The Case for Benchmarking Control Operations in Cloud Native Storage

Alex Merenstein[1], Vasily Tarasov[2], Ali Anwar[2], Deepavali Bhagwat[2],
Lukas Rupprecht[2], Dimitris Skourtis[2], and Erez Zadok[1]
[1]*Stony Brook University* and [2]*IBM Research–Almaden*

Abstract

Storage benchmarking tools and methodologies today suffer from a glaring gap—they are incomplete because they omit storage *control operations*, such as volume creation and deletion, snapshotting, and volume reattachment and resizing. Control operations are becoming a critical part of cloud storage systems, especially in containerized environments like Kubernetes, where such operations can be executed by regular non-privileged users. While plenty of tools exist that simulate realistic data and metadata workloads, control operations are largely overlooked by the community and existing storage benchmarks do not generate control operations. Therefore, for cloud native environments, modern storage benchmarks fall short of serving their main purpose—holistic and realistic performance evaluation. Different storage provisioning solutions implement control operations in different ways, which means we need a unified storage benchmark to contrast and comprehend their performance and expected behaviors. In this position paper, we motivate the need for a cloud native storage benchmark by demonstrating the effect of control operations on storage provisioning solutions and workloads. We identify the challenges and requirements when implementing such benchmark and present our initial ideas for its design.

1 Introduction

Storage benchmarking is a well established and prolific field [32, 36]. Most systems papers conclude with a system evaluation using one or more benchmarks. Industry uses benchmarks for performance and regression testing—and sometimes publishes results to inform and attract customers [1]. Many storage benchmarks are publicly available and widely used [12, 21, 34].

Benchmarks (1) generate workloads and (2) measure a system's performance under those workloads—to analyze system behavior and compare to others. Storage benchmarks can be divided into two classes of workload generation: micro and macro [36]. Micro benchmarks generate a relatively simple workload focusing on a few I/O operations to stress-test one system component or operational mode. Conversely, macro benchmarks try to mimic real-world workloads by generating a complex mix of operations with realistic properties—with the purpose of exercising a system more holistically. Trace analysis is frequently used to create macro benchmarks [33].

Both micro and macro benchmarks focus solely on data operations (e.g., reads, writes) and metadata operations (e.g., file creates and deletes). The third type of operations—*control*—is largely ignored. Control operations include volume creations, deletions, attachments, detachments, restores, and resizes; partitioning and formatting; and snapshot creations and deletions.

In the past, control operations were relatively infrequent—invoked only by system administrators. However, in cloud native environments, control operations are routinely used by non-privileged users and even automated users [11, 15]. In the cloud, owners and users control their applications and storage exclusively, and they do not need administrators. Therefore, the frequency of control operations has surged (see Section 2) and is mostly dependent on the number of users and the speed of software development and deployment.

Modern container clusters contain thousands of physical nodes and can run hundreds of thousands of containers for many users [3]. A storage system (e.g., Amazon Elastic Block Store or a Ceph cluster) serving such large-scale deployments inevitably experiences a high rate of control operations—volume creations, deallocations, attachments, reattachments, etc.

The fundamental shift towards cloud native environments creates a high, previously unseen rate of control operations. The situation is further exacerbated by three more reasons. (1) The use of the microservices architecture [35] in modern applications increases the number of independent containers that may require their own storage volumes [25]. (2) Control operations can be data intensive: this makes them slow, but also increases their impact on the data and metadata operations. For example, volume creation often requires file system formatting; volume resizing may require data migration; and volume reattachment requires cache flushes and warmups. (3) In the past, control operations were functionally tested but rarely benchmarked. Anecdotally, we know that even high-performance storage systems may be slow because they were not designed to service many control operations.

Thus, there is a gap in the available storage benchmarking solutions as they ignore cloud native use cases. We propose to fill this gap by introducing a design of a cloud native storage benchmark that focuses on control operations. This position paper makes the following contributions:

1. We explore the issue of overlooked control operations and their growing importance in cloud native environments.
2. We experimentally demonstrate that the performance of control operations varies widely across storage systems, and even affects data operations in cloud native environment.
3. We present the requirements, challenges, and initial design for a flexible, inclusive cloud native storage benchmark.

