*, IBM Cloud Functions [82], Google Cloud Functions [73], OpenWhisk [149], or Knative [100]); some (*e.g.*, AWS Lambda) have recently added support for file-based storage [11]. This is encouraging, as it indicates that industry has recognized the need for file-based storage in serverless applications. Existing file systems, however, were not designed for serverless platforms and lack important features that would benefit serverless applications. In particular, existing file systems lack functionalities that could accelerate *intermediate data transfer* between the individual actions that make up a serverless application.

Applications in serverless environments are often split into multiple components forming pipelines, where one component writes its output data sequentially to storage, the next component reads the data as input, then the system discards the intermediate data. By specifically facilitating this usage pattern, a storage system can improve data access and transfer performance. We identified three ways a storage system can aid this pattern: (1) storing the intermediate data on less durable, lower-latency local storage, (2) providing hints about the location of data to serverless schedulers so that subsequent stages of a pipeline can be scheduled close to the data, and (3) making it possible for the next stage of a pipeline to begin reading before the previous stage has finished writing.

Durability vs. performance tradeoff Durability provided by storage systems often comes at the cost of performance. For instance, in our experiments, disabling durability features (*e.g.*, erasure coding) increased read/write bandwidth by 42–45%, and using a local disk rather than networked file system further increased read/write bandwidth by 39–86%.

The data transferred in serverless applications is usually *ephemeral* (*i.e.*, short lived) and is needed only until it has been consumed by the reader. This enables a different durability-performance tradeoff to be made. For example, ephemeral data does not necessarily need strong durability features such as replication or erasure coding that are provided by many storage systems. Although durability features can sometimes be disabled in a given storage system, they are typically configurable only at volume or file system granularity. As a result, it is difficult to optimize for workloads that store both ephemeral and non-ephemeral data: both must exist at the same point along the durability-performance continuum.

Locality For data to remain local to a server, the serverless platform's scheduler needs to know where the data a serverless application will consume are located within the cluster. Current storage systems either do not convey this information to serverless platforms, or are designed such that the information is not even applicable (*e.g.*, if, for data protection, the data is distributed across multiple nodes in the cluster). Either way, the result is that data transfers between components

within a serverless application consume network bandwidth and incur the performance penalty associated with transmitting data across the cluster's network.

Reading while writing Finally, it is often desirable to process data in a streaming fashion, *i.e.*, to read and process data while it is written to a file. Doing so speeds up end-to-end processing because a subsequent stage can begin without having to wait for the previous stage to finish. In object storage, it is not possible for an object to be open by a writer and reader at the same time. In distributed file systems, however, it is possible but file systems often make the conservative assumption that when a file is open for reading by one client and for writing by another client, that both clients must use unbuffered file accesses to ensure that readers and writers maintain consistency [39].

Unbuffered access significantly slows both the reader and the writer, negating any performance benefit of the read-while-writeaccess pattern. For data transfer in serverless applications, this is an overly conservative assumption since both reader and writer access the data only sequentially (*i.e.*, data is never modified once written).

In this chapter we address the storage access and data transfer problems for serverless environments. First, we added file system support to a popular open-source serverless platform, OpenWhisk [149], to demonstrate how existing file storage solutions can work with a serverless platform. Next, we implemented a stackable file system, F3, that is designed to optimize the transfer of data between serverless applications and the individual components of a serverless application. F3 distinguishes ephemeral data from that requiring high durability, and transparently directs ephemeral data to node-local disks. This enables F3 to perform up to 6.5× faster when writing data and $2.7\times$ faster when reading data compare to the traditional durable storage.

F3 *further* optimizes data transfer by tracking the location of ephemeral files and exposing that information to serverless schedulers. We modified OpenWhisk's scheduler to use data location information when selecting the server to run the function, which in one experiment reduced network traffic used for data transfer from 2GB down to *zero*.

We designed F3 to stack over existing durable storage systems (*e.g.*, Ceph [39], Lustre [41], and GPFS [74]), making F3 a flexible and transparent extension to existing storage solutions. The resulting file system namespace makes both durable and ephemeral files visible to serverless applications.

Though F3 is generic and can be applied in different environments, we focused our empirical evaluation on a specific, rapidly growing use case—Edge Computing. Industrial edge computing is a new market that is predicted grow from $18B to $31B by 2025 [69]. Edge data centers are smaller facilities that range in size from street-side cabinets to cargo container-like structures that house a limited amount of server infrastructure. By having a smaller form factor than typical data centers (typically only 3–10 servers), edge data centers are relatively easy to move and deploy, making them a good fit for housing IT infrastructure at the edge. Serverless computing enables higher resource usage efficiency in these resource constrained environments through its fine-grained sharing [66]. Our experimental platform, workloads, and evaluation methodology are tailored to serverless computing at the edge.

This chapter makes the following contributions:

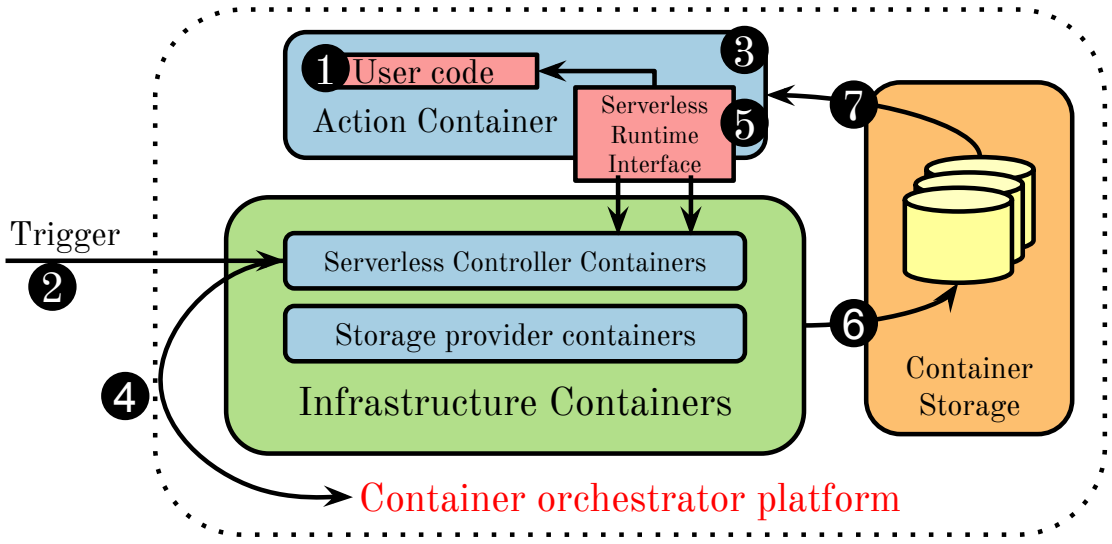


Figure 5.1: Blueprint architecture of edge serverless platform

1. We describe the case for using file systems in serverless computing and extended OpenWhisk to enable attaching actions to file-based storage;
2. We designed and implemented F3, a file system that extends existing storage systems to enhance data transfers between serverless actions;
3. We evaluated F3 and several alternatives for edge computing; and
4. We have made F3 and our modifications to OpenWhisk available as open-source software: *https://github.com/filesystems-for-serverless*.

5.2 Background

In this section we give an overview of how serverless platforms operate (*e.g.*, AWS Lambda [9], Apache OpenWhisk [149]). Figure 5.1 depicts a serverless platform running on top of a container orchestrator.

Operation Serverless platforms run processing ❶ on demand in a containerized environment [16]. Traditionally this processing consisted of snippets of code referred to as "functions." As serverless platforms have become more generalized, more and more of the processing is done by standalone executables (*e.g.*, an entire webserver or video processing utility). The term "function" seems insufficient for these more generalized and complex workloads, so we use the more generic term "action" to refer to both traditional function-style processing and newer more generic processing.

Actions run when triggered ❷ by a request to an HTTP endpoint. The trigger can be initiated in response to an event such as an upload to an object store. Information related to the trigger is passed to the action as parameters (*e.g.*, uploaded object name).

Running actions The containers that run actions ❸ are often managed by a container-orchestration platform such as Kubernetes [165]. When an action is triggered, if there is an appropriate container already running, then that container runs the action. This is referred to as a *warm start*. If no suitable container exists, the serverless platform creates a new container for the action by making a request to the container orchestrator ❹; this is a *cold start*. In either case, a scheduling decision has to be made. If there are multiple warm containers suitable for an action, the serverless platform's scheduler must choose that container to run the action. If a cold start is required, then the containers orchestrator's scheduler must decide the cluster node on which to start the container, possibly using hints from the serverless platform's scheduler.

To avoid the overhead of cold starts, serverless platforms keep action containers running for some time after an action has executed. If the container's resources are needed for something else, however, then the container can be stopped as soon as the action ends. In either case, cluster resources are reserved and paid for only while the action is actually running.

Building and deploying actions In early serverless offerings, actions were built by writing a snippet of code in a language such as JavaScript or Python. When triggered, the code was run using a container image built by the serverless platform. This approach allowed developers to focus solely on their code, but was somewhat restrictive in that it limited the languages supported. Also, because the serverless platform provided the execution environment, developers had little flexibility in the choice of libraries, runtimes, and other external resources.

The simplicity inherent in this approach is still sometimes desirable, and "Function as a Service" (FaaS) platforms continue to offer this method of building and deploying actions. For many use cases, however, more sophisticated actions are needed. These actions might use external libraries, have multiple executables, or require a specific execution environment (*e.g.*, a specific Linux distribution). To support these actions, most modern serverless platforms now allow developers to run an arbitrary container image in response to a trigger. On startup, these containers run a "Serverless Runtime Interface" executable ❺ that interfaces with the serverless platform. When triggered, the container image runs the Serverless Runtime Interface, which retrieves the action's input parameters, executes the action, and returns the results to the serverless platform. Thus, any application that can be containerized can be run as an action on a serverless platform. This approach opens serverless platforms to many more use cases than were originally designed.

Storage Early code snippet-based actions were completely stateless, thus did not require access to persistent storage. When stateful serverless actions were later introduced, object stores were the recommended [124, 43] means to hold the state.

This made sense because (1) early serverless applications appeared mainly in web environments where object storage has been the norm, and (2) object stores are easy to access, requiring only the ability to form an outbound HTTP connection.

Although there are a wide variety of file and block storage options [93, 132] that container orchestrators can provision ❻ and attach ❼ to containers, current serverless platforms have not taken advantage of them.

5.3 Storage for Serverless Computing

In this section we first discuss the differences between file and object storage, then describe features existing file systems lack that would improve efficiency for serverless applications.

5.3.1 Object Stores vs. File Systems

In most serverless platforms, the only storage option available to actions is object storage. Object-based storage uses a key to identify an item of data, is typically accessed using through HTML requests, and supports operations PUT, GET, and DELETE. For many serverless applications, this interface is completely adequate and appropriate. We are not suggesting that the option of object storage in serverless platforms should be taken away.

But many applications that run in generic container images expect a file based interface, where files are identified by their names in a hierarchical namespace, and are accessed using operations such as `open`, `read`, and `write`. While file-to-object translation layers that can be embedded with the application exist, they generally do not support the richer functionality of files—including in-place modification, read-after-write consistency and directory-level operations—thus are not adequate for all applications.

Further, file systems typically provide higher performance than object stores [85, 158, 161]. Although high performance object stores could be implemented, applications that require high performance today are mainly file based [144].

One of the commonly cited benefits of serverless platforms is their near-limitless scalability. It might therefore seem counter-intuitive to suggest bringing file systems, often regarded as having limited scalability, to serverless platforms. Nevertheless, several major cloud providers have added file system support to their serverless platforms. This reinforces our belief that file system support is necessary, and that if serverless platforms are to take the next step toward becoming a generic computing platform, they must support file in addition to object interfaces.

5.3.2 Shortcomings of Existing File Systems

Existing shared file systems such as NFS and CephFS can provide storage for serverless applications. However, these file systems were not designed with serverless platforms in mind and lack features that would benefit serverless environments. Three such features are: (1) support for ephemeral (short-lived) data, (2) the ability to schedule actions based on where their data is located, and (3) support for reading files as they are being written.

Ephemeral data Serverless applications make heavy use of ephemeral data—one that is short lived and that can be easily recreated. Ephemeral data comes from a variety of sources. For example, pipelines that span multiple actions may produce intermediate results generated by one action, consumed by another, and then discarded. Sensor and other user data generated at the edge is often filtered and pre-processed, with much of the original raw data not retained. Moreover, resources such as machine-learning models are frequently replaced with updated versions.

Many storage systems provide durability and reliability features such as replication or erasure coding. These features come with a performance cost. Since ephemeral data does not need these features, there is an opportunity to trade off decreased data reliability for increased performance.

In the case of node or disk failure, ephemeral data can be recreated by rerunning the original actions that created it. Detecting an action failure and rerunning the original actions requires a serverless execution framework that is beyond the scope of this chapter; but we note that a file system could reasonably identify when a disk fails (*e.g.*, EIO errors) and inform the serverless execution framework. This would allow the execution framework to differentiate between regular action failures (*e.g.*, due to an application error) and action failures due to missing or corrupted data caused by disk failure. How a serverless execution framework handles such errors is part of the larger problem of serverless application orchestration (see Section 5.7).

Data locality-aware scheduling When running an action, the serverless platform must decide where to run that action. Assuming the platform uses containers to run actions, this entails either (1) finding an available already running container and assigning the action to that container, or (2) starting a new container to run the action.

There has been a significant amount of work done in trying to avoid cold starts, since starting up a new container to run the action can significantly increase action latency and overall runtime. However, another factor that must be taken into account is the location of the data needed by the application. Running the action close to the data avoids the delay and overhead of moving the data to where the action runs.

Most existing storage systems do not provide the necessary data-locality scheduling hints. Those that do, provide them only at a volume granularity, too coarse for per-file-based scheduling. For example, with volume-level scheduling hints, an application's actions cannot simply write their output to a common output directory. Other systems that have incorporated data locality into serverless action scheduling (i) require applications to be structured in a specific way (*e.g.*, as a DAG) [35] and (ii) require information about the structure of the application before the application runs [35, 125].

Reading-while-writing Pipelines where one process generates data as another process consumes it are common in Unix environments, especially in the form of Unix pipes (*e.g.*, `procA | procB`). Such a pipeline can reduce end-to-end application run times since the second process does not need to wait for the first process to finish before starting its processing.

This technique requires the two processes to share a kernel to facilitate piping the output from one process to the input of the next, and so porting such a pipeline to a serverless platform is not trivial. Note that in Unix pipes, the pipe's data is itself ephemeral and lives temporarily in kernel buffers.

One workaround is to use a temporary file as an intermediary, *e.g.*, `procA >/tmp/f & procB </tmp/f`. This solution can fail, however, since `procB` may read all of `/tmp/f` and exit before `procA` has finished writing, leaving some data unprocessed by `procB`.

A better workaround is to use an intermediary file, but to also have `procB` wait to exit until after `procA` closes `/tmp/f`. This is easy to do with the standard Unix utility `tail`:

```
procA >/tmp/f & PID=$!; tail -pid=$PID -f /tmp/f | procB
```

Here, `tail` waits for additional data until `procA` exits.

This works on a single system where `tail` is able to test if `procA` has exited. However, if `procA` and `procB` are running in different serverless contexts, this workaround does not work.

Because pipelines are such a common idiom in serverless workflows, a file system that optimizes this pattern and increases parallelizablity between stages is highly desirable. When an intermediate file is used to communicate data between two actions, the file system is in a unique position to block the reader as necessary to wait for a concurrently running writer to add additional data to the file, returning end-of-file indication to the reader only after the writer has finished and closed the file.

5.4 Design

We have designed a proof of concept file system, F3, that has all of the desired properties identified in Section 5.3. Figure 5.2 depicts F3's architecture. F3 is designed to layer on top of an existing durable file system, extending it with features benefiting serverless applications. F3 provides faster access to ephemeral data by storing it separately from non-ephemeral data on local, less durable storage without features like replication or erasure coding. Since ephemeral data is stored on node-local devices, F3 interfaces with the serverless platform to aid in scheduling actions on the nodes where their data resides. In the event that this is not possible (*e.g.*, because the load on that node is too high), F3 transparently and efficiently handles transferring the data between nodes.

We describe the design of the three serverless data transfer features in more detail below.

Ephemeral data support F3 provides a common file system interface for both ephemeral and non-ephemeral data. To do this, F3 merges (1) a distributed, reliable, networked file system with (2) a file system on a fast local disk, and exposes a single mount point. Applications use the mount point exposed by F3, and F3 transparently writes file contents to either the networked file system or to the faster local file system.

The networked file system should be a file system that is accessible by every node in the cluster, such as CephFS or NFS. Each node should have its own local data store for ephemeral data. This, for instance, is the case in a hyperconverged architecture, where storage is provided by aggregating disks attached to each compute node rather than using dedicated storage servers.

In our current implementation, users can mark a file or directory as ephemeral by setting the appropriate extended attribute on the file or directory or just use a special predefined file name extension. All data under an ephemeral directory is automatically marked ephemeral. In many workflows an application developer or user can easily identify which files are intermediate hence contain ephemeral data. In other cases some files (*e.g.*, stored in `/tmp`) or opened with O_TMPFILE, could be automatically designated as ephemeral. In the future, we can explore using more advanced automation for identifying ephemeral data.

For each volume, F3 creates a different top level directory on the local and networked file systems. This keeps the volume namespaces separate, so files in separate volumes can share the same name and path. Under this top level directory, F3 maintains the same directory hierarchy on both the local file system and the networked file system: the only difference is where the a file's contents are stored. It creates an empty file as a placeholder in the underlying file system where the file is not stored (*e.g.*, the networked file system if the file is an ephemeral file). However, if a F3 volume is created by extending an existing networked file system volume, F3 does not require any initial synchronization. Instead, F3 lazily creates the network file system's directory hierarchy on the local disk as needed.

F3 uses extended attributes on the copies of the files on the networked file system to track F3-specific metadata about a file. For example, we use extended attributes to mark whether the file is ephemeral, and if so which nodes in the cluster have a copy of that file's data. Storing metadata in the network file systems provides high durability for metadata. When a file is opened by an application, F3 checks the file's extended attributes to determine if the file is ephemeral: if so, it opens the copy of the file on the ephemeral data store and returns the file descriptor to the application. Otherwise, F3 opens the copy of the file on the networked file system and returns that file descriptor. If the extended attributes are missing, F3 assumes that the file is non-ephemeral. This can happen if F3 is extending a networked file system that has already been populated with data, for instance.

When F3 opens an ephemeral file, it first checks if the file contents are available locally. If not, F3 uses the extended attributes to find which nodes in the cluster have the file's contents. F3 then uses a per-node client/server communication to do a point-to-point, direct, efficient transfer of the file contents. As soon as the network transfer is initiated, F3 begins transferring the entire file and returns a file descriptor for the file to the application, which can then read the file as it is being downloaded.

The original copy of data is not deleted, and the data on either node can be used by subsequent actions. For our current implementation, we assume that ephemeral data is written once [97] so this copy of data does not need to be updated. As most ephemeral serverless data is written only once, this is a reasonable assumption. At this time we consider it the responsibility of the application developer to ensure that this assumption holds.