2 Control Operations in Kubernetes

Control operations are becoming more important in cloud native, containerized setups. Compared to traditional virtual machine

environments, the lifecycle of resources in a container environment can be shorter, whereas the number of resources, such as containers and volumes, can be greater. The low overhead of running containers and the micro-service architecture breeds an environment of ephemeral, short-lived resource use. For example, there was a reported "2× increase in the number of containers that live for less than five minutes" in 2019 compared to 2018 on one DevOps platform [23]. Fast container startup times, along with high container churn rate, result in frequent control operations (e.g., creating, attaching, or mounting volumes). Consequently, control operations are becoming latency sensitive, adding significant load to the storage system and affecting I/O operations and user experience.

Next, we describe how Kubernetes, a popular container orchestrator, manages storage volumes. Although we focus on Kubernetes, similar concepts exist in other orchestrators such as Apache Mesos [2] and Docker Swarm [10].

Kubernetes is a platform for running containerized applications in a cloud native fashion [14]. In Kubernetes, if a user wants to deploy a new application that requires persistent storage, the user has to create a Persistent Volume Claim (PVC) to request a new file system or block-based storage volume. A PVC describes the high-level properties of the volume such as size or expected performance. Using the specific storage system's Container Storage Interface (CSI) [7] driver, Kubernetes automatically communicates with the storage system to allocate a volume. When users start an application that uses a provisioned volume, Kubernetes automatically attaches the volume to the node where the application's container is scheduled to run.

Kubernetes defines various resources such as the *Pod* (a group of containers) and the *Service* (a frontend to a set of Pods) as abstractions for deploying applications. Similar to compute and networking resources, the orchestrators provide storage in the form of *volumes* for application consumption. A volume in Kubernetes is called a *Persistent Volume* (PV); it is usually backed by storage hosted on a given storage provider (SP). A PV can be provisioned either dynamically or manually by an administrator.

A user can request a PV by creating a *Persistent Volume Claim* (PVC), specifying properties such as the required volume size and access mode. The backing storage solution and its parameters, referred to as the *provisioner*, can be expressed through a *Storage Class* (SC). Upon creation of a PVC, Kubernetes tries to find a corresponding PV that matches the request or asks the provisioner to create a PV dynamically. The PV is then bound to the PVC.

Assuming that a PVC is bound to a PV, a user can consume it through a Pod as a device or file system. During Pod creation, once the Pod has been scheduled to a node, the PV is published or staged onto that node and then published to the Pod. Alternatively, a PVC may be described as part of a Pod specification. In that case, dynamic provisioning takes place during Pod creation. Staging and publishing operations must complete prior to the Pod becoming available to the user, making their performance critical.

To support different SPs, Kubernetes, as well as other orchestrators, use the Container Storage Interface (CSI) [7]. CSI enables SPs to develop and maintain a single plugin for all orchestrators that support the CSI specification. There are currently 23 operations specified in the CSI specification, ranging from volume and snapshot creation, to publishing volumes to nodes, and listing volume information.

Next we illustrate a Kubernetes-specific example that triggers several control operations. Suppose that an application developer wants to deploy multiple web server instances. Furthermore, assume that the local administrator has set up a storage system that provides a CSI plugin and has also created a Storage Class corresponding to the storage system. The developer is given access to the cluster and creates a Pod for their web server and a PVC for the volume, to be used by the web server. The PVC specifies that the volume should be provisioned from the Storage Class created previously by the cluster administrator.

At this point, Kubernetes creates a PVC resource describing the volume request. This triggers a call to the storage system's CSI plugin's `CreateVolume` implementation, which creates the volume as intended by the SP. Upon creation of the volume `CreateVolume` returns a PV specification, which is then created by Kubernetes and is bound to the PVC that initiated the process.

Once the PV has been bound to the PVC and the web server's Pod is created and scheduled, the CSI plugin's `NodeStageVolume` operation is invoked. `NodeStageVolume` prepares the PV to be used by a Pod by formatting it with a file system and mounting its volume on the Node where the Pod is scheduled.

Finally, the PV's file system needs to be mounted to the specific Pod. This happens through the `NodePublishVolume` operation. Here, the `NodePublishVolume` simply bind-mounts the node directory under a path corresponding to the Pod's volume. The web server Pod now starts and the volume becomes visible to the Pod. In practice, a developer would deploy multiple web server instances, potentially pointing to the same volume, leading to multiple volume staging and publishing events.