If a node or local disk fails and ephemeral data is lost, the action that created the data has to be re-run. This is consistent with the typical requirement that actions are idempotent [12, 86], and the fact that actions may be automatically re-run by the serverless platform in the event of certain kinds of errors [116].

Our current implementation of F3 does not include any garbage collection to delete old data on the local disk. A simple approach would be to delete data as needed when the disk fills up, using an LRU algorithm to choose what data to delete. If a single action writes enough ephemeral data to fill up the local disk by itself, the current implementation of F3 would return `ENOSPC` to the application. Other approaches might be to have F3 copy the ephemeral data to the shared file system, store the data partially on the local disk and partially on the shared file system, or to have the serverless platform rerun the action and have F3 treat the data as non-ephemeral during the second run. We leave exploration of these options, as well as an implementation of a garbage

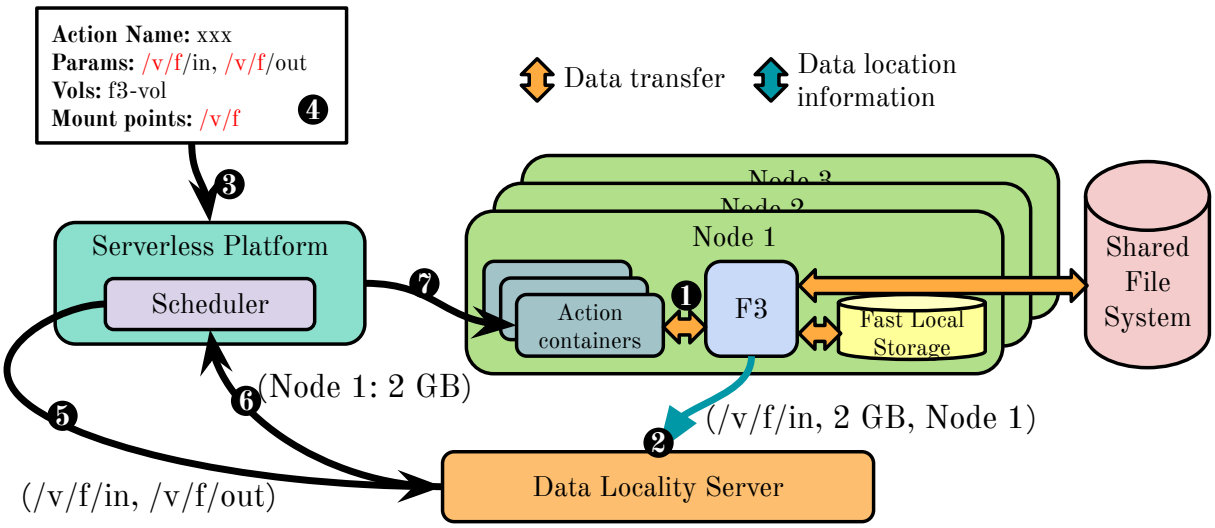


Figure 5.2: CNSBench architecture and locality-aware data operations

collection mechanism, to future work.

Data locality hints for action scheduling Collectively, the F3 file system drivers which run on each node in the cluster know what files are present in their local ephemeral data store. If the serverless platform's scheduler knows what files an action will access, the scheduler can ask the F3 file system for the location of the data and use that information in deciding where to schedule the action.

Rather than making the scheduler query each local instance of F3, F3 includes a simple server that centralizes this data locality information. Each local instance of F3 sends information about what files are on its node to this data locality server. The locality information is sent to the server asynchronously, so the server should not become a bottleneck in data operations.

Figure 5.2 details how the data locality feature in F3 works. When an ephemeral file is written ❶ to an F3 file system, the local instance of F3 on that node sends ❷ the file name, file size, and node name to the data locality server. F3 sends locality information twice: once when the file is created, and again when the file is closed. The locality information sent when the file is created allows the serverless scheduler to schedule pipelined actions on the same node, since it tells the scheduler where the data will be. The locality information sent when the file is closed allows the scheduler to make scheduling decisions based on the actual amount of data that each host has.

When the serverless platform receives ❸ a new action to run, its scheduler has to choose where to run the action. If there are suitable warm containers available, it chooses one of them; otherwise, it creates a new container. When taking data locality into account, the scheduler tries to identify all files that the action is likely to access. Currently this is done by identifying strings in the action parameters that contain the mount point of the F3 file system ❹. This was sufficient for the applications that we used for our evaluation. In the future, more sophisticated methods

such as predictions based on prior action invocations can be used to identify files likely to be accessed. Additionally, a serverless orchestrator or framework such as Kubeflow [102] that knows the relationship between actions could explicitly provide information about what data an action will produce or consume.

The scheduler then sends the list of files to the data locality server ❺. The data locality server then uses the information supplied by the F3 file system drivers to identify for each file in the list what nodes have the file locally and the size of each file. It sums the amount of data available on each node, and returns this information to the scheduler ❻. The scheduler uses the information to choose a container on a node with the largest amount of data available locally ❼. If there are no suitable containers the scheduler then uses this information to tell the container platform which node the new container should be created on.

Data locality is only one of several factors that the scheduler uses to place actions. For instance, if the node with the best data locality is overloaded, then the scheduler may instead decide to run an action on a less heavily utilized node. Ideally, the serverless scheduler would provide a mechanism for letting users decide how these different pieces of information are used when making scheduling decisions, similar to the flexibility offered by the Kubernetes scheduler [111].

Reading-while-writing Usually a process consumes a file by issuing `read` system calls in a loop, stopping when `read` returns zero (*i.e.*, when the end of the file is reached). If the file is being written at the same time as it is being read, the reader would need to periodically poll for new data when `read` returns zero.

The challenge here is that the process needs to know when to stop polling because the writer has finished and closed the file. Unix pipes handle this transparently for a process: rather than returning zero, `read` blocks until more data is available as long as the write end of the pipe remains open.

F3 replicates this behavior by blocking `read` calls from returning if `read` would return zero but the file is open for writing by another process. When more data has been added to the file or the writer closes the file, F3 allows `read` to return to the caller. Since F3 spans the entire cluster, this works even if the writing process is running in a different container or a different node.

This feature makes it possible for a serverless scheduler to schedule the next stage of a pipeline before the previous stage has finished, thus improving concurrency. The same locality hints the scheduler uses to place the reader action can also be used to wait for the previous pipeline stage to create the file. Thus pipeline stages can be scheduled in parallel without code changes to either reader or writer.

If one of the pipeline steps fails, there may be subsequent stages that have already read part of the output from the failed step. If the pipeline previously ran on a single node, then it likely already has logic for dealing with this case and such logic can be reused in the serverless environment as well. For example, the application might cleanup the output from failed steps and then rerun. Since objects are written or read in their entirety, rather than incrementally as files are, additional logic may be needed for applications that currently use an object interface for storage. Detecting when a failure occurs and what recovery steps are needed (*e.g.*, failing downstream actions that are currently reading data from the failed action) is the responsibility of the serverless execution system and is out of scope for F3.

5.5 Implementation

We implemented F3, following the design described in Section 5.4. We targeted OpenWhisk [149] as the serverless platform, which we deployed on top of Kubernetes as the container orchestration platform. F3's implementation consists of four components and a series of modifications to OpenWhisk, described below. We plan to release these components publicly, as open source, available at *url-redacted*.

1. F3 file system driver The F3 file system driver is implemented using FUSE [196, 195]. We used FUSE for this prototype rather than implementing a kernel-based driver due to FUSE's relative simplicity and ease of development. We expect that any performance penalty that FUSE imposes is insignificant compared to the benefits provided by F3 (*e.g.*, fewer network transfers). In the future, a kernel version of F3 could be implemented for production uses.

The F3 FUSE driver is implemented in 2,406 lines of C and C++. An instance of the FUSE driver runs on each node, for each F3 volume mounted on that node.

2. File transfer server & client Ephemeral data written on one node and read on another node must be copied to the reader node via a network transfer. This functionality is implemented in a Go-based client and server, each of which runs on each node of the cluster. Go was chosen due to its strengths as a language for networked applications like file transfer clients and servers [181]. Additionally, Go's ability to compile into a portable executable eases the containerization and deployment of the file transfer and server [121].

The F3 FUSE driver communicates to the client via Unix domain sockets to request that a file's contents be downloaded from another node. The file transfer server and client are written in 574 lines of Go.

3. CSI driver To integrate F3 with Kubernetes, we implemented a CSI (Container Storage Interface) driver [93] to enable provisioning and attaching F3 volumes to Kubernetes pods. The CSI driver is implemented in 811 lines of Go.For example, the CSI specification website lists 83 CSI drivers with source code: of those, 74 are implemented in Go [93]. When users create an F3 volume, they must also create a volume for the networked file system that F3 will use. The F3 volume definition indicates what networked file system volume to use with the Kubernetes label [108] `f3.target-pvc: foo`, where *foo* is the name of the network file system's volume.

When the CSI driver is instructed to attach an F3 volume (*i.e.*, receives a `NodePublishVolume` CSI command), the driver checks to see if the target networked file system volume is already mounted on the node where the F3 volume is being attached. If not, the F3 CSI driver creates a pod on the target node that is attached to the target networked file system. This forces the networked file system to be mounted on the target node. F3's FUSE file system can then access the mount point. We assume that each node's local data store is mounted in advance.

4. File locality server The file locality server aggregates data from each F3 file system driver in the cluster. It is implemented in 214 lines of Go. The locality information about ephemeral data is

stored on disk in a JSON formatted file. The durability of the locality information is not critical, since the data itself is ephemeral and the serverless platform can always fall back to data-unaware scheduling.

5. OpenWhisk Modifications In addition to the new components implemented above, we had to modify the OpenWhisk serverless platform. These changes included (1) adding support for attaching action containers to storage volumes, (2) identifying what files will potentially be accessed by an action, and (3) modifying the OpenWhisk scheduler to query the data locality server and using the response when choosing a container for an action.

In total, we changed about 700 lines of OpenWhisk code, most of it in the Scala language.

5.5.1 Unmodified Applications in Serverless

One of the advantages of file based storage for serverless is that it enables running unmodified applications. To highlight this capability, we wanted to use unmodified, "off-the-shelf" applications in our evaluation of F3.

During our evaluation we tested many combinations of container images, applications, and application command line options. To simplify this process, we implemented a mechanism that enables easily running a command as an OpenWhisk action. The user runs a command with the `ow-run` utility that we created. The user experience with `ow-run` is similar to that of running a command using the command line on their local machine. For example, consider we want to run this command, normally invoked locally, as follows:

```
trimmomatic /data/0.fastq /data/0.fastq.gz
```

To run that command on OpenWhisk using our `ow-run` utility, the command line invocation would be:

```
ow-run  --container-image sunbeam:v0.0.7
        --ow-action trim
        --vol-list f3-pvc
        --mount-path-list /data
        trimmomatic /data/0.fastq /data/0.fastq.gz
```

In this example, the user needs to have already configured the resource limits and requests for the `trim` action and created the F3 volume `f3-pvc`. However, the user needs to make no modifications to `trimmomatic` itself. This allowed us to easily and efficiently test a wide range of applications and application settings.

5.6 Evaluation

Due to the growing amount of data produced by IoT devices, the rising demand for low-latency on-the-spot computing, as well as privacy and security concerns, applications and infrastructure are increasingly deployed at the Edge rather than in the hyper-scale Clouds [188]. The umbrella project for F3 focuses on the growing Edge business opportunities: thus, we designed our experimental platform and workloads to be representative of edge environments and workloads. Furthermore,

our analysis shows that thanks to its higher resource efficiency, the serverless approach could be even more attractive at the resource-constrained Edge than in the Clouds with seemingly unlimited resources.

A typical edge data center is a cluster of only 1–10 servers located either at a customer site (*e.g.*, a factory or a retail sore) or at an Internet access point (*e.g.*, 5G cell tower). The servers in a typical edge data center run standard operating system (*e.g.*, Linux), virtualization software (*e.g.*, KVM), and container orchestrators (*e.g.*, Kubernetes). Due to constraints on clusters' physical footprint, a popular architecture for Edge data centers is hyperconverged setup, where each building block (*e.g.*, a server) provides both compute and storage resources. The testbed described in the following section reflects these characteristics of edge data centers.

5.6.1 Cluster and Storage Setup

We ran our evaluation on CloudLab [56] using a cluster of nine machines connected via a 1Gbps network, with each node running CentOS Linux 7.9.2009. Each machine had two 2.60GHz, ten-core Intel CPUs with hyperthreading, 160GB of RAM, and one 480GB SATA SSD. The cluster was connected via a private 1Gbps network. Our serverless platform was OpenWhisk 1.0.0, using Kubernetes 1.19.0 as the container orchestrator.

One node was dedicated to running the `etcd` server used by Kubernetes to store cluster state; another node was the Kubernetes control node; and a third node was dedicated to running an NFS server used in evaluation. The remaining six nodes were hyper-converged Kubernetes workers that ran both evaluated workloads and storage systems—CephFS, MinIO, and F3.

In our CloudLab setup every node had only one attached disk. Since F3 requires both a local disk and a shared file system, we used LVM to split the single SSD attached to each node into two volumes. We formatted one volume with ext4 and used that as F3's local disk; we used the other volume for CephFS and MinIO.

In our evaluation we assume the case when an edge cluster already has access to durable storage: CephFS (distributed file system), MinIO (object storage), or central NFS server (NAS). F3 can be layered over these solutions (except MinIO) to provide additional performance benefits in serving ephemeral data to serverless functions. We evaluate MinIO to provide a reference point of how applications perform with a popular object storage solution rather than a file system.

CephFS Ceph [39] is a popular storage system built on the RADOS object store [203]. It aggregates storage from each node it is deployed on and exposes a single pool of storage. There are several interfaces for Ceph including CephFS, which exposes a file-system interface to applications. Ceph offers several data durability schemes, such as replication and erasure coding. We evaluated three different Ceph configurations: no replication, 3× replication, and 2-1 erasure coding (two data blocks and one erasure block).

CephFS has both kernel- and FUSE-based user-space drivers. We used the FUSE-based user-space drivers, which are typically more up to date and safer to use than their kernel counterparts. We used Ceph version 15.2.7, deployed on Kubernetes with the Rook [167] operator version 1.5.9.

MinIO MinIO [138] is a popular object store. Like Ceph, it can aggregate storage across multiple nodes and expose a single storage pool. Also like Ceph, MinIO offers several data durability modes. We chose EC-3, which was the default for our sized cluster (six nodes, one disk per node). This mode splits data into three data chunks with three coding chunks. We used MinIO release 2022-09-07T22-25-02Z, deployed on Kubernetes with version 4.5.0 of the MinIO operator.

We used `s3fs` [170] to access MinIO's object API and provide a file-based interface over MinIO. This is representative of the current state of storage for serverless: if a user wishes to run an application on a serverless platform but the application requires a file based storage interface, they would need to use a tool like `s3fs` to access an object store.

NFS NFS [176] is a well-established file system protocol. Although hyper-converged configurations such as those used by Ceph and MinIO are common, architectures that use standalone NAS storage appliances are still used. NFS is mature, and easier to deploy and configure compared to more sophisticated, distributed or networked file systems like CephFS. We used NFS on a standalone, dedicated server in our cluster—to represent this alternate architecture. We used the standard NFS server included with the Linux kernel to export a local disk formatted with ext4. The NFS version was 4.1, which was the default version available on our operating system (CentOS Linux 7.9.2009).

F3 In most experiments we evaluated F3 using CephFS with no replication as our networked file system. The local disks used as a per-node local data store were formatted with ext4, which is a commonly recommended file system and the default for many operating systems [166]. Although we mainly used CephFS as the networked file system for our evaluations, F3 is capable of stacking on top of any underlying networked file system that supports extended attributes. To test this, we verified that F3 also works on a recent NFSv4.2 server with extended attributes support.

We measured the impact of using different networked file systems (CephFS with no replication, $3\times$ replication, 2-1 erasure coding, and NFS) with F3. We found that the choice of underlying file system had little to no impact on the performance of ephemeral data operations: performance in each case was within 3.1% of each other for reads where the data was not available locally, and less than 0.03% and hence statistically indistinguishable. of each other in all other cases. This is because F3 is designed to avoid the networked file system for ephemeral operations. We used unreplicated CephFS as our networked file system throughout our evaluation: any reference of "F3" in the evaluation means "F3, layered on top of unreplicated CephFS."

Since the focus of this evaluation was on F3's features for ephemeral data, all data in our evaluation was marked as being ephemeral. We leave to future work evaluating the performance of mixed ephemeral and non-ephemeral data operations, as well as how to automatically identify whether data is ephemeral or non-ephemeral.

Disk vs. network speed ratios When selecting the server for a file system that accesses disks over the network, disk speeds and network speeds should be on par with each other so that neither dominates as the primary bottleneck. We chose network and disk speeds that were representative of real-world ratios. Each of our servers had only a single disk available for the storage systems

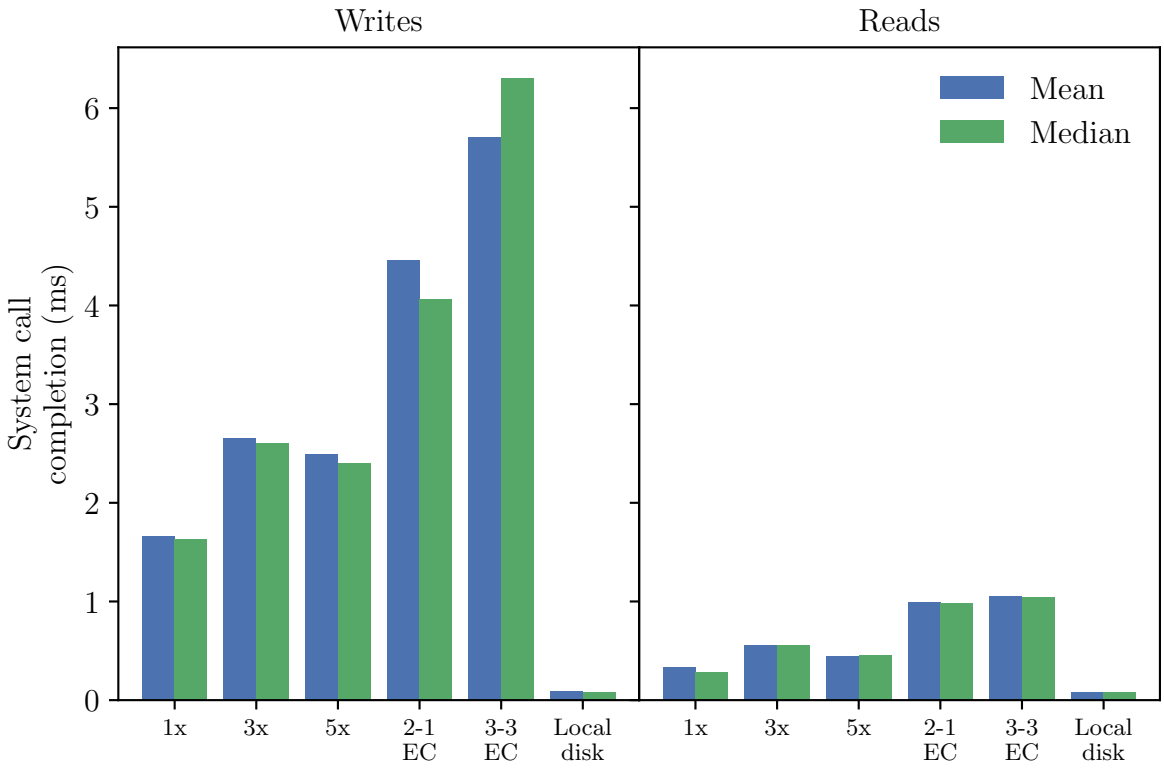


Figure 5.3: Mean and median system call latencies for different configurations of storage system. 1×, 3×, and 5× refer to the degree of replication; 2-1 and 3-3 EC refer to the erasure coding configuration (2 data and 1 coding chunk, and 3 data and 3 coding chunks, respectively).

under evaluation. We measured the disk speed to be 200MB/s, giving a disk to network throughput ratio of approximately 1.6 with the 1Gbps network. Although 1Gbps is slow compared to the networks found in many modern data centers, our disk to network throughput ratio falls within the range typical of real world edge deployments [204]. If we instead had ten disks with a combined throughput of 2000MB/s and a 10Gbps network, the ratio would remain the same.

5.6.2 Data Transfer Micro-Benchmarks

We evaluated the performance of data exchange and the impact of F3's data exchange optimizations. We first show the performance impact of different replication and erasure coding levels, compared to a baseline of accessing a local disk. This is the motivation behind F3's use of a local disk for ephemeral data.

We then show the impact of data locality based scheduling, and avoiding the overhead of transferring data across the network. Next, we show the combined impact of F3's data locality based scheduling and F3's use of local disk for ephemeral data. Finally, we show the impact of F3's optimizations for reading-while-writing.

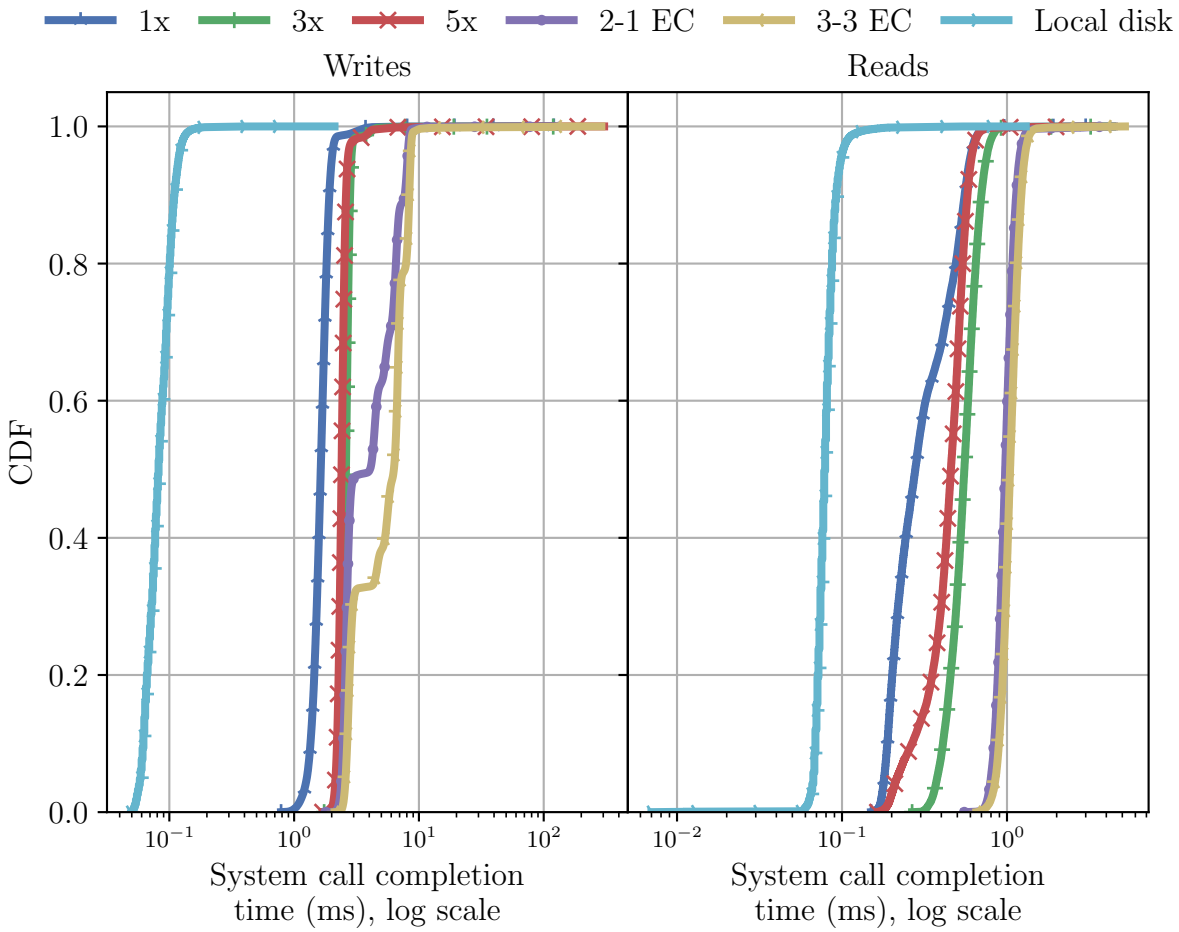


Figure 5.4: CDFs of `read` and `write` system call latencies, for different storage configurations. We used log scale for the x-axis because the system calls exhibited long tails at higher replication degrees.

Impact of replication & local disk storage We evaluated the impact of replication and erasure coding on the latency of `read` and `write` system calls. We ran several experiments to time 100,000 reads and 100,000 writes on CephFS volumes with varying replication and erasure coding options, and compared with the same workload on an ext4 file system on a local disk. Since CephFS uses a FUSE driver, we used a passthrough FUSE file system to access the ext4 file system. This ensured that all `read` and `write` system calls went through a FUSE layer for a more fair comparison. System call times were measured with `strace`, and were generated with `dd` with `bs=4K` and `[o|i]flag=direct`.

Figure 5.3 shows the mean and median system call latency across multiple storage configurations. The distribution of latencies exhibited a long tail, as can be seen in Figure 5.4 (note the log scale). This is expected, as there are multiple sources of variability in the storage and networking stacks, and have been observed before [92, 78, 32, 139]. As the degree of replication increases, we see the tail grow longer, which also makes sense as the number of sources of variability increases.

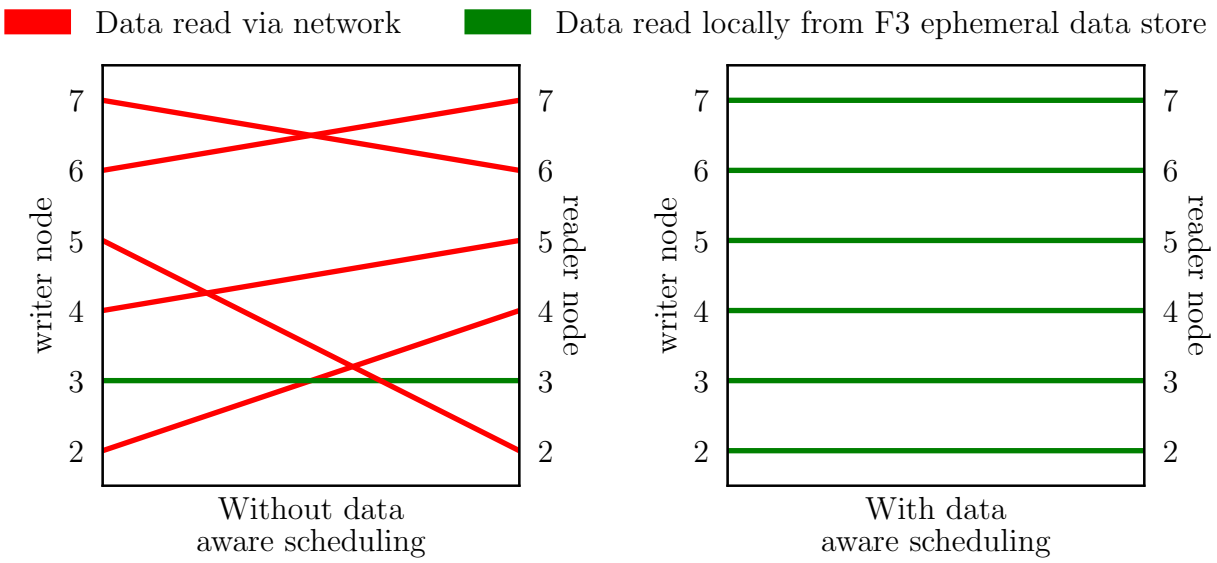


Figure 5.5: Impact of data aware scheduling. Each line connects a writer with its corresponding reader, with the numbers along each side showing what node in the cluster the writer or reader ran on. Red lines indicate that the reader needed to transfer its file from the writer node via a network transfer. Green lines indicate that the reader and writer ran on the same node and the file was read from F3's local ephemeral data store, with no network transfer was needed.

As expected, the local disk performed significantly better than CephFS, especially when writing: 0.1ms vs. 2.2ms for $1\times$ replication). We also see that as the replication degree increased, generally so did system latency. The exception was that for reads, $3\times$ and $5\times$ replication perform about the same or slightly better than $1\times$ replication.

Impact of data locality considerations during action scheduling To demonstrate the impact of locality aware data scheduling, we wrote and then read six ephemeral files. Writers were run manually on each node, one per node, with each writing a unique 400MB file. For each writer, a corresponding reader was run in an OpenWhisk action that read the entire 400MB file. When the reader and writer both run on the same node, the reader reads its file from F3's local disk. However, when the reader and writer each run on separate nodes, the data must be transferred from the writer node to the reader node over the network.

The left-hand side of Figure 5.5 depicts the case where OpenWhisk's default scheduling is used. Here, the readers are assigned to nodes without regard to where the input file they need to read is located; we see that only a single reader (green line) ended up running on the same node as its corresponding writer. The red lines depict instances where the reader ran on a different node from its writer, necessitating a 400MB network transfer to copy the data from the writer node to the reader node. In total, using the default OpenWhisk scheduler resulted in $5 * 400 = 2000$MB of data being transferred across the cluster network.

The right side of Figure 5.5 shows the impact of our modified OpenWhisk scheduler that uti-

lizes F3's data locality hints. All six readers were scheduled on the same node as the corresponding writer, and hence no data was transferred over the network.

Impact of replication, local disk storage, and data locality We used `fio` [63] to measure sequential and random read and write performance of the storage systems. `fio` ran in a pod (container) via a serverless action. Write and read workloads were generated by separate instances of `fio` running in separate pods. The data written by `fio` was marked as ephemeral, and the reader instance ran after the writer instance finished. We measured read performance where the reader pod ran on the same node as the writer pod, as well as when the reader pod ran on a different node. This demonstrates the difference in performance that data locality can have on an I/O workload. We disabled F3's data locality based scheduling to be able to control whether the reader ran on the same or different node as the writer. We used a large (200GB) dataset to mitigate the impact of caching.

Figure 5.6 shows the bandwidth reported by `fio`, in MB/s, and the mean latencies, in milliseconds. We ran `fio` in each configuration three times. Error bars show that variance was small, less than 5% of the mean, with one exception: F3 random reads on the same node, where the variation was 7%.

As expected, F3 had the highest read and write performance when the reader was on the same node as the writer. F3's write bandwidth ranged from 1.40× to 6.46× faster than other storage systems; read bandwidth ranged from 1.84× to 2.30× faster. Latency ranged from 1.40× to 2.64× lower when writing and from 1.84× to 2.73× lower when reading. These performance improvements were due to F3's use of local storage. By using local storage, F3 is not limited by the cluster's network capacity as other storage systems are.

F3's read performance when readers and writers ran on different nodes was similar to NFS. In both cases, the data had to be transferred over the network.

Each of the networked file systems was limited by the cluster's 1Gbps (125MB/s) network. The one exception was writing in the unreplicated configuration of Ceph. This was expected because Ceph breaks files into blocks that are then distributed across each of the storage nodes in the Ceph cluster. Because we were using a hyperconverged architecture, the Ceph storage nodes were the same nodes that run user workloads, including our instance of `fio`. Since we had six nodes in our cluster, we expect then that $\frac{1}{6}$ of the data written by `fio` resided on the node running the `fio` program, and as a result was not limited by the cluster's network.

Impact of reading-while-writing Passing data from one stage of a data processing pipeline to the next is a common pattern. A straightforward implementation is to run the pipeline stages sequentially, where each stage produces an output file that the next stage reads as input. A disadvantage of this approach, however, is that it provides no parallelism between pipeline stages.

Another possible implementation is to run pipeline stages concurrently, streaming the data between stages (*e.g.*, using UNIX pipes to connect them). The added parallelism of streaming can result in lower end-to-end processing times. A limitation of using UNIX pipes, however, is that the stages must all be run on the same node, which is not always convenient.

In this section, we use a third approach where a stage in the pipeline reads input from a file in

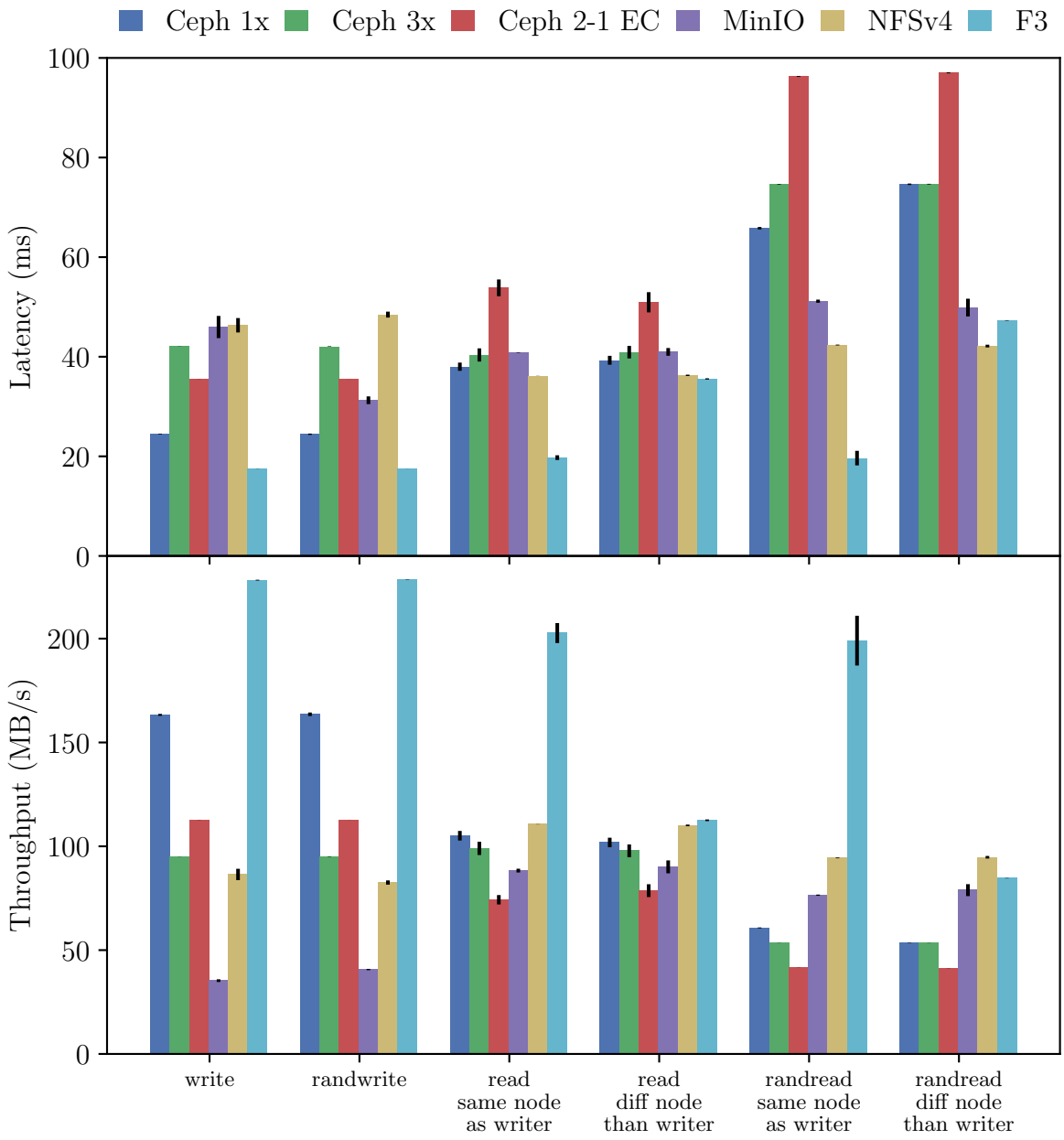


Figure 5.6: Mean latencies and bandwidths of Ceph, NFS, MinIO, and F3. "Ceph 1×" and "Ceph 3×" are configured with 1× and 3× replication, respectively. "Ceph 2-1 EC" uses erasure coding (data split into two data chunks and one coding chunk). F3 was layered on top of an unreplicated CephFS volume.

a shared file system while the previous stage writes the file. We show below how we solved the problem of the reader reaching the end of file before the writer has finished writing all data.

We ran experiments in a serverless environment using CephFS, NFS, and F3 as the shared file system—first with the reader on the same node as the writer, then with the reader on a different node. We used a smaller data set (400MB) than the server's RAM (160GB), so the results reflect the ability of the storage systems to leverage the kernel's page cache rather than being disk bound.

To solve the early EOF problem, we made a few changes to our pipeline stages. First, we modified the writer stage to create an empty file, `/var/data/f.done`, on completion. Next, we split the reader into two parts. The first part was a script that read from `/var/data/f` and wrote to a FIFO pipe, `/tmp/f.pipe`. Whenever the script reached EOF on input, it checked for the existence of the `/var/data/f.done` file, and if not found, slept one second (same duration as `tail -f`), then returned to the top of the loop and continued reading. The second part was the actual reader program (*e.g.*, `grep` or `cat`), except that instead of reading from `/var/data/f`, it read from `/tmp/f.pipe`. Both parts of the reader ran in the same action. This implementation enabled us to run a reading-while-writing workflow in a serverless context.

Because F3 has special support for handling EOF in the reading-while-writing access pattern, it did not require any of the additional implementation: the writer action simply wrote to `/var/data/f` and the reader action simply read from `/var/data/f`.

Note that because CephFS was not designed for this usage pattern, it does not handle the reading-while-writing case efficiently when reader and writer run on different nodes. In this pattern, it falls back to unbuffered reading and writing [40].

MinIO is not capable of reading from an object as it is being written to, so it is omitted from these experiments. This example further highlights the limitation of object-based interfaces.