How many control operations per second should an SP be able to sustain? We develop an example based on statistics from a DevOps environment [23], where (1) each host runs a median of 30 containers; (2) 54% of containers have a lifespan of up to 5 minutes. Furthermore, we assume that the storage system is attached to 1,000 hosts. This results in an estimate of $1000*30/(5*60) = 100$ container creations per second, implying that the SP in this case needs to support 100 attach (and detach) operations per second under typical operation. This shows that the number of control operations in cloud native environments can be far larger than is typically seen elsewhere—hence the need to benchmark control operations carefully.

3 Impact of Control Operations

To better illustrate the need for a cloud native storage benchmark, we demonstrate the importance of control operations using two different experiments: (1) speed of volume creation and attach-

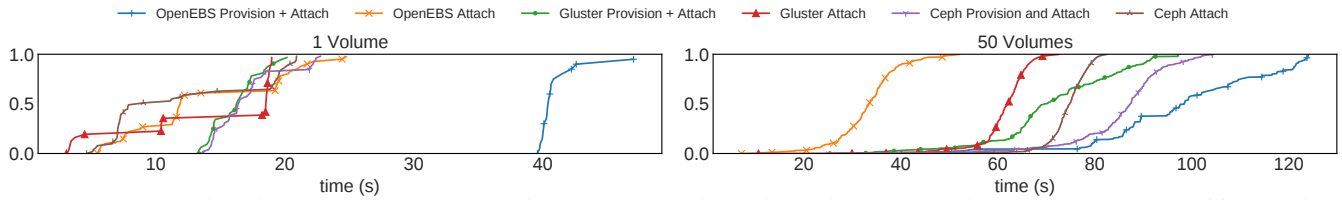

Figure 1: CDFs of how long it took storage providers to create and attach a volume to a Pod. Times were measured by attaching (i) a single volume at a time (in a series of 50) and (ii) 50 volumes simultaneously.

ment; and (2) volume snapshotting impact on I/O operations.

Experimental setup. We instantiated a small Kubernetes cluster consisting of three master nodes in a high availability configuration and two worker nodes. Each node in the cluster was a virtual machine in a VMware vSphere environment and was allocated 8 GB of RAM and 4 vCPUs. The nodes ran Centos 7.7.1908 with Linux kernel version 3.10, and we used Kubernetes version 1.16.

Due to the rapid rise of containers' popularity, many new and existing storage products position themselves as cloud native [4, 13, 16, 17, 19, 22, 24]. We used three popular and different-by-design storage providers—OpenEBS, Gluster, and Ceph—to provision Kubernetes volumes.

OpenEBS [17] is a cloud native storage provider that follows the *Container Attached Storage* approach. This means that OpenEBS deploys a separate container-based controller for each PV which handles control operations for that PV. This approach is more flexible as it permits each application to specify different storage parameters (e.g., replication factor). We used OpenEBS v1.15 and the OpenEBS cStor [8] storage engine, which formats disks with ZFS and then provisions PVs by creating ZFS volumes and formatting them with Ext4.

Gluster [13] is a distributed file system: it aggregates file systems on local nodes to form volumes that can be mounted via NFS or a Gluster FUSE driver. We configured a single node Gluster cluster, running Gluster v6.0 on just one of our Kubernetes workers.

Ceph [4] uses the RADOS [37] object store to provide object, block, or file storage systems. We created a Ceph v14.2.7 storage cluster and provisioned PVs by creating Ceph Block Devices from the storage pool and formatting the block devices with Ext4.

All three storage providers we used support common storage control operations such as provisioning, snapshotting, and resizing volumes; all three expose these operations to Kubernetes through their respective CSI drivers.

Experiment 1: Volume creation and attachment. We compared the performance of volume attachment and creation for OpenEBS, Gluster, and Ceph.

To measure volume attachment and creation, we timed how long it took to start a Pod that used a 1 GiB volume provisioned from one of the three storage providers. We used both pre-existing and new volumes: using pre-existing volumes measures volume attachment time; using new volumes measures the combined volume attachment and volume provisioning time. Because we measure the overall time to start a Pod, both cases also include the extra time needed to create the Pod itself; however, this time is small (only a few seconds) compared to the other two operations and is the same regardless of the storage provider used. To see the effect of a large number of simultaneous control operations, we tested both starting (1) only one Pod at a time and also (2) starting 50 Pods at once.