Figure 5.7 shows the difference in same-node-reader vs. different-node-reader performance. As expected, for all storage systems, read performance is worse when the reader is on a separate node from the writer. However, Ceph's write performance is also lower when the reader is on a separate node. This is because when both reader and writer are on the same node, Ceph can do buffered reading and writing, as only a single client is accessing the file. When the reader and writer are on separate Ceph nodes, however, there are now multiple clients accessing the same file and Ceph falls back to its slower, unbuffered file accesses (plus the additional overhead of network transfers).

5.6.3 Case Study: Bioinformatics Pipeline

We developed a bioinformatics case study in collaboration with an industry partner specializing in large scale processing genetic sequence data. The advent of new genetic-sequencing technologies (*e.g.*, nanopore) has made sequencing more portable, affordable, and accessible. Sequencing can now be done anywhere from hospitals to sea-bound ships and is being used for an increasing number of applications [50].

Sequencing typically produces a large amount of data that is then processed using a series of steps run in a pipeline. The pipeline typically begins by cleaning and filtering the data, for example removing artifacts created as a byproduct of the sequencing technology. After cleaning, the sequence data is then analyzed, for example to identify the species present in a sample.

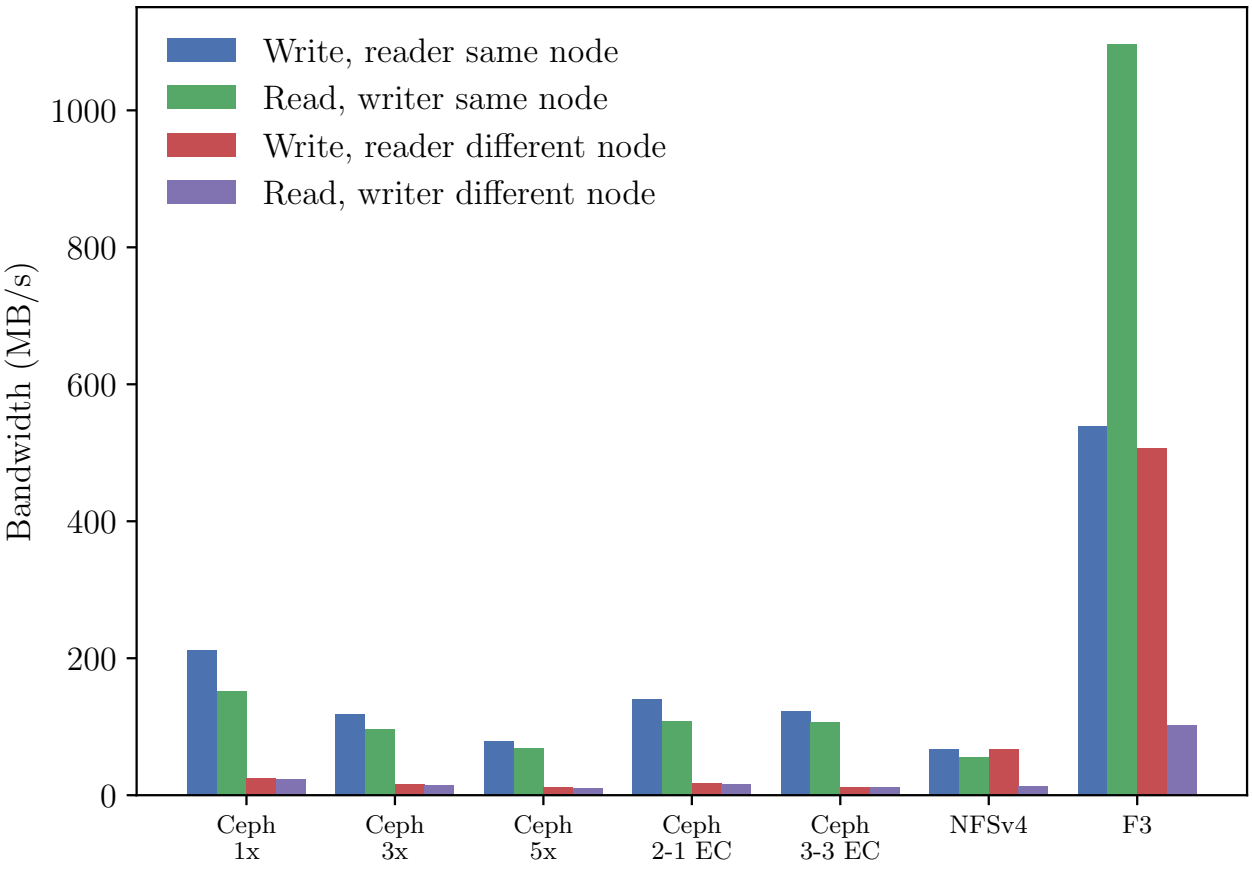


Figure 5.7: Comparison of read-while-write performance, when readers and writers are on the same or different nodes. For CephFS, having the reader and writer on different nodes significantly degrades both read and write performance. MinIO is absent from this experiment due to its inability to read and write data concurrently. F3 was layered on top of an unreplicated CephFS volume.

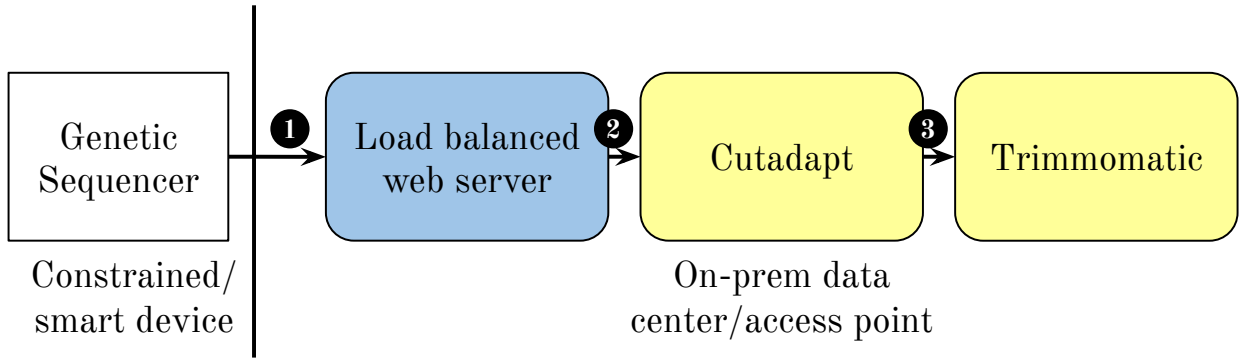


Figure 5.8: Bioinformatics use case architecture

Running all or part of the analysis pipeline at the edge where the sequence data is generated can save significant time and cost associated with moving a large amount of data to the cloud. It is not always possible or desirable to run the entire processing pipeline at the edge, for instance, when the analysis requires more computing power than is available in the edge data center, or when the analysis output is required in the cloud for other reasons (*e.g.*, archival). But running at least the cleaning portion of the pipeline at the edge can still significantly reduce the amount of data uploaded to the cloud.

Because various stages in the pipeline have different resource requirements, running the pipeline in a serverless environment where each stage is run as a separate action provides better resource utilization.

Analysis pipelines are usually built using existing tools developed by other bioinformatics researchers. These tools usually assume a file interface for their inputs and outputs.

We implemented the cleaning stage of a genetic-sequencing pipeline using two commonly used tools: Cutadapt [129] and Trimmomatic [29]. Cutadapt identifies and removes portions of sequences that were added to support the sequencing process and are unrelated to the data being analyzed. Trimmomatic removes sequences that fail to meet a given quality metric. Usually, Cutadapt is run first. Its output becomes the input for Trimmomatic.

Figure 5.8 describes our implementation. We uploaded a 926MB file of genetic-sequence data to a load-balanced web server ❶, which wrote the file to a data store as ephemeral data. Because the web server uses a load-balancer to distribute requests among nodes, the server that receives and stores the sequence data can be any of the worker nodes in the cluster.

Once the receiving node had saved the file, it ran Cutadapt ❷ and Trimmomatic ❸ as OpenWhisk actions. We ran the pipeline in two modes: sequential and pipelined. In the sequential mode, Trimmomatic was started after the completion of Cutadapt. In pipelined mode, Trimmomatic was run at the same time as Cutadapt, operating on Cutadapt's output as it was being written. Cutadapt's output was 926MB and Trimmomatic's output was 126MB. Together, the two applications reduced the input data size by 7.3×.

Additionally, running the tools in separate actions provided better resource efficiency. The memory requirement of Trimmomatic is 1024MB, while that of Cutadapt is only 32MB. If both steps ran in the same context, then the system would have had to reserve the larger memory requirement for the duration of both pipeline stages. By scheduling them as separate actions, however, the larger memory reservation was needed only for the duration of the Trimmomatic stage. Running in separate actions is enabled by providing access to shared, file based storage.

Figure 5.9 shows the end-to-end runtimes of the pipeline. The pipeline ran fastest on F3, ranging from 8% to 34% faster than on other storage systems for the sequential mode, and 9% to 47% faster for the pipelined mode. Note that for the pipelined mode, MinIO results are not shown because it is incapable of being run in this mode (simultaneous reading and writing). The pipeline ran slowest on MinIO, not surprising since the pipeline writes a large amount of data during the Cutadapt stage and MinIO has the worst write performance of all evaluated storage systems.

NFS performed similarly to F3, running only 8% slower. There are two factors that contribute to this: the first is that the size of the data used in the experiment is small. This means that the time spent on I/O compared to the overall runtime is relatively small, and so improvements to that I/O

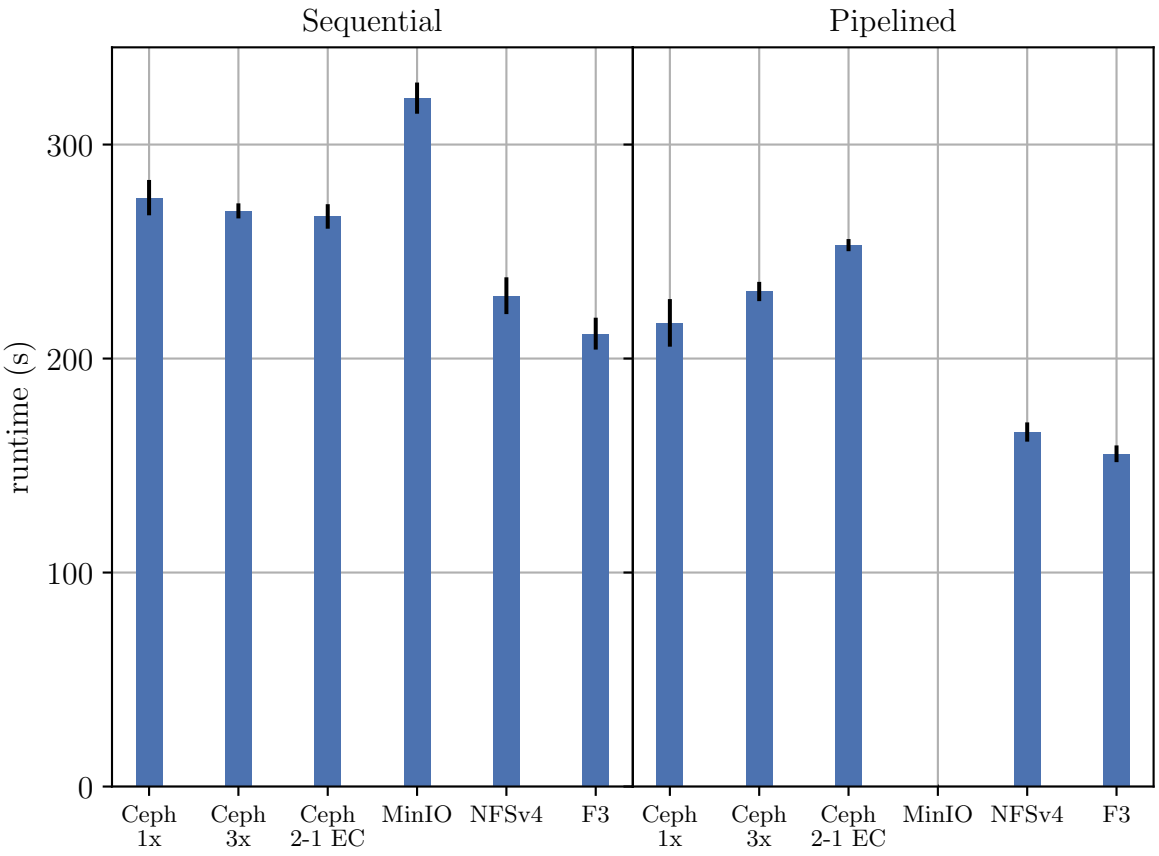


Figure 5.9: Runtime of Cutadapt + Trimmomatic pipeline. F3 was layered on top of an unreplicated CephFS volume.

time have a small impact on the larger runtime.

Second, the experiment was conducted in what are close to "ideal" conditions for NFS: only a single client and no other network traffic. This allowed the data transfers that take place during the experiment to utilize the entire network capacity. As a quick test, we used `iperf` to generate network traffic and re-ran the experiment for NFS: at 50% network utilization F3 performs 16% faster than NFS, 25% better at 75% utilization, and 59% better at 90% utilization. All networked file systems will be subject to performance variation based on the overall network utilization. F3, by using local disks and data locality scheduling, avoids this problem—performing relatively better and better as the network gets more congested.

5.7 Conclusion

Serverless platforms have been steadily growing in popularity. Although so far they have been limited to relatively simple web-based tasks, users and researchers are beginning to appreciate the potential of serverless platforms' on-demand computing capabilities. As serverless platforms

make the shift to being a platform for any generic task, two significant problems remain: access to storage and data transfer.

Some advanced and existing applications require access to file-based storage. To support these applications, serverless platforms need to allow attaching to file-based storage systems. However, existing storage systems were not designed with serverless applications in mind and lack key features that would accelerate the kind of data transfers commonly found in serverless environments: (1) support for ephemeral data, (2) data locality-aware action scheduling, and (3) support for efficient simultaneous data access (*i.e.*, reading files as they are written).

In this chapter, we presented F3, a file system that layers on top of existing storage systems to provide these three key data-transfer features. We additionally described modifications to an open source serverless platform, OpenWhisk, to enable attachment of file-based storage and take advantage of data locality hints provided by F3 when scheduling actions. We evaluated F3 and showed that it is capable of 2.0–6.5× faster write bandwidths and 1.8–2.3× better read bandwidths compared to existing storage systems. Combined with our modifications to OpenWhisk, we demonstrated that F3's data locality hints totally eliminating network traffic caused by data transfers, by enabling OpenWhisk to schedule actions on the same node as the action's data.

Chapter 6

Balancing Costs and Durability for Serverless Data

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 chapter, 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$\times$ while never exceeding baseline costs.

6.1 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% [174, 175, 173, 143]. 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 [174, 135, 143, 208] and data sizes are increasing [198, 61], 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 5.2).

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 [205, 160, 3, 20].

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, 86, 31, 51, 64], meaning that the data created by these actions can be re-created by simply re-running the action. This provides a standard response

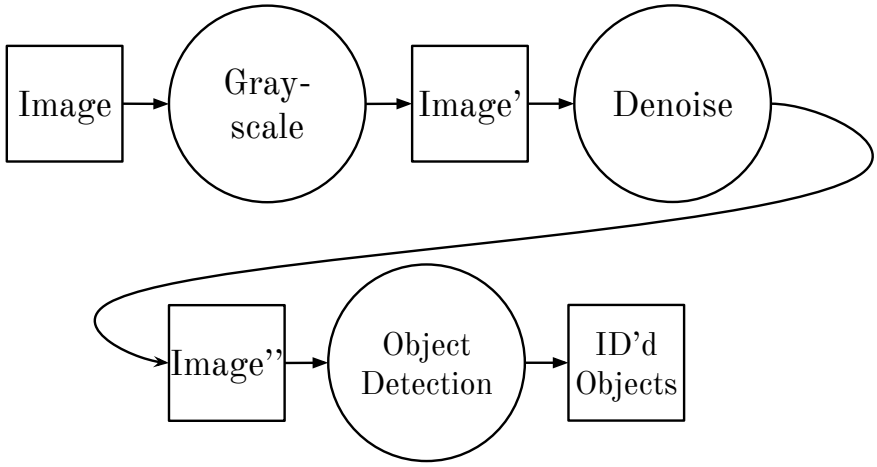


Figure 6.1: Example serverless application

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 [64, 27]. 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 6.1. 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"*.

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.

6.2 Background & Motivation

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 $1 - (AFR * MTTR)^{\textit{failure tolerance}}$ [207, 205], 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 [122, 171].

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.

Ephemeral serverless data is often short lived, with lifetimes on the order of seconds to minutes [99, 97]. 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.

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 6.2 shows the storage plus re-compute costs of a DAG discussed in further detail in Section 6.5.2. We compare the cost of two storage classes, one with high durability and the other with increasingly high failure rate. We

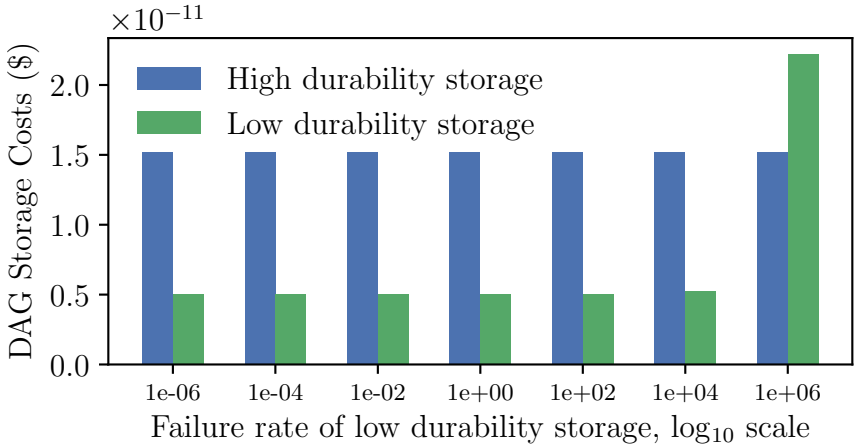


Figure 6.2: Storage and re-execution costs for high and low-durability storage. The failure rate for high-durability storage is zero. The failure rate of the low-durability storage varies, increasing along the X-axis. More details about the experiment settings in Section 6.2.

price the low-durability storage at 3× 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.

Handling lost data Serverless actions are required in most cases to be idempotent [125], 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 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.

6.2.1 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 6.1 lists some criteria that help

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

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 [134] or a machine-learning preprocessing pipeline [189, 71]. 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 [162], or applications that have strict requirements on response time (*e.g.*, applications that handle user requests [26, 19]).

6.3 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.

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 $c_{\mathrm{d}_{n_i}}$ \$ per byte-second and a probability of failure, explained below.
- i-specific cost associated with each set of storage class choices, calculated as $c_{storage} = \sum_{x \in X} x_l \times x_s \times c_{\mathrm{d}_{n_i}}$.
- a base cost, assuming no failures, denoted by $C(n_i, \mathrm{d})$ and calculated as $c_{compute} + c_{storage}$.
- a failure probability associated with a set of storage class choices $\mathrm{d} \in \mathcal{D}$, 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 $\mathbb{E}(cost(n_i, \mathrm{d}_{n_i}))$.

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.

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:

$$\begin{aligned} &\mathbb{E}(cost(n_0, \mathrm{d}_{n_0})) \\ &= \mathbb{P}_s \times C(n_0, \mathrm{d}_{n_0}) + \mathbb{P}_f \times \Big\{ C(n_0, \mathrm{d}_{n_0}) + \mathbb{E}(cost(n_0, \mathrm{d}_{n_0})) \Big\} \end{aligned} \tag{6.1}$$

Solving the above equation (6.1) leads to

$$\mathbb{E}(cost(n_0, \mathrm{d}_{n_0})) = \frac{C(n_0, \mathrm{d}_{n_0})}{1 - \mathbb{P}_f} = \frac{C(n_0, \mathrm{d}_{n_0})}{1 - p_{\mathrm{d}_0}} \tag{6.2}$$

If we assume that each node keeps its data until the end of the DAG's execution, then Equation 6.2 can also be used for calculating the expected cost of a generic node.

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 6.2, we have

$$\begin{aligned}\mathbb{E}(cost(n, \mathrm{d}_n)) = \mathbb{P}_s \times C(n, \mathrm{d}_n) \\ + \mathbb{P}_f \times \Big\{ C(n, \mathrm{d}_n) + \mathbb{E}(cost(n, \mathrm{d}_n)) \\ + \mathbb{E}\big(cost(\text{Pa}(n), \mathrm{d}_{\text{Pa}^{(1)}(n)})\big) \Big\}\end{aligned} \tag{6.3}$$

Solving the above leads to

$$\begin{aligned}&\mathbb{E}(cost(n, \mathrm{d}_n) \\ &= \frac{\Big\{ C(n, \mathrm{d}_n) + \mathbb{P}_f \mathbb{E}\big(cost(\text{Pa}(n), \mathrm{d}_{\text{Pa}^{(1)}(n)})\big) \Big\}}{1 - \mathbb{P}_f} \\ &= \frac{\Big\{ C(n, \mathrm{d}_n) + p_{\mathrm{d}_n} \mathbb{E}\big(cost(\text{Pa}(n), \mathrm{d}_{\text{Pa}^{(1)}(n)})\big) \Big\}}{1 - p_{\mathrm{d}_n}}\end{aligned} \tag{6.4}$$

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

$$\mathbb{E}\big(cost(\text{Pa}(n), \mathrm{d}_{\text{Pa}^{(1)}(n)})\big) = \mathbb{E}\Big(\sum_{j \in \text{Pa}_j(n)} \text{cost}(j, \mathrm{d}_j) \Big). \tag{6.5}$$

This establishes a *recursion* between $\mathbb{E}(cost(n, \mathrm{d}_n))$ and $\mathbb{E}(cost(\text{Pa}(n), \mathrm{d}_{\text{Pa}(n)}))$. Using the recursion in Equation 6.4 and the expected cost of the root node (Equation 6.1), we can calculate $\mathbb{E}(cost(n, \mathrm{d}_n))$ for any particular node n.

Our implementation of SDCM allows a user to choose between using Equations 6.2 and 6.4.

Minimizing DAG execution costs The expected system-level cost is

$$\text{System-cost}(\mathrm{d}_n, n \in \mathcal{N}) = \sum_{n \in \mathcal{N}} \mathbb{E}(cost(n, \mathrm{d}_n)). \tag{6.6}$$

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

$$\min_{\mathrm{d}_n \in \mathcal{D}, n \in \mathcal{N}} \text{System-cost}(\mathrm{d}_n, n \in \mathcal{N}). \tag{6.7}$$

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 6.2) this is $\mathcal{O}(cn)$, and for the correlated case (Equation 6.4) 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

$$\min_{\mathrm{d}_{n'} \in \mathcal{D}} \text{System-cost}(\mathrm{d}_n, n \in \mathcal{N}) \tag{6.8}$$

by fixing $\mathrm{d}_n, n \in \mathcal{N} - \{n'\}$, and iterating over all $n' \in \mathcal{N}$. To solve Equation 6.8, 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 6.8 is equivalent to minimizing the cost of a single successful execution of n', namely

$$\min_{\mathrm{d}_{n'} \in \mathcal{D}} \mathbb{E}(cost(n', \mathrm{d}_{n'})) \tag{6.9}$$

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.

6.3.1 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 [134] 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.

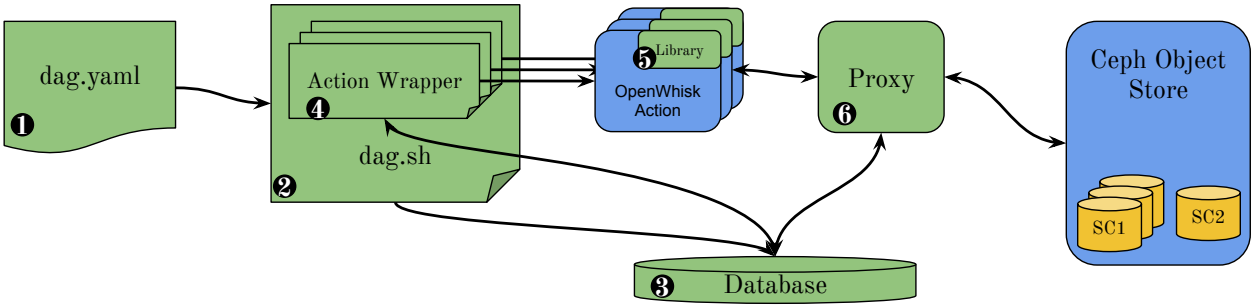


Figure 6.3: Design of execution system

6.4 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 4.2 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 4.2. These components include mitmproxy [140] 6, MongoDB [141] 3, and additional scripts. In total, our execution system consists of around 400 lines of Go and 1,535 lines of Python.

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 1, 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 2. 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 3.

OpenWhisk actions are not invoked directly, but are instead run via a Python runner 4. This runner is responsible for logging metrics such as the runtime of each action, as well as for passing

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 6.2: Hypothetical storage classes

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 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 [169, 38].

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 6.9 described in Section 6.3. The script saves these selections in the database ❸.

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.

6.4.1 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 6.2 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.

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.

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 [88, 76]. 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 [75, 57]. We instead use a simulator developed by Greenan [75] 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.

6.5 Evaluation

In this section, we demonstrate using SDCM to make storage placement decisions and predict DAG execution costs. For all experiments, we use our execution system that handles automatically re-creating lost data.

In Section 6.5.1 we evaluate SDCM's ability to make accurate predictions of DAG execution costs. In Section 6.5.2 we explore how changing model and environmental parameters impacts SDCM's decisions. In Section 6.5.3 we provide an analysis of the impact that re-executing actions has on overall DAG runtimes. Finally, in Section 6.5.4, we demonstrate using SDCM and an execution system for running two real world applications.

6.5.1 Model Accuracy

We evaluated SDCM by executing the DAG depicted in Figure 6.4. Each function in the DAG simply makes a copy of its input data, then exits. We evaluate several factors, including DAG

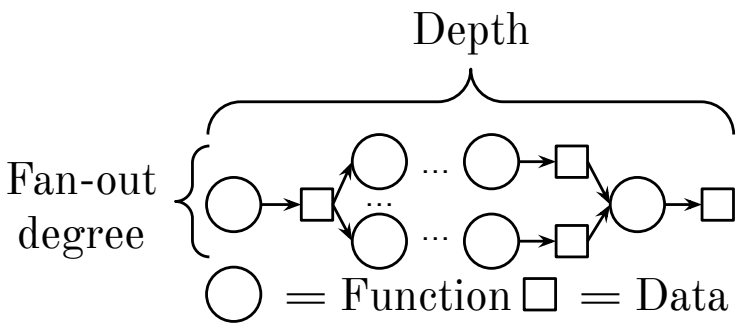


Figure 6.4: DAG Used for Accuracy Evaluation

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 6.4) 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 6.5. 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.

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 6.4.1, $\$1.06 \times 10^{-5}$ per GB-hour, as the

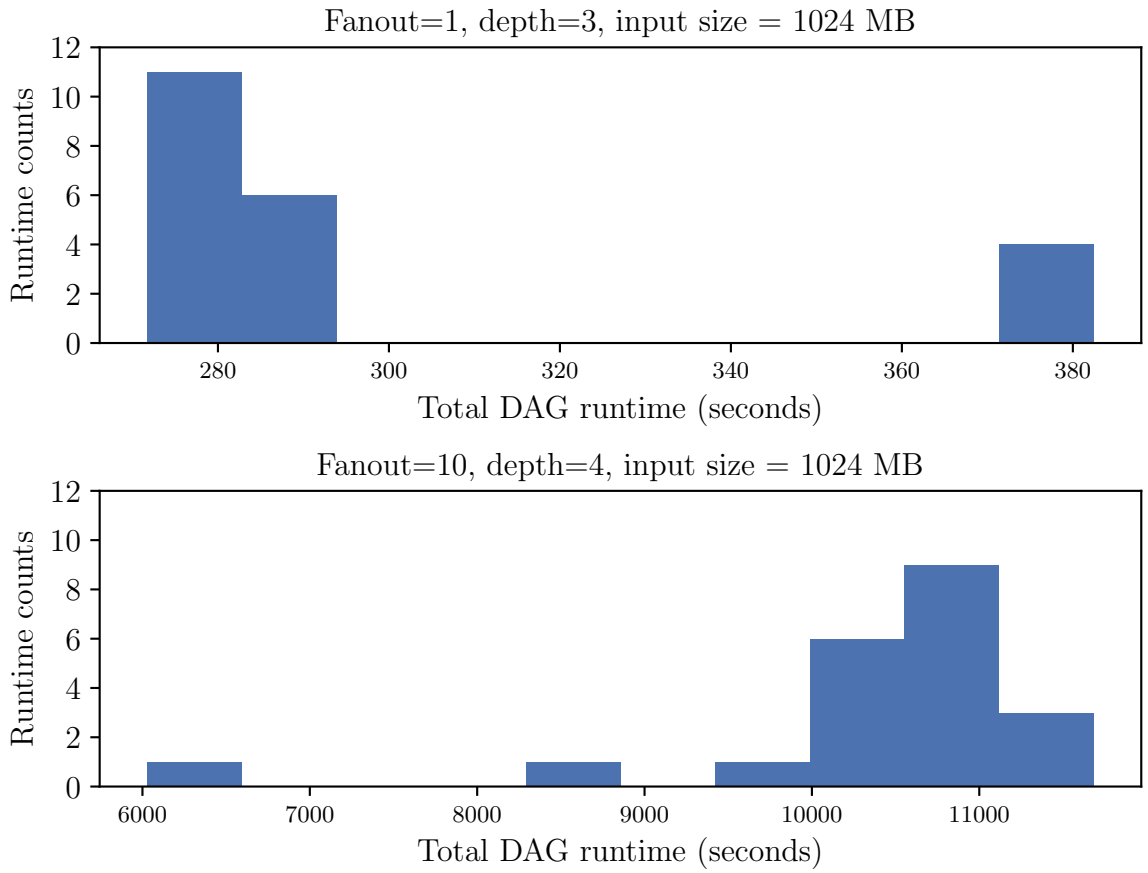


Figure 6.5: Histograms of DAG runtimes for the two DAGs with highest runtime variance. The bimodal distribution of runtimes is a result of some DAG runs encountering errors and other runs encountering no errors.

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 6.6 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).

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 $\pm 5\%$ of actual DAG execution costs. This is better than similar studies, which predicted DAG execution properties to within 15% [125, 126].

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

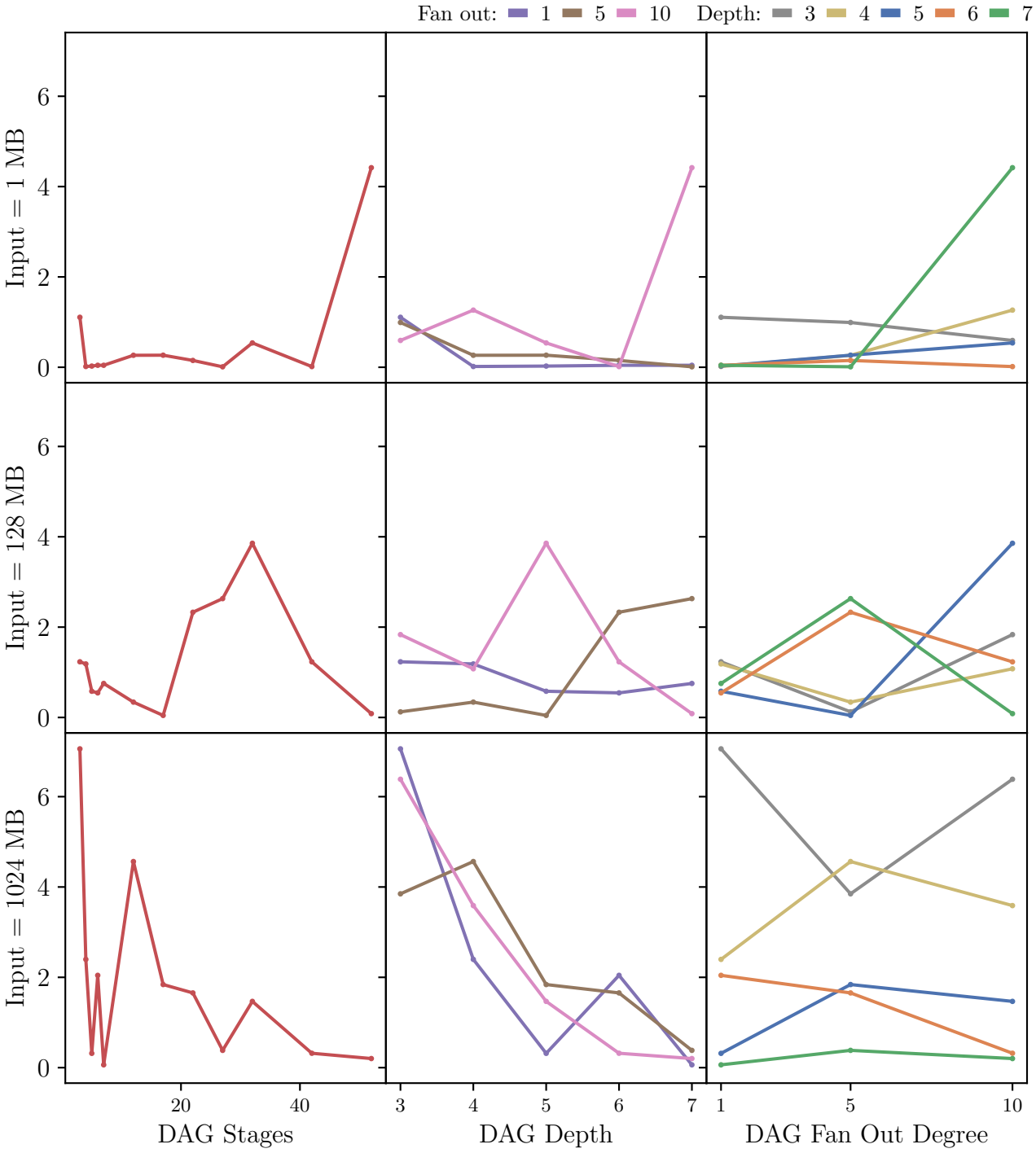


Figure 6.6: Accuracy of SDCM for three different input sizes.

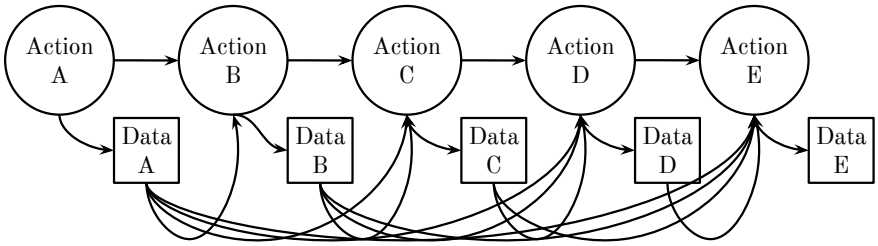


Figure 6.7: DAG used for model parameters exploration. The data produced by each step of the DAG was used by each subsequent step.

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).

6.5.2 Model Parameters

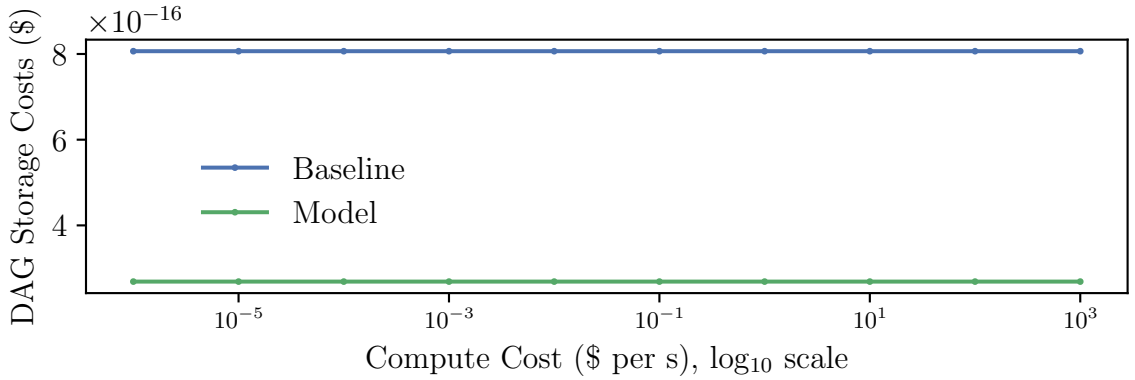
We now describe the impact that different parameters have on SDCM's storage class decisions. We use a simple five-stage DAG for exploring these parameters, shown in Figure 6.7. Each step produced 1MB of output, and each step's output was used by each subsequent step (so the lifetime of data A is highest, 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 6.4.1.

Compute cost We first look at how the cost of compute impacts storage class placement decisions. Figure 6.8a 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.

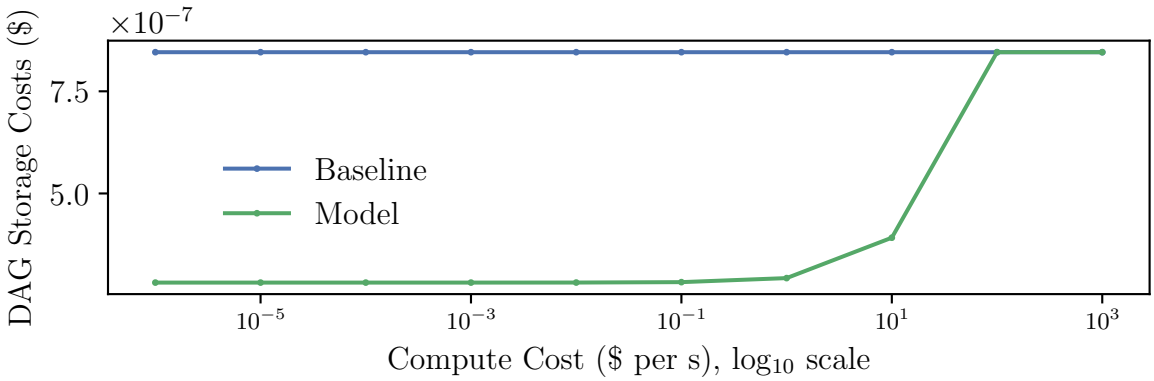
"Baseline" is the storage component of the total cost to execute the DAG if the most-durable storage class is used for all the DAG's data.