Figure 1 shows the results in CDF form. We can clearly see that there are differences in how storage providers perform these control operations. For example, although OpenEBS is the fastest storage provider at attaching pre-provisioned volumes, it is the slowest at provisioning and then attaching volumes. This could be due to the fact that OpenEBS allocates and starts a controller Pod for each new volume, which adds overhead to volume provisioning not present in Gluster or Ceph.

Both the variation in and the overall provision times for all three storage providers are surprisingly high. This is in part due to Kubernetes's asynchronous architecture: some components batch updates and others poll for updates periodically. Since provisioning volumes consists of a series of steps, delays at each step can compound, increasing provisioning times and their variance. More experimentation is needed to determine exactly which parts of the provisioning process account for the high delays and variance.

The speed with which volumes can be provisioned and attached to application Pods mainly affects Pod startup time, which in turn affects failure recovery time and the time required to scale out applications. For example, Pods using OpenEBS volumes would likely recover faster in the event of an application failure, as OpenEBS is fastest at re-attaching volumes to new Pods. Conversely, Pods using Gluster or Ceph volumes could better respond to load increases by scaling out and deploying additional Pods that require additional volumes.

Experiment 2: Snapshotting. Next, we aimed to study the impact of snapshotting on I/O workloads. Although many storage solutions now provide low-overhead volume snapshots, the actual overhead incurred by the application workload depends on various factors, including the characteristics of the workload and the storage solution itself. In addition, snapshots have the potential to impact the workloads of other users that are either co-located on the same node or that share the same underlying storage.

To measure the effects introduced by snapshotting, we used the `fio` [12] benchmark to generate a workload with five `fio` threads accessing the same 5 GB file with an even mix of sequential reads and writes. This workload is based

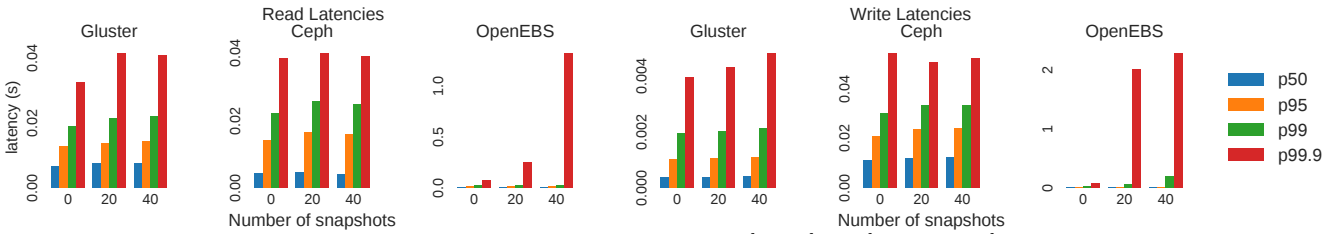

Figure 2: Effect of snapshotting on latency of I/O operations. We show 50^{th}, 95^{th}, 99^{th}, and 99.9^{th} percentiles. Note that the Y axes' scales are different for each plot, because the ranges of latencies varies significantly across storage providers.

on a configuration recommended for evaluating database I/O performance [38]. We ran five instances of `fio` for 30 minutes. Each instance was a separate Pod and was attached to a different volume provisioned from the same storage provider. During the 30 minute run, we cycled through each of the five volumes, taking snapshots at regular intervals: every 90 seconds for the 20 snapshot test; and every 45 seconds for the 40 snapshot test. An individual volume therefore was snapshotted every 7.5 and 3.75 minutes for the 20- and 40-snapshot tests, respectively.

We configured `fio` to log I/O operation latencies. Figure 2 shows how these latencies were affected when a different number of snapshots were taken during the test. Again, the difference in storage providers is apparent. Although Gluster and Ceph are minimally affected, OpenEBS is affected more significantly. For example, when 20 snapshots are taken during `fio`'s run, OpenEBS's 99.9^{th} percentile read-latency grows 3.3× compared to the baseline, and when 40 snapshots are taken it grows to 17×.