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}.

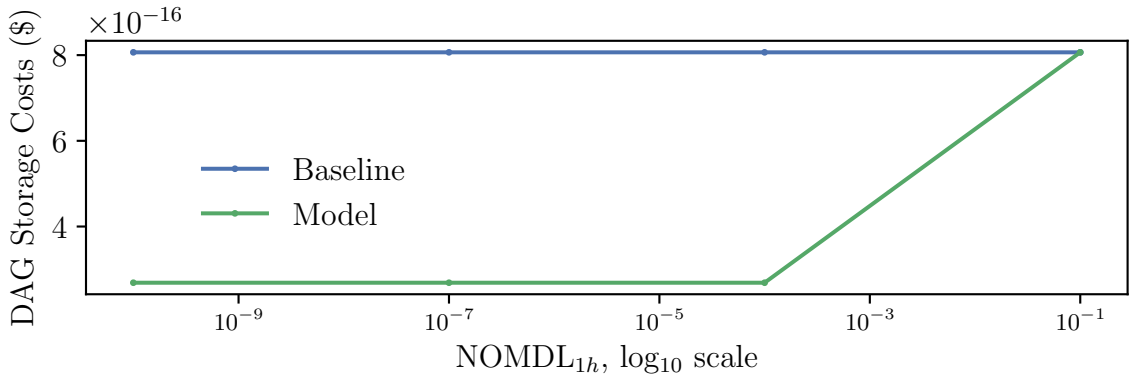
Figure 6.8b shows what happens if we instead have 1TB of data produced at each stage, the



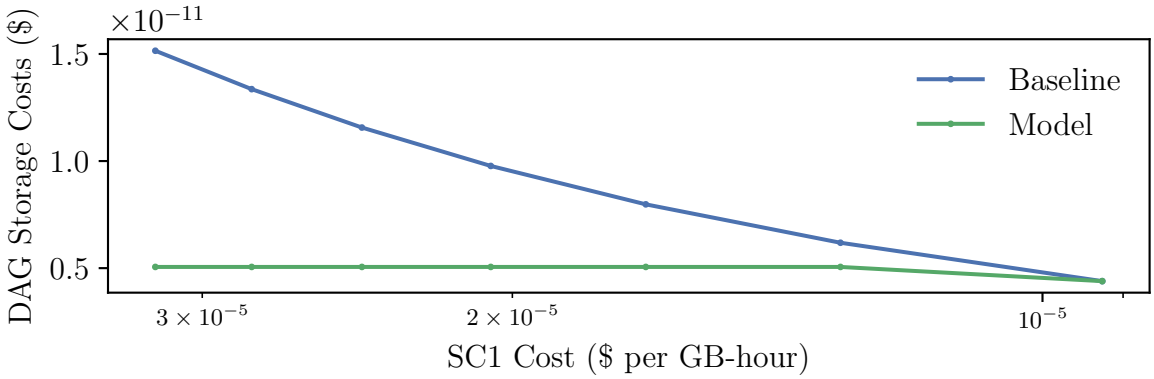
(a) Storage costs during DAG execution vs cost of compute. Note the log scale on the X-axis.



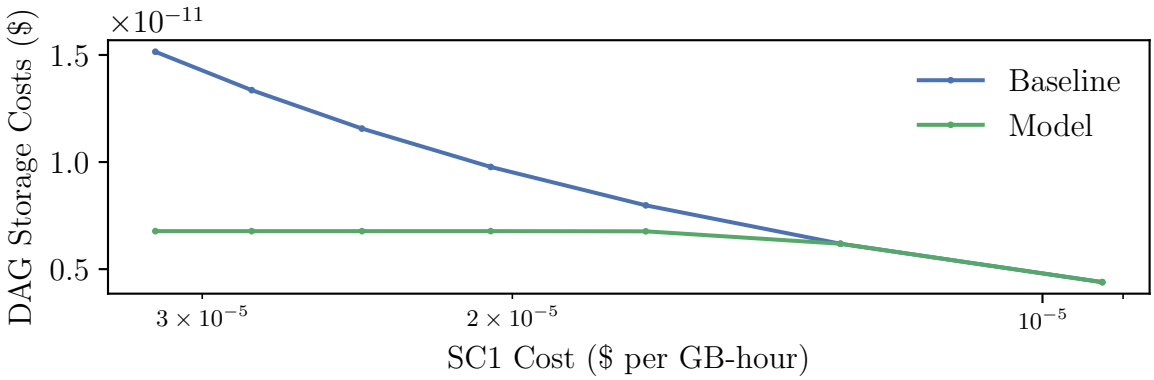
(b) Storage costs during DAG execution vs. cost of compute, if intermediate data is larger, longer lived, and storage is less reliable. Note the log scale on the X-axis.



(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.

Figure 6.8: Impact of different parameters on SDCM storage class choice.

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

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 6.8c 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.

Storage price Finally, we evaluate what it would take for a more-durable storage class to be cost-competitive with lower-durable storage. Figure 6.8d shows SDCM and baseline costs as we

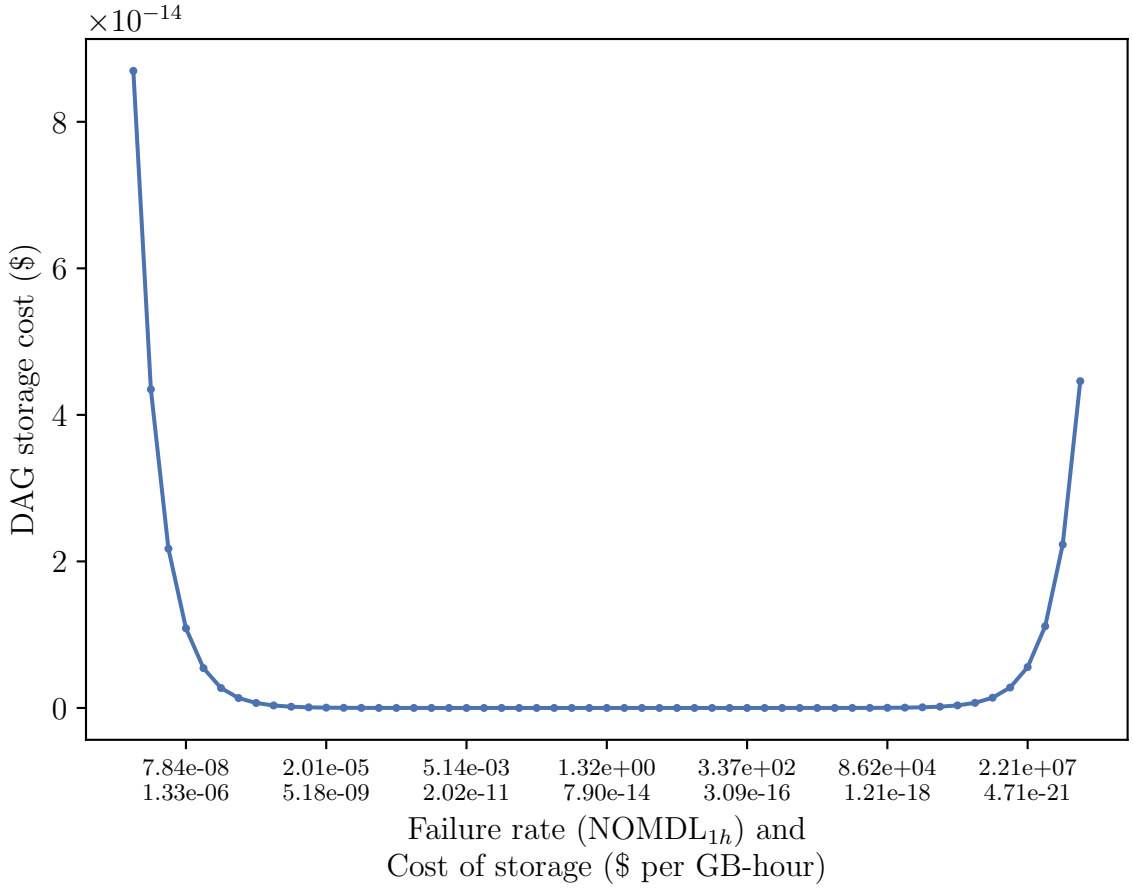


Figure 6.9: Impact of halving storage cost while doubling NOMDL_{1h}

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 [177, 115], it seems unlikely that highly-durable cloud storage will be able to compete on price with lower-durability storage any time soon.

Storage price: how low can we go? Today's storage is fairly reliable, even a single HDD or SSD. But they do cost typically hundreds of dollars per device. In this work we considered using less-durable storage for short time periods. So a natural question arose: can vendors produce

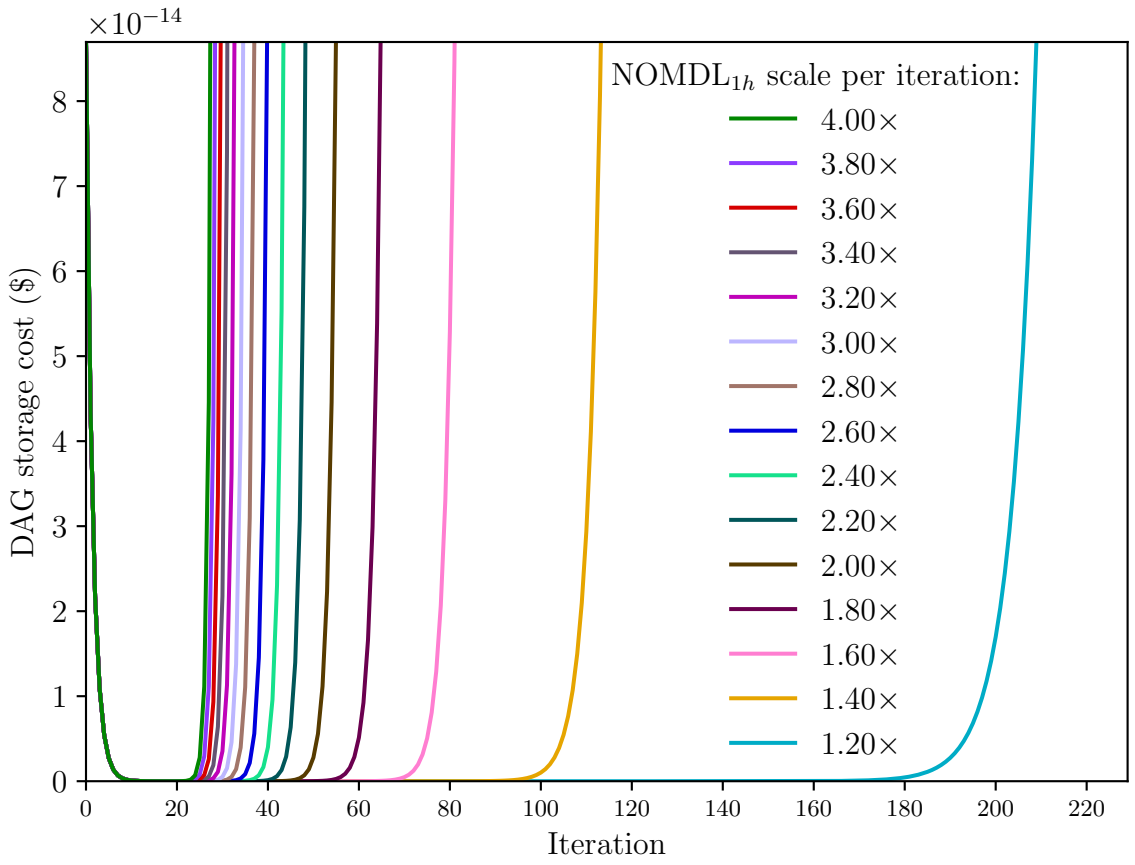


Figure 6.10: Impact of halving storage cost while scaling NOMDL$_{1h}$

devices that are far less durable than today's storage, and therefore far cheaper, but that are still durable enough for this work? Our hypothesis is that there may be a place in the storage market for super-low-cost devices whose durability is fairly low—but are quite suitable for short-term storage where the data can be regenerated if needed.

Therefore, as a hypothetical, we studied how far we can push this durability and cost tradeoff. We halved SC2's cost while doubling its failure rate for several iterations. Figure 6.9 shows the resulting storage cost plus re-execution cost at each iteration. It takes 35 iterations (SC2 cost and durability $2^{34}\times$ lower than at start) before we reach a minimum. After this point, reducing durability further causes failures to be so common that the re-execution costs outweights the savings of further reduced storage costs.

We next repeated this experiment, but varied the amount that NOMDL$_{1h}$ was scaled by each iteration from 1.2× to 4.0×. Storage cost was still halved each iteration (*i.e.*, scaled by 0.5×). The ratio of NOMDL$_{1h}$ scaling to storage cost scaling each iteration therefore ranged from 12:5 to 40:5. Figure 6.10 shows the impact of increasing this ratio. As the ratio increases, we see the point of diminishing returns shifts to the left—*i.e.*, the point where re-execution costs outweighs further cost savings. Note that even if NOMDL$_{1h}$ is increasing at a *slower* rate than the storage cost is

decreasing, we still eventually reach this point of diminishing returns.

Still, our findings here suggest that there is considerable room to decrease durability if cost can be decreased as well. Even when NOMDL_{1h} increases at twice the rate that cost decreases, we are still able to reduce cost by $2^{17}\times$ versus our starting cost for SC2.

6.5.3 Latency Analysis

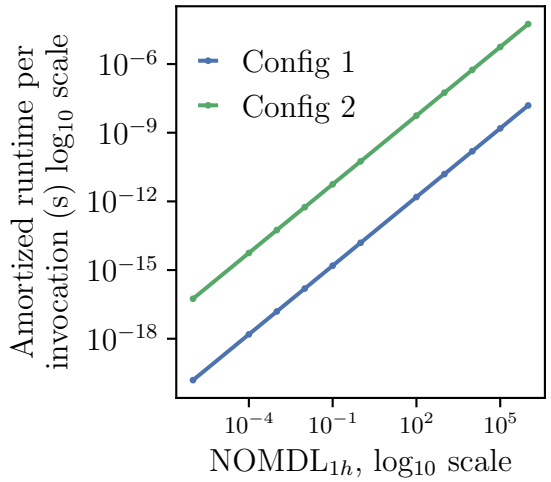
Re-running actions to re-create data incurs not only an additional compute cost, but also adds additional latency. This additional latency is amortized across many application runs, since even with low-durability storage, failures (and therefore re-runs) are infrequent. Still, even if the costs make 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 6.1, 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 6.7 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. 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).

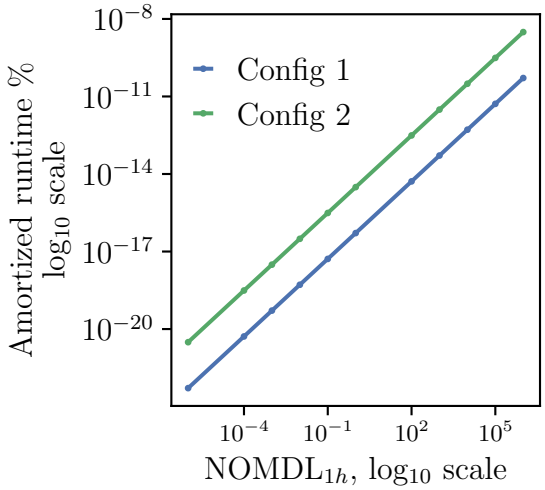
If an action needs to be re-run, the overall runtime of the DAG will increase by up to $2\times$ the runtime of a single action. This is because in the worst case, an action will run until just before completion before the data it is consuming is lost by the storage system. One action will need to be 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 6.11 shows the additional time spent re-running actions as a result of failure, amortized across the predicted number of DAG invocations between failures. Figure 6.11a shows this additional time in seconds, and Figure 6.11b 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 ($\text{NOMDL}_{1h} = 10^5$), 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.



(a) Additional time, amortized across the expected number of DAG invocations between failures.



(b) Additional time, amortized across the expected number of DAG invocations between failures, as a percentage of a single DAG's runtime.

Figure 6.11: 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. Note the $\log_{10}$ scale on both X- and Y-axes.

6.5.4 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 6.1: 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.

Table 6.3 contains the parameters used for each of the experiments. We used the storage classes defined in Section 6.4.1, and $0.0068 per second as our compute cost. Figure 6.12 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.

Video Transcoding We implement a three stage workflow, using FFmpeg [62] to transcode a video to a new resolution. Our workflow is depicted in Figure 6.13 and consists of a split stage, a parallel transcode stage, and finally a combine stage. In total, the workflow contains 12 functions 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 6.14 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).

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 6.3: Parameters used for applications in case study

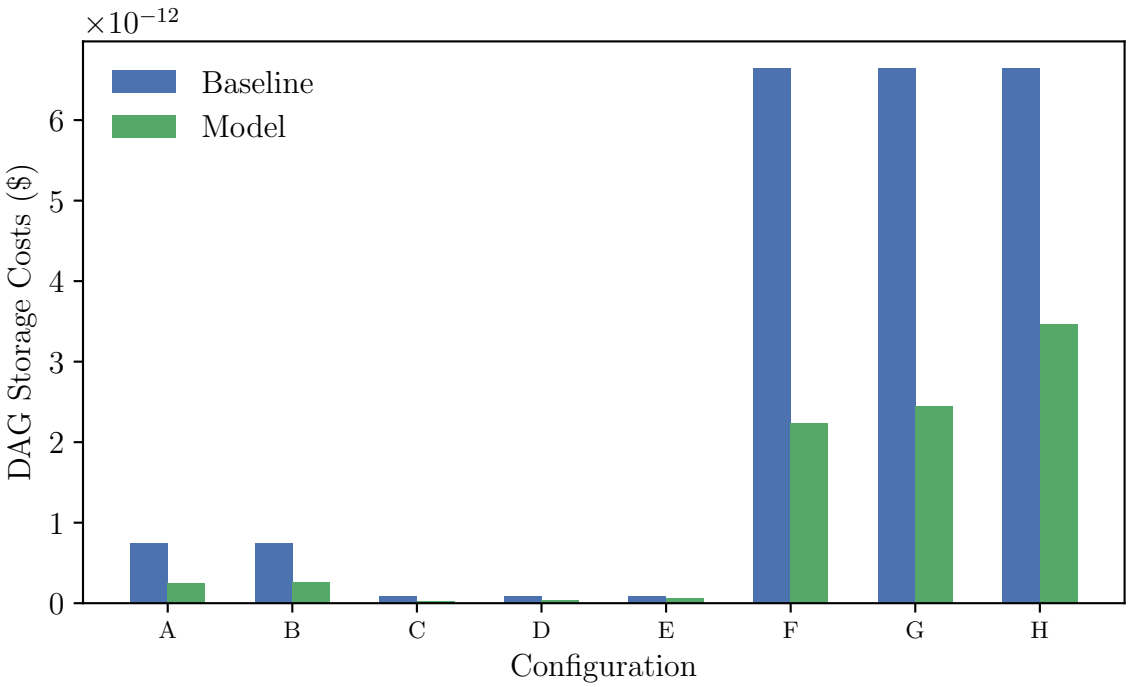


Figure 6.12: Storage component of DAG costs for applications used in case study. See Table 6.3 for configuration specifics.

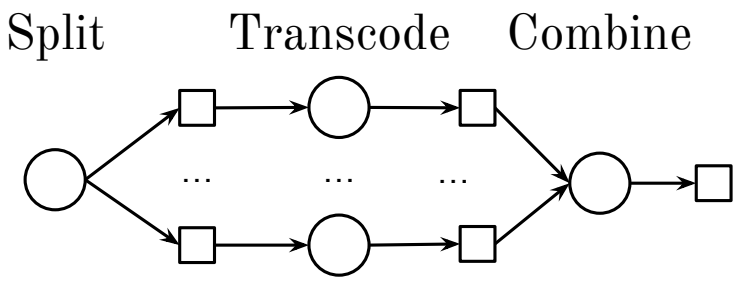
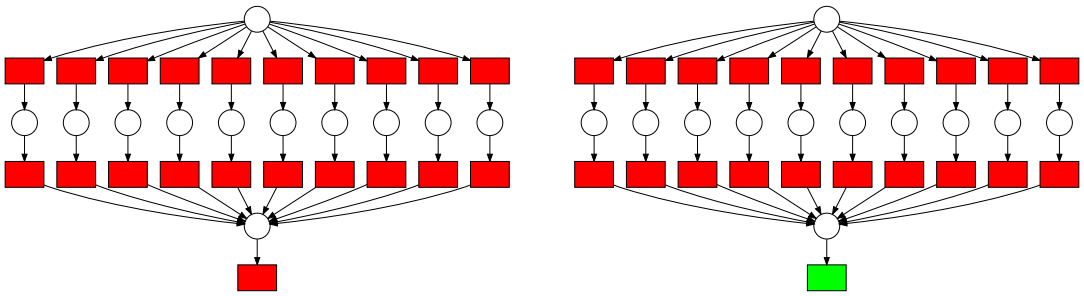


Figure 6.13: Video transcoding DAG



(a) Configuration A

(b) Configuration B

Figure 6.14: Video transcoding storage class selections. Green is SC1, red is SC2. See Table 6.3 for configuration details.

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.

Montage Montage [90] is a toolkit for processing and stitching together astronomical images. We port the workflow implemented for HyperFlow [127, 80] to run on our execution system. We run two sizes of Montage workflow, 0.25 and 1.0, with configurations listed in Table 6.3. Figure 6.15 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 6.16 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 6.3, 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

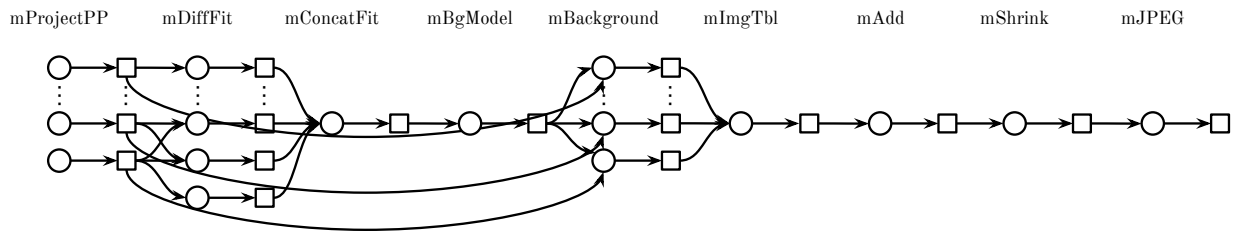


Figure 6.15: Montage DAG

the end of the DAG.

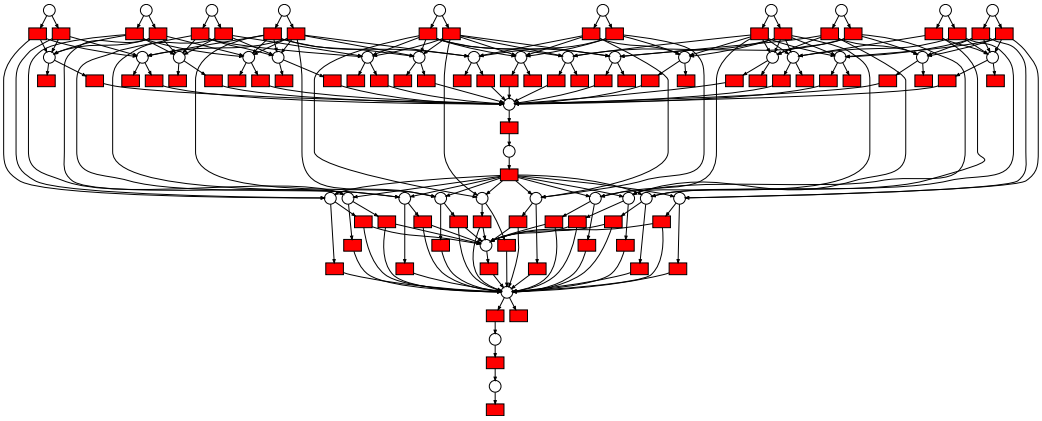
This effect is amplified by increasing NOMDL_{1h}, as seen in configuration E vs. D. Here SDCM selects SC1 for even more data, again all data towards the end of the DAG.

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.

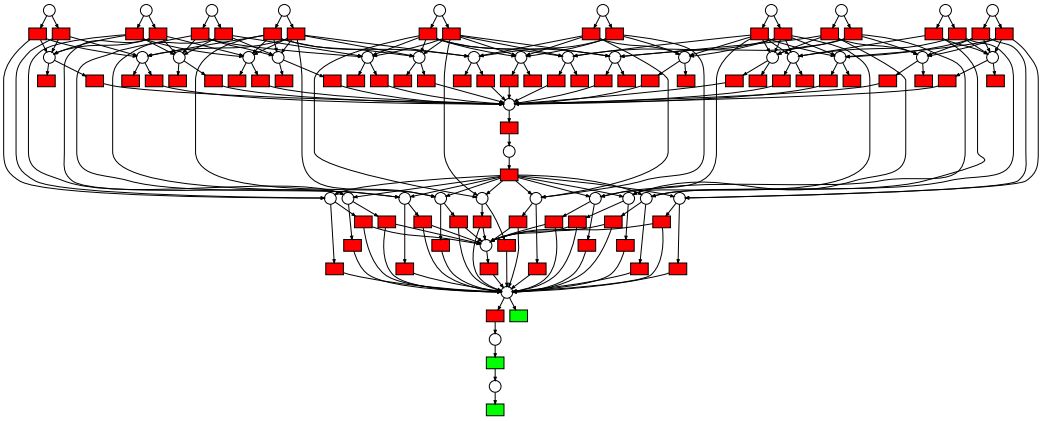
6.6 Conclusion

Storage systems have been built and operated with the assumption that high durability is necessary and desired for all kinds of data. In this chapter, 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.

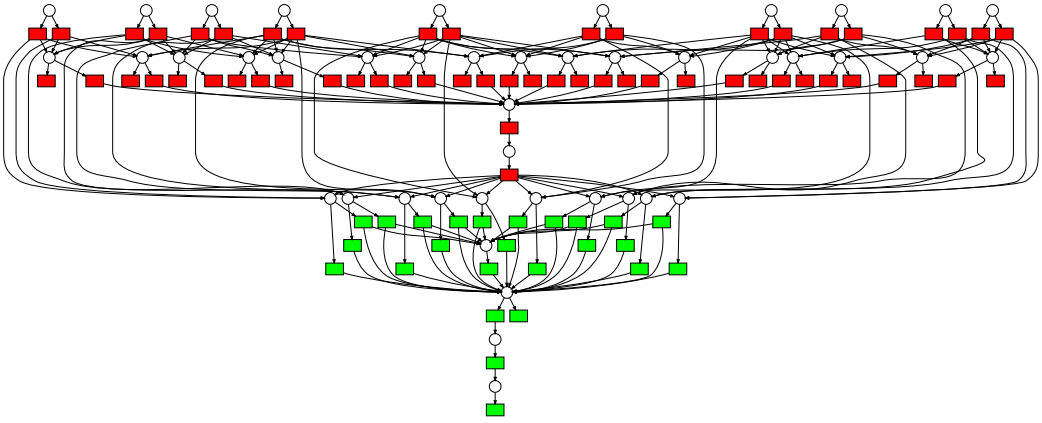
We presented a mathematical model, SDCM, that identifies the most cost effective storage class for serverless data, given application and environmental parameters. We also built an execution system using SDCM to place data, which transparently re-runs functions in response to lost data. Finally, we demonstrate how the placement decisions made by SDCM can lower storage costs when executing serverless DAGs, by up to 3×.



(a) Configuration C



(b) Configuration D



(c) Configuration E

Figure 6.16: Montage 0.25 storage class selections. Green is SC1, red is SC2. See Table 6.3 for configuration details.

Chapter 7

Conclusions

It is our thesis that the new characteristics in modern clouds, especially serverless platforms and applications, necessitate a reexamination of the tools and techniques that were designed for an older generation of cloud systems. Increases in the number of storage control operations, in the diversity of workloads, and in workload dynamism have made existing benchmarking tools insufficient to understanding application and cluster performance. Increases in the amount of ephemeral data used by applications make existing data-handing techniques sub-optimal in terms of both cost and performance, in part due to the use of excessive durability.

To address the problem of benchmarking in modern cloud and serverless environments, we have developed CNSBench. We demonstrate how CNSBench can be used to examine the performance properties of applications and storage systems running in a modern, cloud native, serverless environment.

To address the problem of ephemeral data handling, we have developed the file system called F3. F3 extends the functionality of existing shared file systems, adding optimizations for the handling of data in common cloud native use cases. We show how applications can lower their overall runtime by using F3 for data transfer.

Continuing our exploration of the storage requirements for ephemeral data, we developed a mathematical model called SDCM. SDCM chooses the appropriate durability level for a serverless application's ephemeral data, making a trade-off between durability and cost. We also developed an execution system that handles re-creating data in the event of data loss, ensuring that low-durability storage can be used with no additional burden on developers. We show how using SDCM can reduce storage costs by up to $3\times$.

7.1 Future Work

There are several areas of investigation that could follow this dissertation, which we leave to future work.

7.1.1 Serverless Platforms of the Future

The primary benefits offered by serverless—on demand compute and shifting the burden of infrastructure management to the platform—are significant steps forward in terms of how applications are run. On-demand computing greatly reduces resource wastage, and shifting responsibility for infrastructure continues the trend that began with the move from bare-metal, on-premises clusters to virtualized, cloud based clusters. In the future it is reasonable to expect that most, if not all, applications will want to be deployed on a platform with these two traits. For current serverless platforms to be able this future platform, there are a number of changes and additional capabilities that are needed.

Serverless applications Currently most applications that run on serverless platforms are designed and written specifically for running in a serverless context. This is due in part to the limited environment that serverless applications run in: runtimes are limited, access to network and hardware devices (*e.g.*, GPUs) is limited or non-existent, saving state across invocations is difficult, coordination between applications is difficult. Some of these challenges are seemingly straightforward to address. For example, runtime limits could simply be eliminated.

Others may be more difficult to fix. For instance, although there has been a lot of work on passing state between serverless functions [35, 21, 157, 125], none of these state-sharing methods have been incorporated by the major cloud serverless platforms. In general, it remains to be seen what exactly from the traditional computing world will need to be re-implemented for serverless. For example, will state passing mechanisms need to re-implement the full capabilities offered by the System V IPC API [193]? Will storage interfaces need to implement the full POSIX API [87]?

Serverless execution Our work has involved executing DAG-structured applications on serverless platforms. To do so, we have relied on a system of custom shell and Python scripts to orchestrate the execution of each stage in the DAG. Although there are some frameworks that aim to make the task of executing DAGs on serverless platforms easier [18, 128, 35, 125], none of these satisfied all of our requirements. In particular, maintaining a mapping between data generated by the DAG and the specific action responsible for creating that data is not possible with current frameworks. Additionally, existing frameworks have limited error-handling capabilities and are incapable of re-executing actions to re-create lost data. We believe that serverless DAG execution warrants further exploration.

Another area of potential research is how to re-execute actions to re-create data. Although the requirement that actions be idempotent should mean that actions can simply be re-run to re-create data, idempotency is often difficult to guarantee in practice. Execution techniques that make it easier to ensure idempotency would be helpful. For example, perhaps record-and-replay techniques could be helpful here.

Serverless and emerging memory technologies New memory technologies like CXL [48] can enable new capabilities for serverless. For example, CXL promises to make VM migration faster [45], which in turn can enable serverless functions to be longer lived. Fast migration capabilities could

also give serverless more advanced capabilities, such as allowing serverless platforms to "over-subscribe" hosts: an action's memory usage is not constant, although it must request a certain set amount up front. A serverless platform could schedule more actions than a host has memory for, knowing that if they all require their requested memory at the same time, some actions could be migrated to another host.

CXL memory that is accessible from multiple hosts could change how data is transferred between serverless actions. For one, it will accelerate data transfers that currently rely on network or intermediate storage. Another interesting use case could be the re-implementation of interfaces that applications currently use for intra-host communication. For instance, Linux's shared memory API [192] or named pipes [191] could be re-implemented to work in a serverless context. As mentioned above, it remains to be seen which of these interfaces will be required or desired for the serverless platform of the future.

On-demand memory Serverless platforms offer on-demand compute power. Storage services such AWS's S3 [2] and EFS [89] offer on-demand storage capacity. There is no platform that currently offers on-demand memory. Some virtual-machine hypervisor software such as VMWare ESXi and VirtIO/QEMU allow for some amount of ballooning, but these capabilities are intended for use during periods of extreme memory pressure [58] or must be done manually [77]. Additionally, these capabilities do not apply to containers or serverless actions.

The lack of on-demand memory in serverless environments limits the benefits of its on-demand compute capability. Developers must request a set amount of memory that will be allocated to an action for the action's entire duration. To avoid running out of memory, this amount must necessarily be the peak memory usage of the action. Any time then that the action is utilizing less than this peak represents memory being wasted. Since developers pay based on the amount of memory they request for their action, this means that for much of the action's duration they are paying for memory not being used—exactly the situation serverless solves for compute.

The fact that developers must request a set amount of memory for their action also somewhat negates the benefit of having the serverless platform be responsible for managing infrastructure: although the developer can ignore much of the computing stack, they must still consider their action's memory usage. On-demand memory would enable the developer to truly focus on just their application, without thinking at all about the realities of the underlying computing stack.

Currently the challenge with on-demand memory seems to be that there are limits to the amount of memory an individual host can contain, and accessing off-host memory requires more network capacities than currently exist. With new technologies, on-demand memory may become more feasible. CXL promises to provide both fast access to off-host memory, as well as significantly increasing the amount of memory each host can contain. Additionally, networking speeds continue to increase [183]. Making use of these new hardware capabilities to provide on-demand memory will likely be the subject of future research.

7.1.2 Storage Durability

A general, historical assumption with storage systems is that we want to be able to store data durably and for long time periods. This assumption has driven decisions such as what storage media are used for storage and how storage systems are architected. However, we propose developing a model that shows when lower degrees of durability are appropriate instead. If we no longer have the assumption that we always want high degrees of durability, does that change some of these decisions? For example, perhaps storage systems could be built using old, less reliable, but cheap hard disks.

Another possible direction of future research is to examine how data can be moved across different durability tiers. For example, as data ages it might become appropriate to move the data to higher-durability storage. Cloud storage typically offers data lifecycle management tools that can transition data to lower cost tiers as the data becomes colder [15]. Durability management tools will likely be similar, but not exactly the same as these lifecycle management tools. Whereas current data lifecycle management tools are mostly concerned with the time since data was last accessed, durability management tools would have a broader range of considerations. For example, as time passes, the cost to re-create a piece of data might reduce: a model built using a GPU five years ago will presumably be built much faster, and therefore cheaper, on a newer GPU. As the cost to re-create data decreases, it may be possible (and economical) to move the data to lower-durability storage.

Another consideration could be the age of the media data is stored on. As the media ages, the probability of failure might increase. It may be necessary to move the data to other hardware eventually, or, the data may be able to tolerate the decreased durability. In general, a durability manager will need to consider many of the same factors that we currently consider with SDCM, described in Chapter 6.

Bibliography

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 923–935, USA, 2018. USENIX Association.

[2] Amazon. *Amazon Simple Storage Service Developer Guide API Version 2006-03-01*, 2015. *http://docs.aws.amazon.com/AmazonS3/latest/dev/s3-dg.pdf*.

[3] Amazon s3 faqs, 2024. *https://aws.amazon.com/s3/faqs/*.

[4] George Amvrosiadis and Vasily Tarasov. Filebench github repository, 2016. *https://github.com/filebench/filebench/wiki*.

[5] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *SoCC '18: Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Apache Foundation, The. Hadoop, January 2010. *http://hadoop.apache.org*.

[7] Apache Foundation, The. Hdfs architecture guide, January 2010. *https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html*.

[8] Amazon Web Services (AWS). *https://aws.amazon.com/*.

[9] Building Applications with Serverless Architectures. *https://aws.amazon.com/lambda/serverless-architectures-learn-more/*.

[10] Aws lambda now supports custom runtimes and enables sharing common code between functions, November 2018. *https://aws.amazon.com/about-aws/whats-new/2018/11/aws-lambda-now-supports-custom-runtimes-and-layers/*.

[11] Using amazon efs with lambda. *https://docs.aws.amazon.com/lambda/latest/dg/services-efs.html*.

[12] How do I make my lambda function idempotent?, 2021. *https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/*.

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

[14] Aws lambda enables functions that can run up to 15 minutes, October 2018. *https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/*.

[15] Transitioning objects using amazon s3 lifecycle, 2024. *https://docs.aws.amazon.com/AmazonS3/latest/userguide/lifecycle-transition-general-considerations.html*.

[16] Serverless computing. *https://aws.amazon.com/serverless/*.

[17] Aws step functions. *https://aws.amazon.com/step-functions/*.

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

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

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

[21] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

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

[23] Andrew Bartels, Dave Bartolett, John Rymer, Matthew Guarini, Charlie Dai, and Alyssa Danilow. The public cloud market outlook, 2019 to 2022: Public cloud growth continues to power tech spending. Technical report, Forrester, July 2019.

[24] Michael Behrendt. Ibm cloud functions: we're doubling the time limit on executing actions, April 2018. *https://www.ibm.com/cloud/blog/ibm-cloud-functions-doubling-time-limit-executing-actions*.

[25] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

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

[27] James Beswick, Jerome Van Der Linden, and Dariusz 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/*.

[28] Bloomberg: An early adopter's success with Kubernetes at scale. *https://www.cncf.io/case-studies/bloomberg/*.

[29] Anthony Bolger, Marc Lohse, and Bjoern Usadel. Trimmomatic: a flexible trimmer for illumina sequence data. *Bioinformatics*, 30:2114–2120, August 2014.