Together, the experiments show that (1) storage providers differ in how they perform and scale with frequent control operations and (2) control operations do influence the I/O latency for workloads running in neighboring containers. Both factors impact the end users of cloud services directly and indirectly. Workflows that rely on creating new containers frequently—e.g., Continuous Integration (CI) services—can be impacted directly by the performance of control operations such as volume creation and attachment. Additionally, the overhead caused by control operations can indirectly reduce the user-visible performance of applications [31]. Understanding these behaviors allows cloud administrators to make informed decisions when choosing storage solutions.

4 Benchmark Design

We believe that the aforementioned trend analysis and exploratory experiments justify the need to create a benchmark that treats control operations as first-class citizens.

Such a benchmark would be useful to anyone making design or operational decisions regarding their cloud-based services. For instance, an administrator could use the benchmark to examine the relationship between the number of containers deployed concurrently and their startup times. We first discuss the design requirements for a cloud native storage benchmark and then present our initial design.

Requirements. We identify 9 core requirements a cloud native storage benchmark should address. The requirements cover aspects of workload generation (W), result measurement and visualization (R), and usability (U).

■**W1:** The workload of a cloud native storage benchmark is two-dimensional as it needs to evaluate both the performance of control operations and their effect on I/O operations. Hence, the benchmark should generate a mix of both operation types. Each dimension should be separately controllable to allow measuring the performance of control operations in isolation and in combination with I/O operations, under different loads.

■**W2:** The benchmark's workload description should allow users to specify the type and frequency of control operations and the amount of work performed. In addition to straightforward control operations (volume creation, snapshot deletion, etc.), the benchmark should also trigger aborting a running I/O workload in order to evaluate fault recovery performance. Generating the faults of the storage system itself, however, is out of scope for the benchmark, but tools such as Chaos Monkey [5] could be used for that. It should also be possible to compose different control-operation patterns to generate more complex and realistic workloads.

■**W3:** I/O operations can be generated by a variety of existing sources (e.g., `fio` or real-world applications). The benchmark has to support these different sources using a pluggable architecture. The benchmark should also ship with default sources that can be used without additional configuration efforts.

■**W4:** The target environment of the benchmark are cloud native setups, which normally host a large number of different tenants, isolated through different QoS mechanisms. The benchmark should allow to evaluate the quality and reliability of the mechanisms used for storage I/O isolation. In particular, the benchmark needs to enable users to configure QoS targets to detect potential QoS violations.

■**R1:** As stated in W3, I/O operations can come from a variety of sources. Hence, performance measurements of I/O operations should be decoupled from their workload generation, to be able to capture basic metrics (e.g., IOPS, bandwidth, and latency) for any source. As cloud native environments often already contain elaborate monitoring tools such as Prometheus [20], the benchmark should integrate with those tools. It should also allow one to easily export workload-specific metrics, such as transactions-per-second in a DB benchmark, to specific analysis tools.

■**R2:** A single run of the benchmark can produce a large amount of measurement data, coming from a variety of different sources at different granularities (cluster nodes, Pods, PVs, etc.). In its raw form, this data will be hard to interpret for end users due

to its variety. The benchmark needs to aggregate all the results and present them in a comprehensive, clear, and actionable way.

■**R3:** Cloud native clusters can have thousands of nodes [3]. Hence, the benchmark should scale to large clusters while keeping the overhead of metrics collection low; the benchmark should utilize the deployment management and monitoring capabilities offered by cloud native environments. Experimentation will be needed to measure the overhead and find an acceptable threshold. For large clusters it may be necessary to tune the number and level of detail of the metrics collected, in order to keep the overhead within the desired threshold.

■**U1:** The benchmark should support reproducibility as a first-class citizen to allow easy comparison of different results. To support reproducibility, the benchmark has to collect and store enough information on the experiment, the cluster, and the storage configuration. Experimentation will be needed to determine what information is necessary to collect to ensure reproducibility. To avoid any external dependencies, collection should be restricted to information that can be retrieved natively from the environment (e.g., through Kubernetes's API server).

■**U2:** The benchmark should be easy to deploy and use. While this is a general requirement for any benchmark, it is especially important in this context due to the scale and complexity of cloud native environments. This means the benchmark needs to package and deploy all required dependencies, seamlessly integrate with the platform by relying on available primitives (e.g., Operators and DaemonSets in Kubernetes), and allow for simple workload descriptions in accepted formats (e.g., YAML).

Although there are many popular I/O benchmarks such as `fio` or `filebench`, these tools lack the capabilities and infrastructure necessary to orchestrate running multiple workloads from different sources while also executing control operations. Moreover, we do not view orchestration and control operations as a simple extension to capabilities of I/O storage benchmarks. Further, requirement W3 necessitates being able to use a variety of existing benchmarks to generate I/O workloads. Therefore, we believe a *new* benchmark is needed that includes orchestration, control operations, and support for a variety of existing I/O benchmarks to generate I/O operations.

Design. Our preliminary design addresses some of the above requirements (W1–W3, R1, R3, and U2); we plan to address the remaining ones (W4, R2, and U1) in the final implementation. For the purpose of illustration, we describe our design in terms of a possible implementation for Kubernetes. However, this does not mean that the design is applicable only to Kubernetes; the benchmark could be implemented for any cloud platform that can be interfaced with through an API (e.g., OpenStack or VMWare ESX).

Our design is well suited for Kubernetes' *operator design pattern* [18]. This pattern has two components: (i) a custom user object and (ii) a controller that watches object modifications and acts accordingly [18].

The *benchmark controller* ❶ is the centerpiece of our cloud native storage benchmark (see Figure 3). The controller runs in

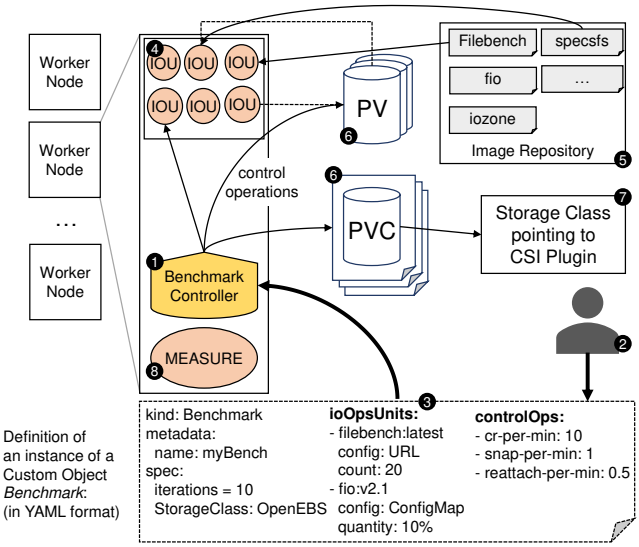

Figure 3: Proposed high-level system design

one or more Pods and subscribes to Kubernetes for the creation, deletion, and modification of *Benchmark* objects. A Benchmark is a custom object type [9] defined in Kubernetes during the installation of the benchmark. To run a benchmark, a user ❷ creates an instance of a Benchmark type, defined as per Kubernetes convention, in YAML format ❸. This instance represents one or more runs of the same workload. Using existing Kubernetes constructs in the benchmark design addresses requirements R3 and U2.

To meet requirements W1–W3, workloads are defined in two parts: (i) `ioOpsUnits`, which describe the characteristics of the I/O operations to run; and (ii) `controlOps`, which describe the control operation characteristics. I/O operations are represented as a collection of I/O units—IOUs ❹—each of which corresponds to a single Pod running some I/O workload consisting of both data and metadata operations. IOU Pods start from container images; we plan to provide a curated set of IOU images (e.g., filebench, fio, iozone, specSFS) in a public image registry like Docker Hub ❺. Users will also be able to specify their own IOU images.

Control operations are specified in the benchmark as part of the `controlOps` section. These operations are directly executed by the controller on PVCs and their corresponding PVs ❻ and Pods. For example, the controller may periodically create several new PVCs, snapshot some PVs, and reschedule IOU Pods to trigger PV reattachment. To fully understand the extent to which users will want to customize their control workloads, more analysis and user feedback is necessary.