[30] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proc. VLDB Endow.*, 15(8):1591–1604, apr 2022.

[31] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: Semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[32] Zhen Cao, Vasily Tarasov, Hari Raman, Dean Hildebrand, and Erez Zadok. On the performance variation in modern storage stacks. In fast2017 [60], pages 329–343.

[33] Don Capps and Tom McNeal. Analyzing NSF client performance with IOzone. In *NFS Industry Conference*. NFS Industry Conference, 2002.

[34] Eric Carter. Sysdig 2019 Container Usage Report. *https://sysdig.com/blog/sysdig-2019-container-usage-report/*.

[35] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Ceph. *https://ceph.io/*.

[37] Ceph Snapshots. *https://docs.ceph.com/en/latest/rbd/rbd-snapshot/*.

[38] Using storage classes. *https://docs.ceph.com/en/latest/radosgw/placement/#using-storage-classes*.

[39] Ceph file system. *https://docs.ceph.com/en/pacific/cephfs/index.html*.

[40] Capabilities in ceph. *https://docs.ceph.com/en/latest/cephfs/capabilities/*.

[41] Cluster File Systems, Inc. Lustre home page. *wiki.lustre.org*, 2010.

[42] Cloud Native Computing Foundation. *https://www.cncf.io/*.

[43] Karen Coombs. Storing data in a serverless application, March 2019. *https://www.oclc.org/developer/news/2019/storing-data-in-a-serverless-application.en.htm,*.

[44] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[45] J. Corbet. Live migration of virtual machines over cxl, May 2023. *https://lwn.net/Articles/931528/*.

[46] Rodrigo Crespo-Cepeda, Giuseppe Agapito, Jose Vazquez-Poletti Luis, and Mario Cannataro. Challenges and opportunities of amazon serverless lambda services in bioinformatics. In *10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 2019.

[47] Marcin Cuber and Kerry Kamil. News UK Keeps New Content and Capabilities Coming Fast with Amazon EKS and New Relic. *https://blog.newrelic.com/product-news/news-uk-content-capabilities-amazon-eks-new-relic/*.

[48] Compute express link™: The breakthrough cpu-to-device interconnect cxl™. *https://www.computeexpresslink.org/*.

[49] The state of serverless, May 2021. *https://www.datadoghq.com/state-of-serverless/*.

[50] Carlos de Rojas. Portable sequencing is reshaping genetics research, April 2022. *https://www.labiotech.eu/in-depth/portable-sequencing-genetics-research/*.

[51] Haoran Ding, Zhaoguo Wang, Zhuohao Shen, Rong Chen, and Haibo Chen. Automated verification of idempotence for stateful serverless applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 887–910, Boston, MA, July 2023. USENIX Association.

[52] Paul Dix. Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB performance for InfluxDB, 2014. *https://www.influxdata.com/benchmarkingleveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-forinfluxdb/* (visited on 05/26/2017).

[53] Docker. *https://docker.com/*.

[54] What is a Container? *https://www.docker.com/resources/what-container*.

[55] Docker Hub. *https://hub.docker.com/*.

[56] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[57] Jon G. Elerath and Jiri Schindler. Beyond mttdl: A closed-form raid 6 reliability equation. *ACM Trans. Storage*, 10(2), mar 2014.

[58] Memory balloon driver, May 2019. *https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-5B45CEFA-6CC6-49F4-A3C7-776AAA22C2A2.html*.

[59] Facebook. RocksDB. *https://rocksdb.org/*, September 2019.

[60] *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February-March 2017. USENIX Association.

[61] Dmitrii Fediuk. Increasing dataset sizes, May 2019. *https://dmitry.ai/t/topic/198*.

[62] Ffmpeg. *https://ffmpeg.org/*.

[63] fio—flexible I/O tester. *http://freshmeat.net/projects/fio/*.

[64] Sara Ford and Martin 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*.

[65] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[66] The Linux Foundation. State of the edge 2021, 2019. *https://www.lfedge.org/wp-content/uploads/2021/08/StateoftheEdgeReport_2021_r3.11.pdf*.

[67] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An Open-source Benchmark Suite for Microservices and their Hardware-software Implications for Cloud & Edge Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[68] The edge will eat the cloud. *https://blogs.gartner.com/thomas_bittman/2017/03/06/the-edge-will-eat-the-cloud/*.

[69] Predicts 2022: The distributed enterprise drives computing to the edge. *https://www.gartner.com/document/4007176*.

[70] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, oct 2003.

[71] Samadrita Ghosh. A comprehensive guide to data preprocessing, August 2023. *https://neptune.ai/blog/data-preprocessing-guide*.

[72] Google Cloud. *https://cloud.google.com/*.

[73] Google cloud functions. *https://cloud.google.com/functions*.

[74] Introducing general parallel file system, March 2021. *https://www.ibm.com/docs/en/gpfs/4.1.0.4?topic=guide-introducing-general-parallel-file-system*.

[75] 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.

[76] James Lee Hafner and KK Rao. Notes on reliability models for non-mds erasure codes. *IBM Research Report*, 2006.

[77] Philipp Matthias Hahn. Memory balloon driver, May 2019. *https://pmhahn.github.io/virtio-balloon/*.

[78] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, Santa Clara, CA, February 2016. USENIX Association.

[79] Red Hat. A hybrid and multicloud strategy for system administrator. Technical Report #F21608_0220, Red Hat, 2020.

[80] Ffmpeg hyperflow wms. *https://github.com/hyperflow-wms*.

[81] IBM Cloud. *https://www.ibm.com/cloud*.

[82] Ibm cloud functions. *https://cloud.ibm.com/functions*.

[83] Containerization. *https://www.ibm.com/cloud/learn/containerization*.

[84] What is faas (function-as-a-service, July 2019. *https://www.ibm.com/cloud/learn/faas*.

[85] Object vs. file vs. block storage: What's the difference?, October 2021. *https://www.ibm.com/cloud/blog/object-vs-file-vs-block-storage*.

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

[87] IEEE/ANSI. Information technology–portable operating system interface (POSIX)–part 1: System application: Program interface (API) [C language]. Technical Report STD-1003.1, ISO/IEC, 1996.

[88] Ilias Iliadis and Vinodh Venkatesan. Rebuttal to "beyond mttdl: A closed-form raid-6 reliability equation". *ACM Trans. Storage*, 11(2), mar 2015.

[89] Amazon Inc. Amazon elastic file system. *https://aws.amazon.com/efs/*, September 2015.

[90] Joseph C. Jacob, Daniel S. Katz, G. Bruce Berriman, John C. Good, Anastasia C. Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas A. Prince, and Roy Williams. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *Int. J. Comput. Sci. Eng.*, 4:73–87, July 2009.

[91] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445—451. Association for Computing Machinery, 2017.

[92] Nikolai Joukov, Ashivay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In OSDI 2006 [151], pages 89–102.

[93] Drivers - kubernetes csi developer documentation, March 2022. *https://kubernetes-csi.github.io/docs/drivers.html*.

[94] Harshad Kasture and Daniel Sanchez. TailBench: A Benchmark Suite and Evaluation Methodology for Latency-critical Applications. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[95] Sachin Katti, John Ousterhout, Guru Parulkar, Marcos Aguilera, and Curt Kolovson. Scalable control plane substrate.

[96] Kibana. *https://www.elastic.co/kibana*.

[97] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 789–794, Boston, MA, July 2018. USENIX Association.

[98] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[99] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[100] Knative is an open-source enterprise-level solution to build serverless and event driven applications. *https://knative.dev/docs/*.

[101] Stefan Kolb. *On the Portability of Applications in Platform as a Service*, volume 34. University of Bamberg Press, 2019.

[102] Kubeflow, 2023. *https://www.kubeflow.org/*.

[103] Kubernetes. *https://kubernetes.io/*.

[104] Dynamic Provisioning and Storage Classes in Kubernetes. *https://bit.ly/2Uh3Qbw*.

[105] Ephemeral volumes. *https://kubernetes.io/docs/concepts/storage/ephemeral-volumes/*.

[106] Improve kubectl cp, so it doesn't require the tar binary in the container #58512. *https://github.com/kubernetes/kubernetes/issues/58512*.

[107] kubectl cp to work on stopped/completed pods #454. *https://github.com/kubernetes/kubectl/issues/454*.

[108] Labels and selectors, August 2022. *https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/*.

[109] Kubernetes Object Management. *https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/*.

[110] Operator pattern. *https://kubernetes.io/docs/concepts/extend-kubernetes/operator/*.

[111] Kubernetes scheduler, December 2022. *https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/*.

[112] Volume Snapshot & Restore - Kubernetes CSI Developer Documentation. *https://kubernetes-csi.github.io/docs/snapshot-restore-feature.html*.

[113] Kubernetes Storage. *https://kubernetes.io/docs/concepts/storage/*.

[114] Tools for monitoring resources. *https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/*.

[115] Leon 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*.

[116] Error handling and automatic retries in aws lambda. *https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html*.

[117] Building large clusters. *https://kubernetes.io/docs/setup/best-practices/cluster-large/*.

[118] D. Lelewer and D. Hirschberg. Data compression. In *ACM Computing Surveys (CSUR)*, pages 261–296. ACM, 1987.

[119] Dinghua Li, Chi-Man Liu, Ruibang Luo, Kunihiko Sadakane, and Tak-Wah Lam. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. In *Bioinformatics*, 2015.

[120] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi 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*, ASPLOS '22, page 782–796, New York, NY, USA, 2022. Association for Computing Machinery.

[121] Johanan Liebermann. Golang, June 2017. *https://codeburst.io/why-golang-is-great-for-portable-apps-94cf1236f481*.

[122] 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/*.

[123] Yijie Liu, Zhuo Huang, Jianhui Yue, Hanxiang Huang, Song Wu, and Jin Hai. Funcstore: Resource efficient ephemeral storage for serverless data sharing. In *Proceedings of the 38th Symposium on Mass Storage Systems and Technologies (MSST)*, 2024.

[124] Gilad David Maayan. Storage options for serverless on aws, June 2020. *https://hackernoon.com/storage-options-for-serverless-on-aws-fo3x3wsv*.

[125] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 285–301. USENIX Association, July 2021.

[126] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh 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)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[127] Maciej Malawski. Towards serverless execution of scientific workflows-hyperflow case study. In *Works@ Sc*, pages 25–33, 2016.

[128] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems*, 110:502–514, 2020.

[129] Marcel Martin. Cutadapt removes adapter sequences from high-throughput sequencing reads. *EMBnet.journal*, 17(1):10–12, May 2011.

[130] Jack McElwee and Allan Krans. Public cloud benchmark: First calendar quarter 2020. Technical report, Technology Business Research, July 2020.

[131] Memcached, 2018. *https://memcached.org/*.

[132] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Julie Lee, Lukas Rupprecht, Dimitris Skourtis, Yang Yang, and Erez Zadok. CNSBench: A cloud native storage benchmark native storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*, Virtual, February 2021. USENIX Association.

[133] Alex Merenstein, Vasily Tarasov, Ali Anwar, Deepavali Bhagwat, Lukas Rupprecht, Dimitris Skourtis, and Erez Zadok. The case for benchmarking control operations in cloud native storage. In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[134] Alex Merenstein, Vasily Tarasov, Ali Anwar, Scott Guthridge, and Erez Zadok. F3: Serving files efficiently in serverless computing. In *Proceedings of the 16th ACM International Systems and Storage Conference (SYSTOR '23)*, Haifa, Israel, June 2023. ACM. Won Best Paper Award.

[135] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur 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)*, pages 177–190, Portland, OR, June 2015. ACM.

[136] Microsoft Azure. *https://azure.microsoft.com/*.

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

[138] Minio. *https://min.io/*.

[139] Pulkit A. Misra, María F. Borge, Íñigo Goiri, Alvin R. Lebeck, Willy Zwaenepoel, and Ricardo Bianchini. Managing tail latency in datacenter-scale file systems under production constraints. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[140] mitmproxy. *https://mitmproxy.org/*.

[141] MongoDB, Inc. MongoDB: The database for modern applications. *https://www.mongodb.com/*, September 2019.

[142] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[143] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD failures in datacenters: What? when? and why? In *Proceedings of the Ninth ACM Israeli*

Experimental Systems Conference (SYSTOR '16), pages 7:1–7:11, Haifa, Israel, May 2016. ACM.

[144] Ebs pricing and performance: A comparison with amazon efs and amazon s3. *https://cloud.netapp.com/blog/ebs-efs-amazons3-best-cloud-storage-system*.

[145] Xingzhi Niu, Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. Leveraging serverless computing to improve performance for sequence comparison. In *10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, 2019.

[146] OpenEBS. *https://openebs.io/*.

[147] OpenEBS cStor CSI driver. *https://github.com/openebs/cstor-csi/blob/master/pkg/driver/controller_utils.go#L243*.

[148] OpenEBS replication.c. *https://github.com/openebs/istgt/blob/replication/src/replication.c\#L1958*.

[149] Open source serverless cloud platform. *https://openwhisk.apache.org/*.

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

[151] *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, November 2006. ACM SIGOPS.

[152] Filebench pre-defined personalities, 2016. *http://filebench.sourceforge.net/wiki/index.php/Pre-defined_personalities*.

[153] pgbench. *https://www.postgresql.org/docs/10/pgbench.html*.

[154] Portworx Kubernetes Snapshots and Backups. *https://docs.portworx.com/portworx-install-with-kubernetes/storage-operations/kubernetes-storage-101/snapshots/*.

[155] PostgreSQL Global Development Team. PostgreSQL. *www.postgresql.org*, 2011.

[156] Evan Powell. Container Attached Storage is Cloud Native Storage (CAS). *https://www.cncf.io/blog/2020/09/22/container-attached-storage-is-cloud-native-storage-cas/*.

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

[158] Object vs file storage: When and why to use them, May 2022. *https://blog.purestorage.com/purely-informational/object-vs-file-storage-when-and-why-to-use-them/*.

[159] Corey Quinn. S3 reduced redundancy storage is dead, April 2017. *https://www.lastweekinaws.com/blog/s3-reduced-redundancy-storage-is-dead/*.

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

[161] File storage vs. object storage: What's the difference and why it matters. *https://www.quobyte.com/storage-explained/file-vs-object-storage*.

[162] Ajeet Raina. Redis use case examples for developers, July 2022. *https://redis.io/blog/5-industry-use-cases-for-redis-developers/*.

[163] Redis Labs. Redis. *https://redis.io/*, September 2019.

[164] H. Reiser. Mongo - the main benchmark script we use for comparing ReiserFS variations. *www.namesys.com/benchmarks/mongo_readme.html*, December 2002.

[165] What is faas?, January 2020. *https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas*.

[166] How to choose your red hat enterprise linux file system, September 2020. *https://access.redhat.com/articles/3129891*.

[167] Rook. *https://rook.io/*.

[168] Frank Della Rosa. Implementation of microservices architecture hastens across industries. Technical Report #US46108319, IDC, 2020.

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

[170] s3fs-fuse: Fuse-based file system backed by amazon s3. *https://github.com/s3fs-fuse/s3fs-fuse*.

[171] Tony 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/*.

[172] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton, and Joseph M. Hellerstein. A faas file system for serverless computing, 2020.

[173] 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)*, pages 1–16, San Jose, CA, February 2007. USENIX Association.

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

[175] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 67–80, Santa Clara, CA, February 2016. USENIX Association.

[176] S. Shepler, M. Eisler, and D. Noveck. NFS version 4 minor version 1 protocol. RFC 5661, Network Working Group, January 2010.

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

[178] SPEC SFS 2014. *https://www.spec.org/sfs2014/*.

[179] SPEC SFS®2014. *https://www.spec.org/sfs2014/*.

[180] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. In *Proceedings of the VLDB Endowment, Volume 13, Issue 12*, pages 2438–2452, 2020.

[181] Golang. *https://stackshare.io/golang*.

[182] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the NodeKernel architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 767–782, Renton, WA, July 2019. USENIX Association.

[183] John Swanson, Jon Ames, Priyank Shukla, and Varun Agrawal. Meeting the world's growing bandwidth demands with a complete 1.6t ethernet ip solution, Feb 2024. *https://www.synopsys.com/blogs/chip-design/1-6t-ethernet-specification-ip.html*.

[184] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.

[185] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *Proceedings of the Linux Symposium*, pages 83–95, Ottawa, Canada, July 2014.

[186] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login:*, 41(1), 2016.

[187] AIM Technology. AIM multiuser benchmark - suite VII version 1.1. *http://sourceforge.net/projects/aimbench*, 2001.

[188] Global edge computing market to reach $156 billion by 2030. *https://www.techrepublic.com/article/global-edge-computing-market/*.

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

[190] Johannes Thönes. Microservices. *IEEE Software*, 32(1), 2015.

[191] Linus Torvalds. pipe(7) — linux manual page. *https://man7.org/linux/man-pages/man7/pipe.7.html*, 2023.

[192] Linus Torvalds. shm_overview(7) — linux manual page. *https://man7.org/linux/man-pages/man7/shm_overview.7.html*, 2023.

[193] Linus Torvalds. sysvipc(7) — linux manual page. *https://man7.org/linux/man-pages/man7/sysvipc.7.html*, 2023.

[194] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P Wright. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2), 2008.

[195] Bharath Kumar Reddy Vangoor, Prafful Agarwal, Manu Mathew, Arun Ramachandran, Swaminathan Sivaraman, Vasily Tarasov, and Erez Zadok. Performance and resource utilization of FUSE user-space file systems. *ACM Transactions on Storage (TOS)*, 15(2), May 2019.

[196] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of user-space file systems. In fast2017 [60], pages 59–72.

[197] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs - a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 319–324, 2018.

[198] Pablo Villalobos and Anson Ho. mitmproxy, September 2022. *https://epochai.org/blog/trends-in-training-dataset-sizes*.

[199] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16. IEEE Press, 2016.

[200] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. Boxer: Data analytics on network-enabled serverless platforms. In *Conference on Innovative Data Systems Research*, 2021.

[201] Going Cloud Native: 6 essential things you need to know. *https://www.weave.works/technologies/going-cloud-native-6-essential-things-you-need-to-know/*.

[202] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In OSDI 2006 [151], pages 307–320.

[203] Sage Weil, Andrew Leung, Scott Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage (PDSW)*, 2007.

[204] Joe Wigglesworth. Inside the storage/compute servers of ibm spectrum fusion hci, August 2022. *https://hardware-fusion.blogspot.com/2022/08/inside-storagecompute-servers-of-ibm.html*.

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

[206] Filebench workload model language (WML), 2016. *https://github.com/filebench/filebench/wiki/Workload-Model-Language*.

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

[208] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. Lessons and actions: What we learned from 10k SSD-Related storage system failures. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 961–976, Renton, WA, July 2019. USENIX Association.

[209] Jie Yu, Saad Ali, and James DeFelice. Container Storage Interface (CSI) Specification. *https://github.com/container-storage-interface/spec/blob/master/spec.md*.

[210] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[211] Qing Zheng, Haopeng Chen, Yaguang Wang, Jian Zhang, and Jiangang Duan. COSBench: Cloud Object Storage Benchmark. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2013.