When the controller receives a Benchmark definition, it creates the specified number of Pods. Kubernetes schedules the IOU Pods in the cluster, pulls the necessary images, and starts the Pods. Benchmarks can typically generate a number of different IOUs, configurable with parameters. Users can specify IOU-specific configurations in the benchmark definition through a ConfigMap [6] or a URL to a config file. PVs used by the benchmark are created in the Storage Class ❼ listed by a user

in the Benchmark object definition. This allows one to specify which storage provider (e.g., OpenEBS) to benchmark.

Besides the workload, a separate Pod will be running to collect the system performance, resource utilization, and performance numbers reported by benchmarks ❽. We plan to utilize the ELK stack [28, 29] to analyze and visualize the collected measurements. This addresses R1.

5 Related Work

The need to test the performance of cloud storage has motivated academia and industry to develop several micro-benchmarks such as YCSB [26] and COSBench [39]. 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. Unlike YCSB, COSBench targets more generic workloads and is not limited to key-value or object storage.

TailBench [30] 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 [27] 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.

Traeger et al. [36] conducted an extensive study of file systems and storage benchmarks. Yet, we are not aware of any studies that focused on benchmarking control operations in cloud native storage systems—this position paper's focus.

6 Conclusion

There are many, diverse cloud native storage solutions but their real-world performance is poorly understood. This is largely due to the shortcomings of existing benchmarks, which are not able to generate control operations. Control operations are an essential part of the cloud native storage workflow and are becoming increasingly more frequent as regular, non-privileged users are able to issue control operations without involving a storage administrator. Therefore, we argue that it is essential for any cloud native storage benchmark to treat control operations as a first-class citizen. In this position paper, we demonstrated this need with two sample studies; we show that control operations can impact the I/O workloads of containers significantly and that they result in large performance variations across different solutions. We presented a set of requirements for cloud native storage benchmarks and an initial design to address those challenges and get one step closer to effectively benchmarking cloud native storage systems.

To implement this design as a *practical* benchmark, community input will be critical. Discussions with academics and professionals will help reach the optimal level of versatility, expressiveness, realism, and ease-of-use. Therefore, we envision the benchmark as an open-source project with a community built around it. In addition, as current storage traces lack control operations, we hope to find partners that can share traces or aggregated statistics from their environments with control operations included. This will be important in developing realistic control workloads.

References

[1] All SPEC SFS 2014 Results Published by SPEC. https://www.spec.org/sfs2014/results/sfs2014.html.

[2] Apache Mesos. https://mesos.apache.org.

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

[4] Ceph. https://ceph.io/.

[5] Chaos monkey. https://github.com/Netflix/chaosmonkey.

[6] Configure a Pod to Use a ConfigMap. https://bit.ly/2Jgx97S.

[7] Container Storage Interface (CSI) Specification. https://bit.ly/3bqQX4b.

[8] cStor. https://docs.openebs.io/docs/next/cstor.html.

[9] Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[10] Docker Swarm. https://github.com/docker/swarm.

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

[12] fio. https://github.com/axboe/fio.

[13] Gluster. https://www.gluster.org/.

[14] Kubernetes. https://kubernetes.io/.

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

[16] NetApp Trident. https://github.com/NetApp/trident.

[17] OpenEBS. https://openebs.io/.

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

[19] Portworx. https://portworx.com/.

[20] Prometheus. https://prometheus.io/.

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

[22] StorageOS. https://storageos.com/.

[23] Sysdig 2019 Container Usage Report. https://sysdig.com/blog/sysdig-2019-container-usage-report/.

[24] The IBM Spectrum Scale Container Storage Interface (CSI) project. https://github.com/IBM/ibm-spectrum-scale-csi.

[25] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[26] 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.

[27] 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.

[28] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide: A Distributed Real-time Search and Analytics Engine*. O'Reilly Media, Inc., 2015.

[29] Yuvraj Gupta. *Kibana Essentials*. Packt Publishing Ltd, 2015.

[30] 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.

[31] Pulkit Misra, Maria Borge, Inigo Goiri, Alvin 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.

[32] 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.

[33] Vasily Tarasov, Santhosh Kumar, Jack Ma, Dean Hildebrand, Anna Povzner, Geoff Kuenning, and Erez Zadok. Extracting Flexible, Replayable Models from Large Block Traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

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

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

[36] 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.

[37] 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.

[38] Mark Wong. Filesystem Performance from a Database Perspective, 2009. `https://bit.ly/33M5RiU`.

[39] 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.